Invoke

Release

Sep 30, 2022

Contents

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

CHAPTER 1

Getting started

Many core ideas & API calls are explained in the tutorial/getting-started document:

1.1 Getting started

This document presents a whirlwind tour of Invoke's feature set. Please see the links throughout for detailed conceptual & API docs. For installation help, see the project's installation page.

1.1.1 Defining and running task functions

The core use case for Invoke is setting up a collection of task functions and executing them. This is pretty easy – all you need is to make a file called `tasks.py` importing the *task* decorator and decorating one or more functions. You will also need to add an arbitrarily-named context argument (convention is to use `c`, `ctx` or `context`) as the first positional arg. Don't worry about using this context parameter yet.

Let's start with a dummy Sphinx docs building task:

```
from invoke import task

@task
def build(c):
    print("Building!")
```

You can then execute that new task by telling Invoke's command line runner, `invoke`, that you want it to run:

```
$ invoke build
Building!
```

The function body can be any Python you want – anything at all.

1.1.2 Task parameters

Functions can have arguments, and thus so can tasks. By default, your task functions' args/kwargs are mapped automatically to both long and short CLI flags, as per *the CLI docs*. For example, if we add a `clean` argument and give it a boolean default, it will show up as a set of toggle flags, `--clean` and `-c`:

```
@task
def build(c, clean=False):
    if clean:
        print("Cleaning!")
    print("Building!")
```

Invocations:

```
$ invoke build -c
$ invoke build --clean
```

Naturally, other default argument values will allow giving string or integer values. Arguments with no default values are assumed to take strings, and can also be given as positional arguments. Take this incredibly contrived snippet for example:

```
@task
def hi(c, name):
    print("Hi {}!".format(name))
```

It can be invoked in the following ways, all resulting in "Hi Name!":

```
$ invoke hi Name
$ invoke hi --name Name
$ invoke hi --name=Name
$ invoke hi -n Name
$ invoke hi -nName
```

Adding metadata via `@task`

`@task` can be used without any arguments, as above, but it's also a convenient vector for additional metadata about the task function it decorates. One common example is describing the task's arguments, via the `help` parameter (in addition to optionally giving task-level help via the docstring):

```
@task(help={'name': "Name of the person to say hi to."})
def hi(c, name):
    """
    Say hi to someone.
    """
    print("Hi {}!".format(name))
```

This description will show up when invoking `--help`:

```
$ invoke --help hi
Usage: inv[oke] [--core-opts] hi [--options] [other tasks here ...]

Docstring:
  Say hi to someone.

Options:
  -n STRING, --name=STRING   Name of the person to say hi to.
```

More details on task parameterization and metadata can be found in *Invoking tasks* (for the command-line & parsing side of things) and in the `task` API documentation (for the declaration side).

1.1.3 Listing tasks

You'll sometimes want to see what tasks are available in a given `tasks.py` – `invoke` can be told to list them instead of executing something:

```
$ invoke --list
Available tasks:

    build
```

This will also print the first line of each task's docstring, if it has one. To see what else is available besides `--list`, say `invoke --help`.

1.1.4 Running shell commands

Many use cases for Invoke involve running local shell commands, similar to programs like Make or Rake. This is done via the `run` function:

```
from invoke import task

@task
def build(c):
    c.run("sphinx-build docs docs/_build")
```

You'll see the command's output in your terminal as it runs:

```
$ invoke build
Running Sphinx v1.1.3
loading pickled environment... done
...
build succeeded, 2 warnings.
```

`run` has a number of arguments controlling its behavior, such as activation of pseudo-terminals for complex programs requiring them, suppression of exit-on-error behavior, hiding of subprocess' output (while still capturing it for later review), and more. See `its API docs` for details.

`run` always returns a useful `Result` object providing access to the captured output, exit code, and other information.

Aside: what exactly is this 'context' arg anyway?

A common problem task runners face is transmission of "global" data - values loaded from *configuration files* or *other configuration vectors*, given via CLI flags, generated in 'setup' tasks, etc.

Some libraries (such as Fabric 1.x) implement this via module-level attributes, which makes testing difficult and error prone, limits concurrency, and increases implementation complexity.

Invoke encapsulates state in explicit `Context` objects, handed to tasks when they execute . The context is the primary API endpoint, offering methods which honor the current state (such as `Context.run`) as well as access to that state itself.

1.1.5 Declaring pre-tasks

Tasks may be configured in a number of ways via the *task* decorator. One of these is to select one or more other tasks you wish to always run prior to execution of your task, indicated by name.

Let's expand our docs builder with a new cleanup task that runs before every build (but which, of course, can still be executed on its own):

```
from invoke import task

@task
def clean(c):
    c.run("rm -rf docs/_build")

@task(clean)
def build(c):
    c.run("sphinx-build docs docs/_build")
```

Now when you `invoke build`, it will automatically run `clean` first.

Note: If you're not a fan of the implicit "positional arguments are pre-run task names" API, you can instead explicitly give the `pre` kwarg: `@task(pre=[clean])`.

Details can be found in *How tasks run*.

1.1.6 Creating namespaces

Right now, our `tasks.py` is implicitly for documentation only, but maybe our project needs other non-doc things, like packaging/deploying, testing, etc. At that point, a single flat namespace isn't enough, so Invoke lets you easily build a *nested namespace*. Here's a quick example.

Let's first rename our `tasks.py` to be `docs.py`; no other changes are needed there. Then we create a new `tasks.py`, and for the sake of brevity populate it with a new, truly top level task called `deploy`.

Finally, we can use a new API member, the *Collection* class, to bind this task and the `docs` module into a single explicit namespace. When Invoke loads your task module, if a *Collection* object bound as `ns` or `namespace` exists it will get used for the root namespace:

```
from invoke import Collection, task
import docs

@task
def deploy(c):
    c.run("python setup.py sdist")
    c.run("twine upload dist/*")

namespace = Collection(docs, deploy)
```

The result:

```
$ invoke --list
Available tasks:

    deploy
    docs.build
    docs.clean
```

For a more detailed breakdown of how namespacing works, please see *the docs*.

CHAPTER 2

The `invoke` CLI tool

Details on the CLI interface to Invoke, available core flags, and tab completion options.

2.1 `inv[oke]` core usage

See also:

This page documents `invoke`'s core arguments, options and behavior (which includes options present in *custom Invoke-based binaries*). For details on invoking user-specified tasks and other parser-related details, see *Invoking tasks*.

2.1.1 Core options and flags

`invoke`'s usage looks like:

```
$ inv[oke] [--core-opts] task1 [--task1-opts] ... taskN [--taskN-opts]
```

All core options & flags are below; almost all of them must be given *before* any task names, with a few (such as *`--help`*) being specially looked for anywhere in the command line. (For parsing details, see *Basic command line layout*.)

`--complete`

Print (line-separated) valid tab-completion options for an Invoke command line given as the 'remainder' (i.e. after a `--`). Used for building *shell completion scripts*.

For example, when the local tasks tree contains tasks named `foo` and `bar`, and when `foo` takes flags `--foo-arg` and `--foo-arg-2`, you might use it like this:

```
# Empty input: just task names
$ inv --complete --
foo
bar
```

```
# Input not ending with a dash: task names still
$ inv --complete -- foo --foo-arg
foo
bar

# Input ending with a dash: current context's flag names
$ inv --complete -- foo -
--foo-arg
--foo-arg-2
```

For more details on how to make best use of this option, see *`--print-completion-script`*.

`--hide``=STRING`
Set default value of run()'s 'hide' kwarg.

`--no-dedupe`
Disable task deduplication.

`--print-completion-script``=SHELL`
Print a completion script for desired `SHELL` (e.g. `bash`, `zsh`, etc). This can be sourced into the current session in order to enjoy *tab-completion for tasks and options*.

These scripts are bundled with Invoke's distributed codebase, and internally make use of *`--complete`*.

`--prompt-for-sudo-password`
Prompt at the start of the session (before executing any tasks) for the `sudo.password` configuration value. This allows users who don't want to keep sensitive material in the config system or their shell environment to rely on user input, without otherwise interrupting the flow of the program.

`--write-pyc`
By default, Invoke disables bytecode caching as it can cause hard-to-debug problems with task files and (for the kinds of things Invoke is typically used for) offers no noticeable speed benefit. If you really want your `.pyc` files back, give this option.

`-c` `STRING`, **`--collection`**`=STRING`
Specify collection name to load.

`-d`, **`--debug`**
Enable debug output.

`--dry`
Echo commands instead of actually running them; specifically, causes any `run` calls to:

- Act as if the `echo` option has been turned on, printing the command-to-be-run to stdout;
- Skip actual subprocess invocation (returning before any of that machinery starts running);
- Return a dummy *`Result`* object with 'blank' values (empty stdout/err strings, `0` exit code, etc).

`-D`, **`--list-depth`**`=INT`
Limit *`--list`* display to the specified number of levels, e.g. `--list-depth 1` to show only top-level tasks and namespaces.

If an argument is given to `--list`, then this depth is relative; so `--list build --list-depth 1` shows everything at the top level of the `build` subtree.

Default behavior if this is not given is to show all levels of the entire task tree.

`-e`, **`--echo`**
Echo executed commands before running.

`-f`, **`--config`**
Specify a *runtime configuration file* to load.

Note that you may instead use the `INVOKE_RUNTIME_CONFIG` environment variable in place of this option. If both are given, the CLI option will win out.

-F, --list-format=STRING
Change the format used to display the output of `--list`; may be one of:

- `flat` (the default): single, flat vertical list with dotted task names.
- `nested`: a nested (4-space indented) vertical list, where each level implicitly includes its parent (with leading dots as a strong visual clue that these are still subcollection tasks.)
- `json`: intended for consumption by scripts or other programs, this format emits JSON representing the task tree, with each 'node' in the tree (the outermost document being the root node, and thus a JSON object) consisting of the following keys:
    - `name`: String name of collection; for the root collection this is typically the module name, so unless you're supplying alternate collection name to the load process, it's usually `"tasks"` (from `tasks.py`.)
    - `help`: First line of collection's docstring, if it came from a module; null otherwise (or if module lacked a docstring.)
    - `tasks`: Immediate children of this collection; an array of objects of the following form:
        * `name`: Task's local name within its collection (i.e. not the full dotted path you might see with the `flat` format; reconstructing that path is left up to the consumer.)
        * `help`: First line of task's docstring, or null if it had none.
        * `aliases`: An array of string aliases for this task.
    - `default`: String naming which task within this collection, if any, is the default task. Is null if no task is the default.
    - `collections`: An array of any sub-collections within this collection, members of which which will have the same structure as this outermost document, recursively.

  The JSON emitted is not pretty-printed, but does end with a trailing newline.

-h STRING, **--help**=STRING
When given without any task names, displays core help; when given with a task name (may come before *or* after the task name) displays help for that particular task.

-l, --list=STRING
List available tasks. Shows all tasks by default; may give an explicit namespace to 'root' the displayed task tree to only that namespace. (This argument may contain periods, as with task names, so it's possible to show only a small, deep portion of the overall tree if desired.)

-p, --pty
Use a pty when executing shell commands.

-r STRING, **--search-root**=STRING
Change root directory used for finding task modules.

-T INT, **--command-timeout**=INT
Set a default command execution timeout of INT seconds. Maps to the `timeouts.command` config setting.

-V, --version
Show version and exit.

-w, --warn-only
Warn, instead of failing, when shell commands fail.

2.1.2 Shell tab completion

Generating a completion script

Invoke's philosophy is to implement generic APIs and then "bake in" a few common use cases built on top of those APIs; tab completion is no different. Generic tab completion functionality (outputting a shell-compatible list of completion tokens for a given command line context) is provided by the `--complete` core CLI option described above.

However, you probably won't need to use that flag yourself: we distribute a handful of ready-made wrapper scripts aimed at the most common shells like `bash` and `zsh` (plus others). These scripts can be automatically generated from Invoke or *any Invoke-driven command-line tool*, using `--print-completion-script`; the printed scripts will contain the correct binary name(s) for the program generating them.

For example, the following command prints (to stdout) a script which works for `zsh`, instructs `zsh` to use it for the `inv` and `invoke` programs, and calls `invoke --complete` at runtime to get dynamic completion information:

```
$ invoke --print-completion-script zsh
```

Note: You'll probably want to source this command or store its output somewhere permanently; more on that in the next section.

Similarly, the Fabric tool inherits from Invoke, and only has a single binary name (`fab`); if you wanted to get Fabric completion in `bash`, you would say:

```
$ fab --print-completion-script bash
```

In the rest of this section, we'll use `inv` in examples, but please remember to replace it with the program you're actually using, if it's not Invoke itself!

Sourcing the script

There are a few ways to utilize the output of the above commands, depending on your needs, where the program is installed, and your shell:

- The simplest and least disruptive method is to `source` the printed completion script inline, which doesn't place anything on disk, and will only affect the current shell session:

  ```
  $ source <(inv --print-completion-script zsh)
  ```

- If you've got the program available in your system's global Python interpreter (and you're okay with running the program at the startup of each shell session - Python's speed is admittedly not its strong point) you could add that snippet to your shell's startup file, such as `~/.zshrc` or `~/.bashrc`.
- If the program's available globally but you'd prefer to *avoid* running an extra Python program at shell startup, you can cache the output of the command in its own file; where this file lives is entirely up to you and how your shell is configured. For example, you might just drop it into your home directory as a hidden file:

  ```
  $ inv --print-completion-script zsh > ~/.invoke-completion.sh
  ```

  and then perhaps add the following to the end of `~/.zshrc`:

  ```
  source ~/.invoke-completion.sh
  ```

But again, this is entirely up to you and your shell.

Note: If you're using `fish`, you *must* use this tactic, as our fish completion script is not suitable for direct sourcing. Fish shell users should direct the output of the command to a file in the `~/.config/fish/completions/` directory.

- Finally, if your copy of the needing-completion program is only installed in a specific environment like a virtualenv, you can use either of the above techniques:
  - Caching the output and referencing it in a global shell startup file will still work in this case, as it does not require the program to be available when the shell loads – only when you actually attempt to tab complete.
  - Using the `source <(inv --print-completion-script yourshell)` approach will work *as long as* you place it in some appropriate per-environment startup file, which will vary depending on how you manage Python environments. For example, if you use `virtualenvwrapper`, you could append the `source` line in `/path/to/virtualenv/bin/postactivate`.

Utilizing tab completion itself

You've ensured that the completion script is active in your environment - what have you gained?

- By default, tabbing after typing `inv` or `invoke` will display task names from your current directory/project's tasks file.
- Tabbing after typing a dash (`-`) or double dash (`--`) will display valid options/flags for the current context: core Invoke options if no task names have been typed yet; options for the most recently typed task otherwise.
  - Tabbing while typing a partial long option will complete matching long options, using your shell's native substring completion. E.g. if no task names have been typed yet, `--e<tab>` will offer `--echo` as a completion option.
- Hitting tab when the most recent typed/completed token is a flag which takes a value, will 'fall through' to your shell's native filename completion.
  - For example, prior to typing a task name, `--config <tab>` will complete local file paths to assist in filling in a config file.

CHAPTER 3

Concepts

Dig deeper into specific topics:

3.1 Configuration

3.1.1 Introduction

Invoke offers a multifaceted configuration mechanism allowing you to configure both core behavior and that of your tasks, via a hierarchy of configuration files, environment variables, *task namespaces* and CLI flags.

The end result of configuration seeking, loading, parsing & merging is a `Config` object, which behaves like a (nested) Python dictionary. Invoke references this object when it runs (determining the default behavior of methods like `Context.run`) and exposes it to users' tasks as `Context.config` or as shorthand attribute access on the `Context` itself.

3.1.2 The configuration hierarchy

In brief, the order in which configuration values override one another is as follows:

1. **Internal default values** for behaviors which are controllable via configuration. See *Default configuration values* for details.
2. **Collection-driven configurations** defined in tasks modules via `Collection.configure`. (See *Collection-based configuration* below for details.)
   - Sub-collections' configurations get merged into the top level collection and the final result forms the basis of the overall configuration setup.
3. **System-level configuration file** stored in `/etc/`, such as `/etc/invoke.yaml`. (See *Configuration files* for details on this and the other config-file entries.)
4. **User-level configuration file** found in the running user's home directory, e.g. `~/.invoke.yaml`.

5. **Project-level configuration file** living next to your top level `tasks.py`. For example, if your run of Invoke loads `/home/user/myproject/tasks.py` (see our docs on *the load process*), this might be `/home/user/myproject/invoke.yaml`.
6. **Environment variables** found in the invoking shell environment.
   - These aren't as strongly hierarchical as the rest, nor is the shell environment namespace owned wholly by Invoke, so we must rely on slightly verbose prefixing instead - see *Environment variables* for details.
7. **Runtime configuration file** whose path is given to `-f`, e.g. `inv -f /random/path/to/config_file.yaml`. This path may also be set via the `INVOKE_RUNTIME_CONFIG` env var.
8. **Command-line flags** for certain core settings, such as `-e`.
9. **Modifications made by user code** at runtime.

3.1.3 Default configuration values

Below is a list of all the configuration values and/or section Invoke itself uses to control behaviors such as `Context.run`'s `echo` and `pty` flags, task deduplication, and so forth.

Note: The storage location for these values is inside the `Config` class, specifically as the return value of `Config.global_defaults`; see its API docs for more details.

For convenience, we refer to nested setting names with a dotted syntax, so e.g. `foo.bar` refers to what would be (in a Python config context) `{'foo': {'bar': <value here>}}`. Typically, these can be read or set on `Config` and `Context` objects using attribute syntax, which looks nearly identical: `c.foo.bar`.

- The `tasks` config tree holds settings relating to task execution.
  - `tasks.dedupe` controls *Task deduplication* and defaults to `True`. It can also be overridden at runtime via `--no-dedupe`.
  - `tasks.auto_dash_names` controls whether task and collection names have underscores turned to dashes on the CLI. Default: `True`. See also *Dashes vs underscores*.
  - `tasks.collection_name` controls the Python import name sought out by *collection discovery*, and defaults to `"tasks"`.
  - `tasks.executor_class` allows users to override the class instantiated and used for task execution.

    Must be a fully-qualified dotted path of the form `module(.submodule...).class`, where all but `.class` will be handed to `importlib.import_module`, and `class` is expected to be an attribute on that resulting module object.

    Defaults to `None`, meaning to use the running `Program` object's `executor_class` attribute.

    Warning: Take care if using this setting in tandem with *custom program binaries*, since custom programs may specify their own default executor class (which your use of this setting will override!) and assume certain behaviors stemming from that.
  - `tasks.ignore_unknown_help` (default: `False`) lets users disable "help keys were supplied for nonexistent arguments" errors. Normally, Invoke assumes such a situation implies a typo in the `help` argument to `@task`, but sometimes users have good reasons for this.
  - `tasks.search_root` allows overriding the default *collection discovery* root search location. It defaults to `None`, which indicates to use the executing process' current working directory.

- The `run` tree controls the behavior of *`Runner.run`*. Each member of this tree (such as `run.echo` or `run.pty`) maps directly to a *`Runner.run`* keyword argument of the same name; see that method's docstring for details on what these settings do & what their default values are.
- The `runners` tree controls _which_ runner classes map to which execution contexts; if you're using Invoke by itself, this will only tend to have a single member, `runners.local`. Client libraries may extend it with additional key/value pairs, such as `runners.remote`.
- The `sudo` tree controls the behavior of *`Context.sudo`*:
  - `sudo.password` controls the autoresponse password submitted to sudo's password prompt. Default: `None`.

    Warning: While it's possible to store this setting, like any other, in *configuration files* – doing so is inherently insecure. We highly recommend filling this config value in at runtime from a secrets management system of some kind.

  - `sudo.prompt` holds the sudo password prompt text, which is both supplied to `sudo -p`, and searched for when performing *auto-response*. Default: `[sudo] password:`.
- A top level config setting, `debug`, controls whether debug-level output is logged; it defaults to `False`.

  `debug` can be toggled via the *`-d`* CLI flag, which enables debugging after CLI parsing runs. It can also be toggled via the `INVOKE_DEBUG` environment variable which - unlike regular env vars - is honored from the start of execution and is thus useful for troubleshooting parsing and/or config loading.
- A small config tree, `timeouts`, holds various kinds of timeout controls. At present, for Invoke, this only holds a `command` subkey, which controls subprocess execution timeouts.
  - Client code often adds more to this tree, and Invoke itself may add more in the future as well.

3.1.4 Configuration files

Loading

For each configuration file location mentioned in the previous section, we search for files ending in `.yaml`, `.yml`, `.json` or `.py` (**in that order!**), load the first one we find, and ignore any others that might exist.

For example, if Invoke is run on a system containing both `/etc/invoke.yml` *and* `/etc/invoke.json`, **only the YAML file will be loaded**. This helps keep things simple, both conceptually and in the implementation.

Format

Invoke's configuration allows arbitrary nesting, and thus so do our config file formats. All three of the below examples result in a configuration equivalent to `{'debug': True, 'run': {'echo': True}}`:

- **YAML**

```yaml
debug: true
run:
    echo: true
```

- **JSON**

```
{
    "debug": true,
    "run": {
        "echo": true
    }
}
```

- **Python**:

```
debug = True
run = {
    "echo": True
}
```

For further details, see these languages' own documentation.

3.1.5 Environment variables

Environment variables are a bit different from other configuration-setting methods, since they don't provide a clean way to nest configuration keys, and are also implicitly shared amongst the entire system's installed application base.

In addition, due to implementation concerns, env vars must be pre-determined by the levels below them in the config hierarchy (in other words - env vars may only be used to override existing config values). If you need Invoke to understand a `FOOBAR` environment variable, you must first declare a `foobar` setting in a configuration file or in your task collections.

Basic rules

To mitigate the shell namespace problem, we simply prefix all our env vars with `INVOKE_`.

Nesting is performed via underscore separation, so a setting that looks like e.g. `{'run': {'echo': True}}` at the Python level becomes `INVOKE_RUN_ECHO=1` in a typical shell. See *Nesting vs underscored names* below for more on this.

Type casting

Since env vars can only be used to override existing settings, the previous value of a given setting is used as a guide in casting the strings we get back from the shell:

- If the current value is a string or Unicode object, it is replaced with the value from the environment, with no casting whatsoever;
  - Depending on interpreter and environment, this means that a setting defaulting to a non-Unicode string type (eg a `str` on Python 2) may end up replaced with a Unicode string, or vice versa. This is intentional as it prevents users from accidentally limiting themselves to non-Unicode strings.
- If the current value is `None`, it too is replaced with the string from the environment;
- Booleans are set as follows: `0` and the empty value/string (e.g. `SETTING=`, or `unset SETTING`, or etc) evaluate to `False`, and any other value evaluates to `True`.
- Lists and tuples are currently unsupported and will raise an exception;
  - In the future we may implement convenience transformations, such as splitting on commas to form a list; however since users can always perform such operations themselves, it may not be a high priority.
- All other types - integers, longs, floats, etc - are simply used as constructors for the incoming value.

  - For example, a `foobar` setting whose default value is the integer `1` will run all env var inputs through `int`, and thus `FOOBAR=5` will result in the Python value `5`, not `"5"`.

Nesting vs underscored names

Since environment variable keys are single strings, we must use some form of string parsing to allow access to nested configuration settings. As mentioned above, in basic use cases this just means using an underscore character: `{'run': {'echo': True}}` becomes `INVOKE_RUN_ECHO=1`.

However, ambiguity is introduced when the settings names themselves contain underscores: is `INVOKE_FOO_BAR=baz` equivalent to `{'foo': {'bar': 'baz'}}`, or to `{'foo_bar': 'baz'}`? Thankfully, because env vars can only be used to modify settings declared at the Python level or in config files, we look at the current state of the config to determine the answer.

There is still a corner case where *both* possible interpretations exist as valid config paths (e.g. `{'foo': {'bar': 'default'}, 'foo_bar': 'otherdefault'}`). In this situation, we honor the Zen of Python and refuse to guess; an error is raised instead, counseling users to modify their configuration layout or avoid using env vars for the setting in question.

3.1.6 `Collection`-based configuration

`Collection` objects may contain a config mapping, set via *`Collection.configure`*, and (as per *the hierarchy*) this typically forms the lowest level of configuration in the system.

When collections are *nested*, configuration is merged 'downwards' by default: when conflicts arise, outer namespaces closer to the root will win, versus inner ones closer to the task being invoked.

Note: 'Inner' tasks here are specifically those on the path from the root to the one housing the invoked task. 'Sibling' subcollections are ignored.

A quick example of what this means:

```python
from invoke import Collection, task

# This task & collection could just as easily come from
# another module somewhere.
@task
def mytask(c):
    print(c['conflicted'])
inner = Collection('inner', mytask)
inner.configure({'conflicted': 'default value'})

# Our project's root namespace.
ns = Collection(inner)
ns.configure({'conflicted': 'override value'})
```

The result of calling `inner.mytask`:

```
$ inv inner.mytask
override value
```

3.1.7 Example of real-world config use

The previous sections had small examples within them; this section provides a more realistic-looking set of examples showing how the config system works.

Setup

We'll start out with semi-realistic tasks that hardcode their values, and build up to using the various configuration mechanisms. A small module for building Sphinx docs might begin like this:

```
from invoke import task

@task
def clean(c):
    c.run("rm -rf docs/_build")

@task
def build(c):
    c.run("sphinx-build docs docs/_build")
```

Then maybe you refactor the build target:

```
target = "docs/_build"

@task
def clean(c):
    c.run("rm -rf {}".format(target))

@task
def build(c):
    c.run("sphinx-build docs {}".format(target))
```

We can also allow runtime parameterization:

```
default_target = "docs/_build"

@task
def clean(c, target=default_target):
    c.run("rm -rf {}".format(target))

@task
def build(c, target=default_target):
    c.run("sphinx-build docs {}".format(target))
```

This task module works for a single set of users, but what if we want to allow reuse? Somebody may want to use this module with a different default target. Using the configuration data (made available via the context arg) to configure these settings is usually the better solution[1].

Configuring via task collection

The configuration *`setting`* and *`getting`* APIs enable moving otherwise 'hardcoded' default values into a config structure which downstream users are free to redefine. Let's apply this to our example. First we add an explicit namespace object:

[1] Copying and modifying the file breaks code reuse; overriding the module-level `default_path` variable won't play well with concurrency; wrapping the tasks with different default arguments works but is fragile and adds boilerplate.

```python
from invoke import Collection, task

default_target = "docs/_build"

@task
def clean(c, target=default_target):
    c.run("rm -rf {}".format(target))

@task
def build(c, target=default_target):
    c.run("sphinx-build docs {}".format(target))

ns = Collection(clean, build)
```

Then we can move the default build target value into the collection's default configuration, and refer to it via the context. At this point we also change our kwarg default value to be `None` so we can determine whether or not a runtime value was given. The result:

```python
@task
def clean(c, target=None):
    if target is None:
        target = c.sphinx.target
    c.run("rm -rf {}".format(target))

@task
def build(c, target=None):
    if target is None:
        target = c.sphinx.target
    c.run("sphinx-build docs {}".format(target))

ns = Collection(clean, build)
ns.configure({'sphinx': {'target': "docs/_build"}})
```

The result isn't significantly more complex than what we began with, and as we'll see next, it's now trivial for users to override your defaults in various ways.

Configuration overriding

The lowest-level override is, of course, just modifying the local *`Collection`* tree into which a distributed module has been imported. E.g. if the above module is distributed as `myproject.docs`, someone can define a `tasks.py` that does this:

```python
from invoke import Collection, task
from myproject import docs

@task
def mylocaltask(c):
    # Some local stuff goes here
    pass

# Add 'docs' to our local root namespace, plus our own task
ns = Collection(mylocaltask, docs)
```

And then they can add this to the bottom:

```
# Our docs live in 'built_docs', not 'docs/_build'
ns.configure({'sphinx': {'target': "built_docs"}})
```

Now we have a `docs` sub-namespace whose build target defaults to `built_docs` instead of `docs/_build`. Runtime users can still override this via flags (e.g. `inv docs.build --target='some/other/dir'`) just as before.

If you prefer configuration files over in-Python tweaking of your namespace tree, that works just as well; instead of adding the line above to the previous snippet, instead drop this into a file next to `tasks.py` named `invoke.yaml`:

```
sphinx:
    target: built_docs
```

For this example, that sort of local-to-project conf file makes the most sense, but don't forget that the *config hierarchy* offers additional configuration methods which may be suitable depending on your needs.

3.2 Invoking tasks

This page explains how to invoke your tasks on the CLI, both in terms of parser mechanics (how your tasks' arguments are exposed as command-line options) and execution strategies (which tasks actually get run, and in what order).

(For details on Invoke's core flags and options, see *inv[oke] core usage*.)

3.2.1 Basic command line layout

Invoke may be executed as `invoke` (or `inv` for short) and its command line layout looks like this:

```
$ inv [--core-opts] task1 [--task1-opts] ... taskN [--taskN-opts]
```

Put plainly, Invoke's `CLI parser` splits your command line up into multiple "`parser contexts`" which allows it to reason about the args and options it will accept:

- Before any task names are given, the parser is in the "core" parse context, and looks for core options and flags such as `--echo`, `--list` or `--help`.
- Any non-argument-like token (such as `mytask`) causes a switch into a per-task context (or an error, if no task matching that name seems to exist in the *loaded collection*).
- At this point, argument-like tokens are expected to correspond to the arguments for the previously named task (see *Task command-line arguments*).
- Then this cycle repeats infinitely, allowing chained execution of arbitrary numbers of tasks. (In practice, most users only execute one or two at a time.)

For the core arguments and flags, see *inv[oke] core usage*; for details on how your tasks affect the CLI, read onwards.

Note: There is a minor convenience-minded exception to how parse contexts behave: core options *may* also be given inside per-task contexts, *if and only if* there is no conflict with similarly-named/prefixed arguments of the being-parsed task.

For example, `invoke mytask --echo` will behave identically to `invoke --echo mytask`, *unless* `mytask` has its own `echo` flag (in which case that flag is handed to the task context, as normal).

Similarly, `invoke mytask -e` will turn on command echoing too, unless `mytask` has its own argument whose shortflag ends up set to `-e` (e.g. `def mytask(env)`).

3.2.2 Task command-line arguments

The simplest task invocation, for a task requiring no parameterization:

```
$ inv mytask
```

Tasks may take parameters in the form of flag arguments:

```
$ inv build --format=html
$ inv build --format html
$ inv build -f pdf
$ inv build -f=pdf
```

Note that both long and short style flags are supported, and that equals signs are optional in both cases.

Boolean options are simple flags with no arguments:

```
$ inv build --progress-bar
```

Naturally, more than one flag may be given at a time:

```
$ inv build --progress-bar -f pdf
```

Type casting

Natively, a command-line string is just that – a string – requiring some leaps of logic to arrive at any non-string values on the Python end. Invoke has a number of these tricks already at hand, and more will be implemented in the future. Currently:

- Arguments with default values use those default values as a type hint, so `def mytask(c, count=1)` will see `inv mytask --count=5` and result in the Python integer value `5` instead of the string `"5"`.
  - Default values of `None` are effectively the same as having no default value at all - no type casting occurs and you're left with a string.
- The primary exception to the previous rule is booleans: default values of `True` or `False` cause those arguments to show up as actual non-value-taking flags (`--argname` to set the value to `True` if the default was `False`, or `--no-argment` in the opposite case). See *Automatic Boolean inverse flags* for more examples.
- List values (which you wouldn't want to set as an argument's default value anyways – it's a common Python misstep) are served by a special `@task` flag - see *Iterable flag values* below.
- There's currently no way to set other compound values (such as dicts) on the command-line; solving this more complex problem is left as an exercise to the reader (though we may add helpers for such things in the future).

Per-task help / printing available flags

To get help for a specific task, you can give the task name as an argument to the core `--help`/`-h` option, or give `--help`/`-h` after the task (which will trigger custom-to-`help` behavior where the task name itself is given to `--help` as its argument value).

When help is requested, you'll see the task's docstring (if any) and per-argument/flag help output:

```
$ inv --help build  # or invoke build --help

Docstring:
  none

Options for 'build':
  -f STRING, --format=STRING  Which build format type to use
  -p, --progress-bar          Display progress bar
```

Globbed short flags

Boolean short flags may be combined into one flag expression, so that e.g.:

```
$ inv build -qv
```

is equivalent to (and expanded into, during parsing):

```
$ inv build -q -v
```

If the first flag in a globbed short flag token is not a boolean but takes a value, the rest of the glob is taken to be the value instead. E.g.:

```
$ inv build -fpdf
```

is expanded into:

```
$ inv build -f pdf
```

and **not**:

```
$ inv build -f -p -d -f
```

Optional flag values

You saw a hint of this with `--help` specifically, but non-core options may also take optional values, if declared as `optional`. For example, say your task has a `--log` flag that activates logging:

```
$ inv compile --log
```

but you also want it to be configurable regarding *where* to log:

```
$ inv compile --log=foo.log
```

You could implement this with an additional argument (e.g. `--log` and `--log-location`) but sometimes the concise API is the more useful one.

To enable this, specify which arguments are of this 'hybrid' optional-value type inside `@task`:

```
@task(optional=['log'])
def compile(c, log=None):
    if log:
        log_file = '/var/log/my.log'
        # Value was given, vs just-True
        if isinstance(log, unicode):
            log_file = log
        # Replace w/ your actual log setup...
        set_log_destination(log_file)
    # Do things that might log here...
```

When optional flag values are used, the values seen post-parse follow these rules:

- If the flag is not given at all (`invoke compile`) the default value is filled in as normal.
- If it is given with a value (`invoke compile --log=foo.log`) then the value is stored normally.
- If the flag is given with no value (`invoke compile --log`), it is treated as if it were a `bool` and set to `True`.

Resolving ambiguity

There are a number of situations where ambiguity could arise for a flag that takes an optional value:

- When a task takes positional arguments and they haven't all been filled in by the time the parser arrives at the optional-value flag;
- When the token following one of these flags looks like it is itself a flag; or
- When that token has the same name as another task.

In most of these situations, Invoke's parser will refuse the temptation to guess and raise an error.

However, in the case where the ambiguous token is flag-like, the current parse context is checked to resolve the ambiguity:

- If the token is an otherwise legitimate argument, it is assumed that the user meant to give that argument immediately after the current one, and no optional value is set.
  - E.g. in `invoke compile --log --verbose` (assuming `--verbose` is another legit argument for `compile`) the parser decides the user meant to give `--log` without a value, and followed it up with the `--verbose` flag.
- Otherwise, the token is interpreted literally and stored as the value for the current flag.
  - E.g. if `--verbose` is *not* a legitimate argument for `compile`, then `invoke compile --log --verbose` causes the parser to assign `"--verbose"` as the value given to `--log`. (This will probably cause other problems in our contrived use case, but it illustrates our point.)

Iterable flag values

A not-uncommon use case for CLI programs is the desire to build a list of values for a given option, instead of a single value. While this *can* be done via sub-string parsing – e.g. having users invoke a command with `--mylist item1,item2,item3` and splitting on the comma – it's often preferable to specify the option multiple times and store the values in a list (instead of overwriting or erroring.)

In Invoke, this is enabled by hinting to the parser that one or more task arguments are `iterable` in nature (similar to how one specifies `optional` or `positional`):

```
@task(iterable=['my_list'])
def mytask(c, my_list):
    print(my_list)
```

When not given at all, the default value for `my_list` will be an empty list; otherwise, the result is a list, appending each value seen, in order, without any other manipulation (so no deduplication, etc):

```
$ inv mytask
[]
$ inv mytask --my-list foo
['foo']
$ inv mytask --my-list foo --my-list bar
['foo', 'bar']
$ inv mytask --my-list foo --my-list bar --my-list foo
['foo', 'bar', 'foo']
```

Incrementable flag values

This is arguably a sub-case of *iterable flag values* (seen above) - it has the same core interface of "give a CLI argument multiple times, and have that do something other than error or overwrite a single value." However, 'incrementables' (as you may have guessed) increment an integer instead of building a list of strings. This is commonly found in verbosity flags and similar functionality.

An example of exactly that:

```
@task(incrementable=['verbose'])
def mytask(c, verbose=0):
    print(verbose)
```

And its use:

```
$ inv mytask
0
```

```
$ inv mytask --verbose
1
$ inv mytask -v
1
$inv mytask -vvv
3
```

Happily, because in Python `0` is 'falsey' and `1` (or any other number) is 'truthy', this functions a lot like a boolean flag as well, at least if one defaults it to `0`.

Note: You may supply any integer default value for such arguments (it simply serves as the starting value), but take care that consumers of the argument are written understanding that it is always going to appear 'truthy' unless it's `0`!

Dashes vs underscores in flag names

In Python, it's common to use `underscored_names` for keyword arguments, e.g.:

```
@task
def mytask(c, my_option=False):
    pass
```

However, the typical convention for command-line flags is dashes, which aren't valid in Python identifiers:

```
$ inv mytask --my-option
```

Invoke works around this by automatically generating dashed versions of underscored names, when it turns your task function signatures into command-line parser flags.

Therefore, the two examples above actually work fine together – `my_option` ends up mapping to `--my-option`.

In addition, leading (`_myopt`) and trailing (`myopt_`) underscores are ignored, since `invoke ---myopt` and `invoke --myopt-` don't make much sense.

Automatic Boolean inverse flags

Boolean flags tend to work best when setting something that is normally `False`, to `True`:

```
$ inv mytask --yes-please-do-x
```

However, in some cases, you want the opposite - a default of `True`, which can be easily disabled. For example, colored output:

```
@task
def run_tests(c, color=True):
    # ...
```

Here, what we really want on the command line is a `--no-color` flag that sets `color=False`. Invoke handles this for you: when setting up CLI flags, booleans which default to `True` generate a `--no-<name>` flag instead.

3.2.3 How tasks run

Base case

In the simplest case, a task with no pre- or post-tasks runs one time. Example:

```
@task
def hello(c):
    print("Hello, world!")
```

Execution:

```
$ inv hello
Hello, world!
```

Pre- and post-tasks

Tasks that should always have another task executed before or after them, may use the `@task` deocator's `pre` and/or `post` kwargs, like so:

```
@task
def clean(c):
    print("Cleaning")

@task
def publish(c):
    print("Publishing")

@task(pre=[clean], post=[publish])
def build(c):
    print("Building")
```

Execution:

```
$ inv build
Cleaning
Building
Publishing
```

These keyword arguments always take iterables. As a convenience, pre-tasks (and pre-tasks only) may be given as positional arguments, in a manner similar to build systems like `make`. E.g. we could present part of the above example as:

```
@task
def clean(c):
    print("Cleaning")

@task(clean)
def build(c):
    print("Building")
```

As before, `invoke build` would cause `clean` to run, then `build`.

Recursive/chained pre/post-tasks

Pre-tasks of pre-tasks will also be invoked (as will post-tasks of pre-tasks, pre-tasks of post-tasks, etc) in a depth-first manner, recursively. Here's a more complex (if slightly contrived) tasks file:

```
@task
def clean_html(c):
    print("Cleaning HTML")

@task
def clean_tgz(c):
    print("Cleaning .tar.gz files")

@task(clean_html, clean_tgz)
def clean(c):
    print("Cleaned everything")

@task
def makedirs(c):
    print("Making directories")

@task(clean, makedirs)
def build(c):
    print("Building")

@task(build)
def deploy(c):
    print("Deploying")
```

With a depth-first behavior, the below is hopefully intuitive to most users:

```
$ inv deploy
Cleaning HTML
Cleaning .tar.gz files
Cleaned everything
Making directories
Building
Deploying
```

Parameterizing pre/post-tasks

By default, pre- and post-tasks are executed with no arguments, even if the task triggering their execution was given some. When this is not suitable, you can wrap the task objects with `call` objects which allow you to specify a call signature:

```
@task
def clean(c, which=None):
    which = which or 'pyc'
    print("Cleaning {}".format(which))

@task(pre=[call(clean, which='all')]) # or call(clean, 'all')
def build(c):
    print("Building")
```

Example output:

```
$ inv build
Cleaning all
Building
```

Task deduplication

By default, any task that would run more than once during a session (due e.g. to inclusion in pre/post tasks), will only be run once. Example task file:

```python
@task
def clean(c):
    print("Cleaning")

@task(clean)
def build(c):
    print("Building")

@task(build)
def package(c):
    print("Packaging")
```

With deduplication turned off (see below), the above would execute `clean` -> `build` -> `build` again -> `package`. With deduplication, the double `build` does not occur:

```
$ inv build package
Cleaning
Building
Packaging
```

Note: Parameterized pre-tasks (using *`call`*) are deduped based on their argument lists. For example, if `clean` was parameterized and hooked up as a pre-task in two different ways - e.g. `call(clean, 'html')` and `call(clean, 'all')` - they would not get deduped should both end up running in the same session.

However, two separate references to `call(clean, 'html')` *would* become deduplicated.

Disabling deduplication

If you prefer your tasks to run every time no matter what, you can give the `--no-dedupe` core CLI option at runtime, or set the `tasks.dedupe` *config setting* to `False`. While it doesn't make a ton of real-world sense, let's imagine we wanted to apply `--no-dedupe` to the above example; we'd see the following output:

```
$ inv --no-dedupe build package
Cleaning
Building
Building
Packaging
```

The build step is now running twice.

3.3 Using Invoke as a library

While most of our documentation involves the user/CLI facing use cases of task management and command execution, Invoke was designed for its constituent parts to be usable independently by advanced users - either out of the box or with a minimum of extra work. CLI parsing, subprocess command execution, task organization, etc, are all written as broadly separated concerns.

This document outlines use cases already known to work (because downstream tools like Fabric are already utilizing them).

3.3.1 Reusing Invoke's CLI module as a distinct binary

A major use case is distribution of your own program using Invoke under the hood, bound to a different binary name, and usually setting a specific task *namespace* as the default. (This maps somewhat closely to things like `argparse` from the standard library.) In some cases, removing, replacing and/or adding core CLI flags is also desired.

Getting set up

Say you want to distribute a test runner called `tester` offering two subcommands, `unit` and `integration`, such that users could `pip install tester` and have access to commands like `tester unit`, `tester integration`, or `tester integration --fail-fast`.

First, as with any distinct Python package providing CLI 'binaries', you'd inform your `setup.py` of your entrypoint:

```
setup(
    name='tester',
    version='0.1.0',
    packages=['tester'],
    install_requires=['invoke'],
    entry_points={
        'console_scripts': ['tester = tester.main:program.run']
    }
)
```

Note: This is just an example snippet and is not a fully valid `setup.py`; if you don't know how Python packaging works, a good starting place is the Python Packaging User's Guide.

Nothing here is specific to Invoke - it's a standard way of telling Python to install a `tester` script that executes the `run` method of a `program` object defined inside the module `tester.main`.

Creating a `Program`

In our `tester/main.py`, we start out importing Invoke's public CLI functionality:

```
from invoke import Program
```

Then we define the `program` object we referenced in `setup.py`, which is a simple *`Program`* to do the heavy lifting, giving it our version number for starters:

```
program = Program(version='0.1.0')
```

At this point, installing `tester` would give you the same functionality as Invoke's *built-in CLI tool*, except named `tester` and exposing its own version number:

```
$ tester --version
Tester 0.1.0
$ tester --help
Usage: tester [--core-opts] task1 [--task1-opts] ... taskN [--taskN-opts]

Core options:
    ... core Invoke options here ...

$ tester --list
Can't find any collection named 'tasks'!
```

This doesn't do us much good yet - there aren't any subcommands (and our users don't care about arbitrary 'tasks', so Invoke's own default `--help` and `--list` output isn't a good fit).

Specifying subcommands

For `tester` to expose `unit` and `integration` subcommands, we need to define them, in a regular Invoke tasks module or *namespace*. For our example, we'll just create `tester/tasks.py` (but as you'll see in a moment, this too is arbitrary and can be whatever you like):

```
from invoke import task

@task
def unit(c):
    print("Running unit tests!")

@task
def integration(c):
    print("Running integration tests!")
```

As described in *Constructing namespaces*, you can arrange this module however you want - the above snippet uses an implicit namespace for brevity's sake.

Note: It's important to realize that there's nothing special about these "subcommands" - you could run them just as easily with vanilla Invoke, e.g. via `invoke --collection=tester.tasks --list`.

Now the useful part: telling our custom `Program` that this namespace of tasks should be used as the subcommands for `tester`, via the `namespace` kwarg:

```
from invoke import Collection, Program
from tester import tasks

program = Program(namespace=Collection.from_module(tasks), version='0.1.0')
```

The result?

```
$ tester --version
Tester 0.1.0
$ tester --help
Usage: tester [--core-opts] <subcommand> [--subcommand-opts] ...

Core options:
  ... core options here, minus task-related ones ...

Subcommands:
  unit
  integration

$ tester --list
No idea what '--list' is!
```

```
$ tester unit
Running unit tests!
```

Notice how the 'usage' line changed (to specify 'subcommands' instead of 'tasks'); the list of specific subcommands is now printed as part of `--help`; and `--list` has been removed from the options.

You can enable *tab-completion* for your distinct binary and subcommands.

Modifying core parser arguments

A common need for this use case is tweaking the core parser arguments. `Program` makes it easy: default core `Arguments` are returned by `Program.core_args`. Extend this method's return value with `super` and you're done:

```python
# Presumably, this is your setup.py-designated CLI module...

from invoke import Program, Argument

class MyProgram(Program):
    def core_args(self):
        core_args = super(MyProgram, self).core_args()
        extra_args = [
            Argument(names=('foo', 'f'), help="Foo the bars"),
            # ...
        ]
        return core_args + extra_args

program = MyProgram()
```

Warning: We don't recommend *omitting* any of the existing core arguments; a lot of basic functionality relies on their existence, even when left to default values.

3.3.2 Customizing the configuration system's defaults

Besides the CLI-oriented content of the previous section, another area of functionality that frequently needs updating when redistributing an Invoke codebase (CLI or no CLI) is configuration. There are typically two concerns here:

- Configuration filenames and the env var prefix - crucial if you ever expect your users to use the configuration system;
- Default configuration values - less critical (most defaults aren't labeled with anything Invoke-specific) but still sometimes desirable.

Note: Both of these involve subclassing `Config` (and, if using the CLI machinery, informing your `Program` to use that subclass instead of the default one.)

Changing filenames and/or env var prefix

By default, Invoke's config system looks for files like `/etc/invoke.yaml`, `~/.invoke.json`, etc. If you're distributing client code named something else, like the `Tester` example earlier, you might instead want the config system to load `/etc/tester.json` or `$CWD/tester.py`.

Similarly, the environment variable config level looks for env vars like `INVOKE_RUN_ECHO`; you might prefer `TESTER_RUN_ECHO`.

There are a few *`Config`* attributes controlling these values:

- `prefix`: A generic, catchall prefix used directly as the file prefix, and used via all-caps as the env var prefix;
- `file_prefix`: For overriding just the filename prefix - otherwise, it defaults to the value of `prefix`;
- `env_prefix`: For overriding just the env var prefix - as you might have guessed, it too defaults to the value of `prefix`.

Continuing our 'Tester' example, you'd do something like this:

```
from invoke import Config

class TesterConfig(Config):
    prefix = 'tester'
```

Or, to seek `tester.yaml` as before, but `TEST_RUN_ECHO` instead of `TESTER_RUN_ECHO`:

```
class TesterConfig(Config):
    prefix = 'tester'
    env_prefix = 'TEST'
```

Modifying default config values

Default config values are simple - they're just the return value of the staticmethod *`Config.global_defaults`*, so override that and return whatever you like - ideally something based on the superclass' values, as many defaults are assumed to exist by the rest of the system. (The helper function *`invoke.config.merge_dicts`* can be useful here.)

For example, say you want Tester to always echo shell commands by default when your codebase calls *`Context.run`*:

```
from invoke import Program
from invoke.config import Config, merge_dicts

class TesterConfig(Config):
    @staticmethod
    def global_defaults():
        their_defaults = Config.global_defaults()
        my_defaults = {
            'run': {
                'echo': True,
            },
        }
        return merge_dicts(their_defaults, my_defaults)

program = Program(config_class=TesterConfig, version='0.1.0')
```

For reference, Invoke's own base defaults (the. . . default defaults, you could say) are documented at *Default configuration values*.

3.4 Loading collections

The core of Invoke's execution model involves one or more Collection objects. While these may be created programmatically, in typical use Invoke will create them for you from Python modules it finds or is told to use.

3.4.1 Task module discovery

With no other configuration, simply calling `invoke` will look for a single Python module or package named `tasks`, and will treat it as the root namespace. `tasks` (or any other name given via *loading configuration options*) is searched for in the following ways:

- First, if a valid tasks module by that name already exists on Python's sys.path, no more searching is done – that module is selected.
- Failing that, search towards the root of the local filesystem, starting with the user's current working directory (os.getcwd) and try importing again with each directory temporarily added to `sys.path`.
  - Due to how Python's import machinery works, this approach will always favor a package directory (`tasks/` containing an `__init__.py`) over a module file (`tasks.py`) in the same location.
  - If a candidate is found and successfully imported, its parent directory will **stay** on `sys.path` during the rest of the Python session – this allows task code to make convenient assumptions concerning sibling modules' importability.

Candidate modules/packages are introspected to make sure they can actually be used as valid task collections. Any that fail are discarded, the `sys.path` munging done to import them is reverted, and the search continues.

3.4.2 Configuring the loading process

You can configure the above behavior, requesting that Invoke alter the collection name searched for and/or the path where filesystem-level loading starts looking.

For example, you may already have a project-level `tasks.py` that you can't easily rename; or you may want to host a number of tasks collections stored outside the project root and make it easy to switch between them; or any number of reasons.

Both the sought collection name and the search root can be specified via *configuration file options* or as *runtime CLI flags*:

- **Change the collection name**: Set the `tasks.collection_name` configuration option, or use *`--collection`*. It should be a Python module name and not a file name (so `mytasks`, not `mytasks.py` or `mytasks/`.)
- **Change the root search path**: Configure `tasks.search_root` or use *`--search-root`*. This value may be any valid directory path.

3.5 Constructing namespaces

The *base case* of loading a single module of tasks works fine initially, but advanced users typically need more organization, such as separating tasks into a tree of nested namespaces.

The *`Collection`* class provides an API for organizing tasks (and *their configuration*) into a tree-like structure. When referenced by strings (e.g. on the CLI or in pre/post hooks) tasks in nested namespaces use a dot-separated syntax, e.g. `docs.build`.

In this section, we show how building namespaces with this API is flexible but also allows following Python package layouts with minimal boilerplate.

3.5.1 Starting out

One unnamed `Collection` is always the namespace root; in the implicit base case, Invoke creates one for you from the tasks in your tasks module. Create your own, named `namespace` or `ns`, to set up an explicit namespace (i.e. to skip the default "pull in all Task objects" behavior):

```
from invoke import Collection

ns = Collection()
# or: namespace = Collection()
```

Add tasks with *`Collection.add_task`*. *`add_task`* can take an *`Task`* object, such as those generated by the *`task`* decorator:

```
from invoke import Collection, task

@task
def release(c):
    c.run("python setup.py sdist register upload")

ns = Collection()
ns.add_task(release)
```

Our available tasks list now looks like this:

```
$ invoke --list
Available tasks:

    release
```

3.5.2 Naming your tasks

By default, a task's function name is used as its namespace identifier, but you may override this by giving a `name` argument to either *`@task`* (i.e. at definition time) or *`Collection.add_task`* (i.e. at binding/attachment time).

For example, say you have a variable name collision in your tasks module – perhaps you want to expose a `dir` task, which shadows a Python builtin. Naming your function itself `dir` is a bad idea, but you can name the function something like `dir_` and then tell `@task` the "real" name:

```
@task(name='dir')
def dir_(c):
    # ...
```

On the other side, you might have obtained a task object that doesn't fit with the names you want in your namespace, and can rename it at attachment time. Maybe we want to rename our `release` task to be called `deploy` instead:

```
ns = Collection()
ns.add_task(release, name='deploy')
```

The result:

```
$ invoke --list
Available tasks:

    deploy
```

Note: The `name` kwarg is the 2nd argument to *`add_task`*, so those in a hurry can phrase it as:

```
ns.add_task(release, 'deploy')
```

Aliases

Tasks may have additional names or aliases, given as the `aliases` keyword argument; these are appended to, instead of replacing, any implicit or explicit `name` value:

```
ns.add_task(release, aliases=('deploy', 'pypi'))
```

Result, with three names for the same task:

```
$ invoke --list
Available tasks:

    release
    deploy
    pypi
```

Note: The convenience decorator *`@task`* is another method of setting aliases (e.g. `@task(aliases=('foo', 'bar'))`, and is useful for ensuring a given task always has some aliases set no matter how it's added to a namespace.

Dashes vs underscores

In the common case of functions-as-tasks, you'll often find yourself writing task names that contain underscores:

```
@task
def my_awesome_task(c):
    print("Awesome!")
```

Similar to how task arguments are processed to turn their underscores into dashes (since that's a common command-line convention) all underscores in task or collection names are interpreted to be dashes instead, by default:

```
$ inv --list
Available tasks:

  my-awesome-task

$ inv my-awesome-task
Awesome!
```

If you'd prefer the underscores to remain instead, you can update your configuration to set `tasks.auto_dash_names` to `False` in one of the non-runtime *config files* (system, user, or project.) For example, in `~/.invoke.yml`:

```
tasks:
    auto_dash_names: false
```

Note: In the interests of avoiding confusion, this setting is "exclusive" in nature - underscored version of task names *are not valid* on the CLI unless `auto_dash_names` is disabled. (However, at the pure function level within Python, they must continue to be referenced with underscores, as dashed names are not valid Python syntax!)

3.5.3 Nesting collections

The point of namespacing is to have sub-namespaces; to do this in Invoke, create additional *`Collection`* instances and add them to their parent collection via *`Collection.add_collection`*. For example, let's say we have a couple of documentation tasks:

```
@task
def build_docs(c):
    c.run("sphinx-build docs docs/_build")

@task
def clean_docs(c):
    c.run("rm -rf docs/_build")
```

We can bundle them up into a new, named collection like so:

```
docs = Collection('docs')
docs.add_task(build_docs, 'build')
docs.add_task(clean_docs, 'clean')
```

And then add this new collection under the root namespace with `add_collection`:

```
ns.add_collection(docs)
```

The result (assuming for now that `ns` currently just contains the original `release` task):

```
$ invoke --list
Available tasks:

    release
    docs.build
    docs.clean
```

As with tasks, collections may be explicitly bound to their parents with a different name than they were originally given (if any) via a `name` kwarg (also, as with `add_task`, the 2nd regular arg):

```
ns.add_collection(docs, 'sphinx')
```

Result:

```
$ invoke --list
Available tasks:

    release
    sphinx.build
    sphinx.clean
```

3.5.4 Importing modules as collections

A simple tactic which Invoke itself uses in the trivial, single-module case is to use *Collection.from_module* – a classmethod serving as an alternate Collection constructor which takes a Python module object as its first argument.

Modules given to this method are scanned for Task instances, which are added to a new Collection. By default, this collection's name is taken from the module name (the __name__ attribute), though it can also be supplied explicitly.

Note: As with the default task module, you can override this default loading behavior by declaring a ns or namespace *Collection* object at top level in the loaded module.

For example, let's reorganize our earlier single-file example into a Python package with several submodules. First, tasks/release.py:

```
from invoke import task

@task
def release(c):
    c.run("python setup.py sdist register upload")
```

And tasks/docs.py:

```
from invoke import task

@task
def build(c):
    c.run("sphinx-build docs docs/_build")

@task
def clean(c):
    c.run("rm -rf docs/_build")
```

Tying them together is tasks/__init__.py:

```
from invoke import Collection

import release, docs

ns = Collection()
ns.add_collection(Collection.from_module(release))
ns.add_collection(Collection.from_module(docs))
```

This form of the API is a little unwieldy in practice. Thankfully there's a shortcut: add_collection will notice when handed a module object as its first argument and call Collection.from_module for you internally:

```
ns = Collection()
ns.add_collection(release)
ns.add_collection(docs)
```

Either way, the result:

```
$ invoke --list
Available tasks:
    release.release
    docs.build
    docs.clean
```

3.5.5 Default tasks

Tasks may be declared as the default task to invoke for the collection they belong to, e.g. by giving `default=True` to *@task* (or to *Collection.add_task*.) This is useful when you have a bunch of related tasks in a namespace but one of them is the most commonly used, and maps well to the namespace as a whole.

For example, in the documentation submodule we've been experimenting with so far, the `build` task makes sense as a default, so we can say things like `invoke docs` as a shortcut to `invoke docs.build`. This is easy to do:

```
@task(default=True)
def build(c):
    # ...
```

When imported into the root namespace (as shown above) this alters the output of `--list`, highlighting the fact that `docs.build` can be invoked as `docs` if desired:

```
$ invoke --list
Available tasks:

    release.release
    docs.build (docs)
    docs.clean
```

Default subcollections

As of version 1.5, this functionality is also extended to subcollections: a subcollection can be specified as the default when being added to its parent collection, and that subcollection's own default task (or sub-subcollection!) will be invoked as the default for the parent.

An example probably makes that clearer. Here's a tiny inline task tree with two subcollections, each with their own default task:

```
from invoke import Collection, task

@task(default=True)
def build_all(c):
    print("build ALL THE THINGS!")

@task
def build_wheel(c):
    print("Just the wheel")

build = Collection(all=build_all, wheel=build_wheel)

@task(default=True)
def build_docs(c):
    print("Code without docs is no code at all")

docs = Collection(build_docs)
```

Then we tie those into one top level collection, setting the `build` subcollection as the overall default:

```
ns = Collection()
ns.add_collection(build, default=True)
ns.add_collection(docs)
```

The result is that `build.all` becomes the absolute default task:

```
$ invoke
build ALL THE THINGS!
```

3.5.6 Mix and match

You're not limited to the specific tactics shown above – now that you know the basic tools of `add_task` and `add_collection`, use whatever approach best fits your needs.

For example, let's say you wanted to keep things organized into submodules, but wanted to "promote" `release.release` back to the top level for convenience's sake. Just because it's stored in a module doesn't mean we must use `add_collection` – we could instead import the task itself and use `add_task` directly:

```
from invoke import Collection

import docs
from release import release

ns = Collection()
ns.add_collection(docs)
ns.add_task(release)
```

Result:

```
$ invoke --list
Available tasks:

    release
    docs.build
    docs.clean
```

3.5.7 More shortcuts

Finally, you can even skip `add_collection` and `add_task` if your needs are simple enough – `Collection`'s constructor will take unknown arguments and build the namespace from their values as appropriate:

```
from invoke import Collection

import docs, release

ns = Collection(release.release, docs)
```

Notice how we gave both a task object (`release.release`) and a module containing tasks (`docs`). The result is identical to the above:

```
$ invoke --list
Available tasks:

    release
    docs.build
    docs.clean
```

If given as keyword arguments, the keywords act like the `name` arguments do in the `add_*` methods. Naturally, both can be mixed together as well:

```
ns = Collection(docs, deploy=release.release)
```

Result:

```
$ invoke --list
Available tasks:

    deploy
    docs.build
    docs.clean
```

Note: You can still name these `Collection` objects with a leading string argument if desired, which can be handy when building sub-collections.

3.6 Testing Invoke-using codebases

Strategies for testing codebases that use Invoke; some applicable to code focused on CLI tasks, and others applicable to more generic/refactored setups.

3.6.1 Subclass & modify Invoke 'internals'

A quick foreword: most users will find the subsequent approaches suitable, but advanced users should note that Invoke has been designed so it is itself easily testable. This means that in many cases, even Invoke's "internals" are exposed as low/no-shared-responsibility, publicly documented classes which can be subclassed and modified to inject test-friendly values or mocks. Be sure to look over the *API documentation*!

3.6.2 Use `MockContext`

An instance of subclassing Invoke's public API for test purposes is our own *`MockContext`*. Codebases which revolve heavily around *`Context`* objects and their methods (most task-oriented code) will find it easy to test by injecting *`MockContext`* objects which have been instantiated to yield partial *`Result`* objects.

For example, take this task:

```
from invoke import task

@task
def get_platform(c):
    uname = c.run("uname -s").stdout.strip()
    if uname == 'Darwin':
        return "You paid the Apple tax!"
    elif uname == 'Linux':
        return "Year of Linux on the desktop!"
```

An example of testing it with *MockContext* could be the following:

```
from invoke import MockContext, Result
from mytasks import get_platform

def test_get_platform_on_mac():
    c = MockContext(run=Result("Darwin\n"))
    assert "Apple" in get_platform(c)

def test_get_platform_on_linux():
    c = MockContext(run=Result("Linux\n"))
    assert "desktop" in get_platform(c)
```

Putting the `Mock` in `MockContext`

Starting in Invoke 1.5, *MockContext* will attempt to import the `mock` library at instantiation time and wrap its methods within `Mock` objects. This lets you not only present realistic return values to your code, but make test assertions about what commands your code is running.

Here's another "platform sensitive" task, being tested with the assumption that the test environment has some flavor of `mock` installed (here we'll pretend it's Python 3.6 or later, and also use some f-strings for brevity):

```
from invoke import task

@task
def replace(c, path, search, replacement):
    # Assume systems have sed, and that some (eg macOS w/ Homebrew) may
    # have gsed, implying regular sed is BSD style.
    has_gsed = c.run("which gsed", warn=True, hide=True)
    # Set command to run accordingly
    binary = "gsed" if has_gsed else "sed"
    c.run(f"{binary} -e 's/{search}/{replacement}/g' {path}")
```

The test code (again, which presumes that eg `MockContext.run` is now a `Mock` wrapper) relies primarily on 'last call' assertions (`Mock.assert_called_with`) but you can of course use any `Mock` methods you need. It also shows how you can set the mock context to respond to multiple possible commands, using a dict value:

```
from invoke import MockContext, Result
from mytasks import replace

def test_regular_sed():
    expected_sed = "sed -e s/foo/bar/g file.txt"
    c = MockContext(run={
        "which gsed": Result(exited=1),
        expected_sed: Result(),
    })
    replace(c, 'file.txt', 'foo', 'bar')
    c.run.assert_called_with(expected_sed)

def test_homebrew_gsed():
    expected_sed = "gsed -e s/foo/bar/g file.txt"
    c = MockContext(run={
        "which gsed": Result(exited=0),
        expected_sed: Result(),
    })
    replace(c, 'file.txt', 'foo', 'bar')
    c.run.assert_called_with(expected_sed)
```

Boolean mock results

You may have noticed the above example uses a handful of 'empty' `Result` objects; these stand in for "succeeded, but otherwise had no useful attributes" command executions (as `Result` defaults to an exit code of `0` and empty strings for stdout/stderr).

This is relatively common - think "interrogative" commands where the caller only cares for a boolean result, or times when a command is called purely for its side effects. To support this, there's a shorthand in `MockContext`: passing `True` or `False` to stand in for otherwise blank Results with exit codes of `0` or `1` respectively.

The example tests then look like this:

```
from invoke import MockContext, Result
from mytasks import replace

def test_regular_sed():
    expected_sed = "sed -e s/foo/bar/g file.txt"
    c = MockContext(run={
        "which gsed": False,
        expected_sed: True,
    })
    replace(c, 'file.txt', 'foo', 'bar')
    c.run.assert_called_with(expected_sed)

def test_homebrew_gsed():
    expected_sed = "gsed -e s/foo/bar/g file.txt"
    c = MockContext(run={
        "which gsed": True,
        expected_sed: True,
    })
    replace(c, 'file.txt', 'foo', 'bar')
    c.run.assert_called_with(expected_sed)
```

String mock results

Another convenient shorthand is using string values, which are interpreted to be the stdout of the resulting `Result`. This only really saves you from writing out the class itself (since `stdout` is the first positional arg of `Result`!) but "command X results in stdout Y" is a common enough use case that we implemented it anyway.

By example, let's modify an earlier example where we cared about stdout:

```
from invoke import MockContext
from mytasks import get_platform

def test_get_platform_on_mac():
    c = MockContext(run="Darwin\n")
    assert "Apple" in get_platform(c)

def test_get_platform_on_linux():
    c = MockContext(run="Linux\n")
    assert "desktop" in get_platform(c)
```

As with everything else in this document, this tactic can be applied to iterators or mappings as well as individual values.

Regular expression command matching

The dict form of *MockContext* kwarg can accept regular expression objects as keys, in addition to strings; ideal for situations where you either don't know the exact command being invoked, or simply don't need or want to write out the entire thing.

Imagine you're writing a function to run package management commands on a few different Linux distros and you're trying to test its error handling. You might want to set up a context that pretends any arbitrary `apt` or `yum` command fails, and ensure the function returns stderr when it encounters a problem:

```python
import re
from invoke import MockContext
from mypackage.tasks import install

package_manager = re.compile(r"^(apt(-get)?|yum) .*")

def test_package_success_returns_True():
    c = MockContext(run={package_manager: True})
    assert install(c, package="somepackage") is True

def test_package_explosions_return_stderr():
    c = MockContext(run={
        package_manager: Result(stderr="oh no!", exited=1),
    })
    assert install(c, package="otherpackage") == "oh no!"
```

A bit contrived - there are a bunch of other ways to organize this exact test code so you don't truly need the regex - but hopefully it's clear that when you *do* need this flexibility, this is how you could go about it.

Repeated results

By default, the values in these mock structures are consumed, causing *MockContext* to raise `NotImplementedError` afterwards (as it does for any unexpected command executions). This was designed with the assumption that most code under test will run a given command once.

If your situation doesn't match this, give `repeat=True` to the constructor, and you'll see values repeat indefinitely instead (or in cycles, for iterables).

3.6.3 Expect `Results`

The core Invoke subprocess methods like *run* all return *Result* objects - which (as seen above) can be readily instantiated by themselves with only partial data (e.g. standard output, but no exit code or standard error).

This means that well-organized code can be even easier to test and doesn't require as much use of *MockContext*.

An iteration on the initial *MockContext*-using example above:

```python
from invoke import task

@task
def get_platform(c):
    print(platform_response(c.run("uname -s")))

def platform_response(result):
    uname = result.stdout.strip()
    if uname == 'Darwin':
        return "You paid the Apple tax!"
    elif uname == 'Linux':
        return "Year of Linux on the desktop!"
```

With the logic encapsulated in a subroutine, you can just unit test that function by itself, deferring testing of the task or its context:

```
from invoke import Result
from mytasks import platform_response

def test_platform_response_on_mac():
    assert "Apple" in platform_response(Result("Darwin\n"))

def test_platform_response_on_linux():
    assert "desktop" in platform_response(Result("Linux\n"))
```

3.6.4 Avoid mocking dependency code paths altogether

This is more of a general software engineering tactic, but the natural endpoint of the above code examples would be where your primary logic doesn't care about Invoke at all – only about basic Python (or locally defined) data types. This allows you to test logic in isolation and either ignore testing the Invoke side of things, or write targeted tests solely for where your code interfaces with Invoke.

Another minor tweak to the task code:

```
from invoke import task

@task
def show_platform(c):
    uname = c.run("uname -s").stdout.strip()
    print(platform_response(uname))

def platform_response(uname):
    if uname == 'Darwin':
        return "You paid the Apple tax!"
    elif uname == 'Linux':
        return "Year of Linux on the desktop!"
```

And the tests:

```
from mytasks import platform_response

def test_platform_response_on_mac():
    assert "Apple" in platform_response("Darwin\n")

def test_platform_response_on_linux():
    assert "desktop" in platform_response("Linux\n")
```

3.7 Automatically responding to program output

3.7.1 Background

Command-line programs tend to be designed for interactive shells, which frequently manifests as waiting around for user input, or "prompts". Well-designed programs offer options for pre-empting such prompts, resulting in an easily automated workflow – but with the rest, interactivity is unavoidable.

Thankfully, Invoke's `Runner` class not only forwards your standard input to the running program (allowing you to manually respond to prompts) but it can also be configured to respond automatically on your behalf.

3.7.2 Basic use

The mechanism for this automation is the `watchers` kwarg to the `Runner.run` method (and its wrappers elsewhere, such as `Context.run` and `invoke.run`), which is a list of `StreamWatcher`-subclass instances configured to watch for patterns & respond accordingly. The simplest of these is `Responder`, which just replies with its configured response every time its pattern is seen; others can be found in the *watchers module*.

Note: As with all other arguments to `run`, you can also set the default set of watchers globally via *configuration files*.

Take for example this program which expects a manual response to a yes/no prompt:

```
$ excitable-program
When you give the OK, I'm going to do the things. All of them!!
Are you ready? [Y/n] y
OK! I just did all sorts of neat stuff. You're welcome! Bye!
```

You *could* call `run("excitable-program")`, manually watch for the prompt, and mash Y by hand. But if you instead supply a `Responder` like so:

```
@task
def always_ready(c):
    responder = Responder(
        pattern=r"Are you ready? \[Y/n\] ",
        response="y\n",
    )
    c.run("excitable-program", watchers=[responder])
```

Then `Runner` passes the program's `stdout` and `stderr` through `responder`, which watches for `"Are you ready? [Y/n] "` and automatically writes `y` (plus `\n` to simulate hitting Enter/Return) to the program's `stdin`.

Note: The pattern argument to `Responder` is treated as a `regular expression`, requiring more care (note how we had to escape our square-brackets in the above example) but providing more power as well.

CHAPTER 4

API

Know what you're looking for & just need API details? View our auto-generated API documentation:

4.1 `__init__`

`invoke.`**`run`**`(`*command*, ***kwargs*`)`

Run `command` in a subprocess and return a *`Result`* object.

See *`Runner.run`* for API details.

Note: This function is a convenience wrapper around Invoke's *`Context`* and *`Runner`* APIs.

Specifically, it creates an anonymous *`Context`* instance and calls its *`run`* method, which in turn defaults to using a *`Local`* runner subclass for command execution.

New in version 1.0.

`invoke.`**`sudo`**`(`*command*, ***kwargs*`)`

Run `command` in a `sudo` subprocess and return a *`Result`* object.

See *`Context.sudo`* for API details, such as the `password` kwarg.

Note: This function is a convenience wrapper around Invoke's *`Context`* and *`Runner`* APIs.

Specifically, it creates an anonymous *`Context`* instance and calls its *`sudo`* method, which in turn defaults to using a *`Local`* runner subclass for command execution (plus sudo-related bits & pieces).

New in version 1.4.

4.2 `collection`

class `invoke.collection.`**`Collection`**`(`**args*, ***kwargs*`)`

A collection of executable tasks. See *Constructing namespaces*.

New in version 1.0.

`__getitem__``(`*name=None*`)`

Returns task named `name`. Honors aliases and subcollections.

If this collection has a default task, it is returned when `name` is empty or `None`. If empty input is given and no task has been selected as the default, ValueError will be raised.

Tasks within subcollections should be given in dotted form, e.g. 'foo.bar'. Subcollection default tasks will be returned on the subcollection's name.

New in version 1.0.

`add_collection``(`*coll*, *name=None*, *default=None*`)`

Add `Collection` `coll` as a sub-collection of this one.

Parameters

- **`coll`** – The `Collection` to add.
- **`name`** (`str`) – The name to attach the collection as. Defaults to the collection's own internal name.
- **`default`** – Whether this sub-collection('s default task-or-collection) should be the default invocation of the parent collection.

New in version 1.0.

Changed in version 1.5: Added the `default` parameter.

`add_task``(`*task*, *name=None*, *aliases=None*, *default=None*`)`

Add `Task` `task` to this collection.

Parameters

- **`task`** – The `Task` object to add to this collection.
- **`name`** – Optional string name to bind to (overrides the task's own self-defined `name` attribute and/or any Python identifier (i.e. `.func_name`.)
- **`aliases`** – Optional iterable of additional names to bind the task as, on top of the primary name. These will be used in addition to any aliases the task itself declares internally.
- **`default`** – Whether this task should be the collection default.

New in version 1.0.

`configuration``(`*taskpath=None*`)`

Obtain merged configuration values from collection & children.

Parameters **`taskpath`** – (Optional) Task name/path, identical to that used for `__getitem__` (e.g. may be dotted for nested tasks, etc.) Used to decide which path to follow in the collection tree when merging config values.

Returns A `dict` containing configuration values.

New in version 1.0.

configure(*options*)

(Recursively) merge `options` into the current *configuration*.

Options configured this way will be available to all tasks. It is recommended to use unique keys to avoid potential clashes with other config options

For example, if you were configuring a Sphinx docs build target directory, it's better to use a key like `'sphinx.target'` than simply `'target'`.

Parameters **options** – An object implementing the dictionary protocol.

Returns `None`.

New in version 1.0.

classmethod from_module(*module*, *name=None*, *config=None*, *loaded_from=None*, *auto_dash_names=None*)

Return a new `Collection` created from `module`.

Inspects `module` for any `Task` instances and adds them to a new `Collection`, returning it. If any explicit namespace collections exist (named `ns` or `namespace`) a copy of that collection object is preferentially loaded instead.

When the implicit/default collection is generated, it will be named after the module's `__name__` attribute, or its last dotted section if it's a submodule. (I.e. it should usually map to the actual `.py` filename.)

Explicitly given collections will only be given that module-derived name if they don't already have a valid `.name` attribute.

If the module has a docstring (`__doc__`) it is copied onto the resulting `Collection` (and used for display in help, list etc output.)

Parameters

- **name** (*str*) – A string, which if given will override any automatically derived collection name (or name set on the module's root namespace, if it has one.)
- **config** (*dict*) – Used to set config options on the newly created `Collection` before returning it (saving you a call to `configure`.)

  If the imported module had a root namespace object, `config` is merged on top of it (i.e. overriding any conflicts.)
- **loaded_from** (*str*) – Identical to the same-named kwarg from the regular class constructor - should be the path where the module was found.
- **auto_dash_names** (*bool*) – Identical to the same-named kwarg from the regular class constructor - determines whether emitted names are auto-dashed.

New in version 1.0.

serialized()

Return an appropriate-for-serialization version of this object.

See the documentation for `Program` and its `json` task listing format; this method is the driver for that functionality.

New in version 1.0.

subcollection_from_path(*path*)

Given a `path` to a subcollection, return that subcollection.

New in version 1.0.

`task_names`
Return all task identifiers for this collection as a one-level dict.

Specifically, a dict with the primary/"real" task names as the key, and any aliases as a list value.

It basically collapses the namespace tree into a single easily-scannable collection of invocation strings, and is thus suitable for things like flat-style task listings or transformation into parser contexts.

New in version 1.0.

`task_with_config`(*name*)
Return task named `name` plus its configuration dict.

E.g. in a deeply nested tree, this method returns the `Task`, and a configuration dict created by merging that of this `Collection` and any nested `Collections`, up through the one actually holding the `Task`.

See `__getitem__` for semantics of the `name` argument.

Returns Two-tuple of (`Task`, `dict`).

New in version 1.0.

`to_contexts`(*ignore_unknown_help=None*)
Returns all contained tasks and subtasks as a list of parser contexts.

Parameters **`ignore_unknown_help`** (`bool`) – Passed on to each task's `get_arguments()` method. See the config option by the same name for details.

New in version 1.0.

Changed in version 1.7: Added the `ignore_unknown_help` kwarg.

`transform`(*name*)
Transform `name` with the configured auto-dashes behavior.

If the collection's `auto_dash_names` attribute is `True` (default), all non leading/trailing underscores are turned into dashes. (Leading/trailing underscores tend to get stripped elsewhere in the stack.)

If it is `False`, the inverse is applied - all dashes are turned into underscores.

New in version 1.0.

4.3 config

class `invoke.config.`**`Config`**(*overrides=None*, *defaults=None*, *system_prefix=None*, *user_prefix=None*, *project_location=None*, *runtime_path=None*, *lazy=False*)
Invoke's primary configuration handling class.

See *Configuration* for details on the configuration system this class implements, including the *configuration hierarchy*. The rest of this class' documentation assumes familiarity with that document.

Access

Configuration values may be accessed and/or updated using dict syntax:

```
config['foo']
```

or attribute syntax:

```
config.foo
```

Nesting works the same way - dict config values are turned into objects which honor both the dictionary protocol and the attribute-access method:

```
config['foo']['bar']
config.foo.bar
```

A note about attribute access and methods

This class implements the entire dictionary protocol: methods such as `keys`, `values`, `items`, `pop` and so forth should all function as they do on regular dicts. It also implements new config-specific methods such as *`load_system`*, *`load_collection`*, *`merge`*, *`clone`*, etc.

Warning: Accordingly, this means that if you have configuration options sharing names with these methods, you **must** use dictionary syntax (e.g. `myconfig['keys']`) to access the configuration data.

Lifecycle

At initialization time, *`Config`*:

- creates per-level data structures;
- stores any levels supplied to *`__init__`*, such as defaults or overrides, as well as the various config file paths/filename patterns;
- and loads config files, if found (though typically this just means system and user-level files, as project and runtime files need more info before they can be found and loaded.)
  - This step can be skipped by specifying `lazy=True`.

At this point, *`Config`* is fully usable - and because it pre-emptively loads some config files, those config files can affect anything that comes after, like CLI parsing or loading of task collections.

In the CLI use case, further processing is done after instantiation, using the `load_*` methods such as *`load_overrides`*, *`load_project`*, etc:

- the result of argument/option parsing is applied to the overrides level;
- a project-level config file is loaded, as it's dependent on a loaded tasks collection;
- a runtime config file is loaded, if its flag was supplied;
- then, for each task being executed:
  - per-collection data is loaded (only possible now that we have collection & task in hand);
  - shell environment data is loaded (must be done at end of process due to using the rest of the config as a guide for interpreting env var names.)

At this point, the config object is handed to the task being executed, as part of its execution *`Context`*.

Any modifications made directly to the *`Config`* itself after this point end up stored in their own (topmost) config level, making it easier to debug final values.

Finally, any *deletions* made to the *`Config`* (e.g. applications of dict-style mutators like `pop`, `clear` etc) are also tracked in their own structure, allowing the config object to honor such method calls without mutating the underlying source data.

Special class attributes

The following class-level attributes are used for low-level configuration of the config system itself, such as which file paths to load. They are primarily intended for overriding by subclasses.

- `prefix`: Supplies the default value for `file_prefix` (directly) and `env_prefix` (uppercased). See their descriptions for details. Its default value is `"invoke"`.
- `file_prefix`: The config file 'basename' default (though it is not a literal basename; it can contain path parts if desired) which is appended to the configured values of `system_prefix`, `user_prefix`, etc, to arrive at the final (pre-extension) file paths.

  Thus, by default, a system-level config file path concatenates the `system_prefix` of `/etc/` with the `file_prefix` of `invoke` to arrive at paths like `/etc/invoke.json`.

  Defaults to `None`, meaning to use the value of `prefix`.
- `env_prefix`: A prefix used (along with a joining underscore) to determine which environment variables are loaded as the env var configuration level. Since its default is the value of `prefix` capitalized, this means env vars like `INVOKE_RUN_ECHO` are sought by default.

  Defaults to `None`, meaning to use the value of `prefix`.

New in version 1.0.

`__init__`(*overrides=None*, *defaults=None*, *system_prefix=None*, *user_prefix=None*, *project_location=None*, *runtime_path=None*, *lazy=False*)

Creates a new config object.

Parameters

- **`defaults`** (*`dict`*) – A dict containing default (lowest level) config data. Default: *`global_defaults`*.
- **`overrides`** (*`dict`*) – A dict containing override-level config data. Default: `{}`.
- **`system_prefix`** (*`str`*) – Base path for the global config file location; combined with the prefix and file suffixes to arrive at final file path candidates.

  Default: `/etc/` (thus e.g. `/etc/invoke.yaml` or `/etc/invoke.json`).
- **`user_prefix`** (*`str`*) – Like `system_prefix` but for the per-user config file. These variables are joined as strings, not via path-style joins, so they may contain partial file paths; for the per-user config file this often means a leading dot, to make the final result a hidden file on most systems.

  Default: `~/.` (e.g. `~/.invoke.yaml`).
- **`project_location`** (*`str`*) – Optional directory path of the currently loaded *`Collection`* (as loaded by *`Loader`*). When non-empty, will trigger seeking of per-project config files in this directory.
- **`runtime_path`** (*`str`*) – Optional file path to a runtime configuration file.

  Used to fill the penultimate slot in the config hierarchy. Should be a full file path to an existing file, not a directory path or a prefix.
- **`lazy`** (*`bool`*) – Whether to automatically load some of the lower config levels.

  By default (`lazy=False`), `__init__` automatically calls *`load_system`* and *`load_user`* to load system and user config files, respectively.

  For more control over what is loaded when, you can say `lazy=True`, and no automatic loading is done.

Note: If you give `defaults` and/or `overrides` as `__init__` kwargs instead of waiting to use *`load_defaults`* or *`load_overrides`* afterwards, those *will* still end up 'loaded' immediately.

clone(*into=None*)
Return a copy of this configuration object.

The new object will be identical in terms of configured sources and any loaded (or user-manipulated) data, but will be a distinct object with as little shared mutable state as possible.

Specifically, all `dict` values within the config are recursively recreated, with non-dict leaf values subjected to `copy.copy` (note: *not* `copy.deepcopy`, as this can cause issues with various objects such as compiled regexen or threading locks, often found buried deep within rich aggregates like API or DB clients).

The only remaining config values that may end up shared between a config and its clone are thus those 'rich' objects that do not `copy.copy` cleanly, or compound non-dict objects (such as lists or tuples).

Parameters **into** – A *Config* subclass that the new clone should be "upgraded" to.

Used by client libraries which have their own *Config* subclasses that e.g. define additional defaults; cloning "into" one of these subclasses ensures that any new keys/subtrees are added gracefully, without overwriting anything that may have been pre-defined.

Default: `None` (just clone into another regular *Config*).

Returns A *Config*, or an instance of the class given to `into`.

Raises `TypeError`, if `into` is given a value and that value is not a *Config* subclass.

New in version 1.0.

static global_defaults()
Return the core default settings for Invoke.

Generally only for use by *Config* internals. For descriptions of these values, see *Default configuration values*.

Subclasses may choose to override this method, calling `Config.global_defaults` and applying *merge_dicts* to the result, to add to or modify these values.

New in version 1.0.

load_collection(*data*, *merge=True*)
Update collection-driven config data.

load_collection is intended for use by the core task execution machinery, which is responsible for obtaining collection-driven data. See *Collection-based configuration* for details.

New in version 1.0.

load_defaults(*data*, *merge=True*)
Set or replace the 'defaults' configuration level, from `data`.

Parameters

- **data** (*dict*) – The config data to load as the defaults level.
- **merge** (*bool*) – Whether to merge the loaded data into the central config. Default: `True`.

Returns `None`.

New in version 1.0.

load_overrides(*data*, *merge=True*)
Set or replace the 'overrides' configuration level, from `data`.

Parameters

- **data** (*dict*) – The config data to load as the overrides level.
- **merge** (*bool*) – Whether to merge the loaded data into the central config. Default: `True`.

Returns `None`.

New in version 1.0.

load_project(*merge=True*)
Load a project-level config file, if possible.

Checks the configured `_project_prefix` value derived from the path given to *set_project_location*, which is typically set to the directory containing the loaded task collection.

Thus, if one were to run the CLI tool against a tasks collection `/home/myuser/code/tasks.py`, *load_project* would seek out files like `/home/myuser/code/invoke.yml`.

Parameters merge (*bool*) – Whether to merge the loaded data into the central config. Default: `True`.

Returns `None`.

New in version 1.0.

load_runtime(*merge=True*)
Load a runtime-level config file, if one was specified.

When the CLI framework creates a *Config*, it sets `_runtime_path`, which is a full path to the requested config file. This method attempts to load that file.

Parameters merge (*bool*) – Whether to merge the loaded data into the central config. Default: `True`.

Returns `None`.

New in version 1.0.

load_shell_env()
Load values from the shell environment.

load_shell_env is intended for execution late in a *Config* object's lifecycle, once all other sources (such as a runtime config file or per-collection configurations) have been loaded. Loading from the shell is not terrifically expensive, but must be done at a specific point in time to ensure the "only known config keys are loaded from the env" behavior works correctly.

See *Environment variables* for details on this design decision and other info re: how environment variables are scanned and loaded.

New in version 1.0.

load_system(*merge=True*)
Load a system-level config file, if possible.

Checks the configured `_system_prefix` path, which defaults to `/etc`, and will thus load files like `/etc/invoke.yml`.

Parameters merge (*bool*) – Whether to merge the loaded data into the central config. Default: `True`.

Returns `None`.

New in version 1.0.

load_user(*merge=True*)
Load a user-level config file, if possible.

Checks the configured `_user_prefix` path, which defaults to `~/.`, and will thus load files like `~/.invoke.yml`.

Parameters **merge** (`bool`) – Whether to merge the loaded data into the central config. Default: `True`.

Returns `None`.

New in version 1.0.

merge()
Merge all config sources, in order.

New in version 1.0.

set_project_location(*path*)
Set the directory path where a project-level config file may be found.

Does not do any file loading on its own; for that, see `load_project`.

New in version 1.0.

set_runtime_path(*path*)
Set the runtime config file path.

New in version 1.0.

class `invoke.config.`**DataProxy**
Helper class implementing nested dict+attr access for `Config`.

Specifically, is used both for `Config` itself, and to wrap any other dicts assigned as config values (recursively).

Warning: All methods (of this object or in subclasses) must take care to initialize new attributes via `self._set(name='value')`, or they'll run into recursion errors!

New in version 1.0.

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

classmethod from_data(*data*, *root=None*, *keypath=()*)
Alternate constructor for 'baby' DataProxies used as sub-dict values.

Allows creating standalone DataProxy objects while also letting subclasses like `Config` define their own `__init__` without muddling the two.

Parameters

- **data** (`dict`) – This particular DataProxy's personal data. Required, it's the Data being Proxied.
- **root** – Optional handle on a root DataProxy/Config which needs notification on data updates.
- **keypath** (`tuple`) – Optional tuple describing the path of keys leading to this DataProxy's location inside the `root` structure. Required if `root` was given (and vice versa.)

New in version 1.0.

`invoke.config.`**`copy_dict`**`(`*source*`)`

Return a fresh copy of `source` with as little shared state as possible.

Uses *`merge_dicts`* under the hood, with an empty `base` dict; see its documentation for details on behavior.

New in version 1.0.

`invoke.config.`**`excise`**`(`*dict_*, *keypath*`)`

Remove key pointed at by `keypath` from nested dict `dict_`, if exists.

New in version 1.0.

`invoke.config.`**`merge_dicts`**`(`*base*, *updates*`)`

Recursively merge dict `updates` into dict `base` (mutating `base`.)

- Values which are themselves dicts will be recursed into.
- Values which are a dict in one input and *not* a dict in the other input (e.g. if our inputs were `{'foo': 5}` and `{'foo': {'bar': 5}}`) are irreconciliable and will generate an exception.
- Non-dict leaf values are run through `copy.copy` to avoid state bleed.

Note: This is effectively a lightweight `copy.deepcopy` which offers protection from mismatched types (dict vs non-dict) and avoids some core deepcopy problems (such as how it explodes on certain object types).

Returns The value of `base`, which is mostly useful for wrapper functions like *`copy_dict`*.

New in version 1.0.

`invoke.config.`**`obliterate`**`(`*base*, *deletions*`)`

Remove all (nested) keys mentioned in `deletions`, from `base`.

New in version 1.0.

4.4 `context`

`class` `invoke.context.`**`Context`**`(`*config=None*`)`

Context-aware API wrapper & state-passing object.

`Context` objects are created during command-line parsing (or, if desired, by hand) and used to share parser and configuration state with executed tasks (see *Aside: what exactly is this 'context' arg anyway?*).

Specifically, the class offers wrappers for core API calls (such as *`run`*) which take into account CLI parser flags, configuration files, and/or changes made at runtime. It also acts as a proxy for its *`config`* attribute - see that attribute's documentation for details.

Instances of *`Context`* may be shared between tasks when executing sub-tasks - either the same context the caller was given, or an altered copy thereof (or, theoretically, a brand new one).

New in version 1.0.

`__init__``(`*config=None*`)`

Parameters **`config`** – *`Config`* object to use as the base configuration.

Defaults to an anonymous/default *`Config`* instance.

cd(*path*)

Context manager that keeps directory state when executing commands.

Any calls to *run*, *sudo*, 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.

Because use of *cd* affects all such invocations, any code making use of the *cwd* property 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 user's `$HOME`) and may be nested as well.

Below is a "normal" attempt at using the shell 'cd', which doesn't work since all commands are executed in individual subprocesses – state is **not** kept between invocations of *run* or *sudo*:

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

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

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

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

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

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

New in version 1.0.

Changed in version 1.5: Explicitly cast the `path` argument (the only argument) to a string; this allows any object defining `__str__` to be handed in (such as the various `Path` objects out there), and not just string literals.

command_cwds = None

A list of directories to 'cd' into before running commands with *run* or *sudo*; intended for management via *cd*, please see its docs for details.

command_prefixes = None

A list of commands to run (via "&&") before the main argument to any *run* or *sudo* calls. Note that the primary API for manipulating this list is *prefix*; see its docs for details.

config

The fully merged *Config* object appropriate for this context.

Config settings (see their documentation for details) may be accessed like dictionary keys (`c.config['foo']`) or object attributes (`c.config.foo`).

As a convenience shorthand, the *Context* object proxies to its `config` attribute in the same way - e.g. `c['foo']` or `c.foo` returns the same value as `c.config['foo']`.

cwd

Return the current working directory, accounting for uses of *cd*.

New in version 1.0.

prefix(*command*)

Prefix all nested *run*/*sudo* commands with given command plus `&&`.

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 c.prefix('workon myvenv'):
    c.run('./manage.py migrate')
```

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

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

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

```
with c.cd('/path/to/app'):
    with c.prefix('workon myvenv'):
        c.run('./manage.py migrate')
        c.run('./manage.py loaddata fixture')
```

Which would result in executions like so:

```
$ cd /path/to/app && workon myvenv && ./manage.py migrate
$ cd /path/to/app && workon myvenv && ./manage.py loaddata fixture
```

Finally, as alluded to above, *prefix* may be nested if desired, e.g.:

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

The result:

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

Contrived, but hopefully illustrative.

New in version 1.0.

run(*command*, ***kwargs*)

Execute a local shell command, honoring config options.

Specifically, this method instantiates a *Runner* subclass (according to the `runner` config option; default is *Local*) and calls its `.run` method with `command` and `kwargs`.

See *Runner.run* for details on `command` and the available keyword arguments.

New in version 1.0.

sudo(*command*, ***kwargs*)

Execute a shell command via `sudo` with password auto-response.

Basics

This method is identical to `run` but adds a handful of convenient behaviors around invoking the `sudo` program. It doesn't do anything users could not do themselves by wrapping `run`, but the use case is too common to make users reinvent these wheels themselves.

Note: If you intend to respond to sudo's password prompt by hand, just use `run("sudo command")` instead! The autoresponding features in this method will just get in your way.

Specifically, `sudo`:

- Places a `FailingResponder` into the `watchers` kwarg (see *Automatically responding to program output*) which:
  - searches for the configured `sudo` password prompt;
  - responds with the configured sudo password (`sudo.password` from the *configuration*);
  - can tell when that response causes an authentication failure (e.g. if the system requires a password and one was not configured), and raises `AuthFailure` if so.
- Builds a `sudo` command string using the supplied `command` argument, prefixed by various flags (see below);
- Executes that command via a call to `run`, returning the result.

Flags used

`sudo` flags used under the hood include:

- `-S` to allow auto-responding of password via stdin;
- `-p <prompt>` to explicitly state the prompt to use, so we can be sure our auto-responder knows what to look for;
- `-u <user>` if `user` is not `None`, to execute the command as a user other than `root`;
- When `-u` is present, `-H` is also added, to ensure the subprocess has the requested user's `$HOME` set properly.

Configuring behavior

There are a couple of ways to change how this method behaves:

- Because it wraps `run`, it honors all `run` config parameters and keyword arguments, in the same way that `run` does.
  - Thus, invocations such as `c.sudo('command', echo=True)` are possible, and if a config layer (such as a config file or env var) specifies that e.g. `run.warn = True`, that too will take effect under `sudo`.
- `sudo` has its own set of keyword arguments (see below) and they are also all controllable via the configuration system, under the `sudo.*` tree.
  - Thus you could, for example, pre-set a sudo user in a config file; such as an `invoke.json` containing `{"sudo": {"user": "someuser"}}`.

Parameters

- **password** (*str*) – Runtime override for `sudo.password`.
- **user** (*str*) – Runtime override for `sudo.user`.

New in version 1.0.

class `invoke.context.`**`MockContext`**(*config=None*, ***kwargs*)

A `Context` whose methods' return values can be predetermined.

Primarily useful for testing Invoke-using codebases.

Note: If this class' constructor is able to import the `Mock` class at runtime (via the `mock` or `unittest.mock` modules, in that order) it will wraps its `run`, etc methods in `Mock` objects. This allows you to easily assert that the methods (still returning the values you prepare them with) were actually called.

Note: Methods not given `Results` to yield will raise `NotImplementedError` if called (since the alternative is to call the real underlying method - typically undesirable when mocking.)

New in version 1.0.

Changed in version 1.5: Added conditional `Mock` wrapping of `run` and `sudo`.

`__init__`(*config=None*, ***kwargs*)

Create a `Context`-like object whose methods yield `Result` objects.

Parameters

- **`config`** – A Configuration object to use. Identical in behavior to `Context`.
- **`run`** – A data structure indicating what `Result` objects to return from calls to the instantiated object's `run` method (instead of actually executing the requested shell command).

  Specifically, this kwarg accepts:

  – A single `Result` object.

  – A boolean; if True, yields a `Result` whose `exited` is `0`, and if False, `1`.

  – An iterable of the above values, which will be returned on each subsequent call to `.run` (the first item on the first call, the second on the second call, etc).

  – A dict mapping command strings or compiled regexen to the above values (including an iterable), allowing specific call-and-response semantics instead of assuming a call order.
- **`sudo`** – Identical to `run`, but whose values are yielded from calls to `sudo`.
- **`repeat`** (`bool`) – A flag determining whether results yielded by this class' methods repeat or are consumed.

  For example, when a single result is indicated, it will normally only be returned once, causing `NotImplementedError` afterwards. But when `repeat=True` is given, that result is returned on every call, forever.

  Similarly, iterable results are normally exhausted once, but when this setting is enabled, they are wrapped in `itertools.cycle`.

  Default: `False` (for backwards compatibility reasons).

Raises `TypeError`, if the values given to `run` or other kwargs aren't of the expected types.

Changed in version 1.5: Added support for boolean and string result values.

Changed in version 1.5: Added support for regex dict keys.

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

set_result_for(*attname*, *command*, *result*)

Modify the stored mock results for given `attname` (e.g. `run`).

This is similar to how one instantiates `MockContext` with a `run` or `sudo` dict kwarg. For example, this:

```
mc = MockContext(run={'mycommand': Result("mystdout")})
assert mc.run('mycommand').stdout == "mystdout"
```

is functionally equivalent to this:

```
mc = MockContext()
mc.set_result_for('run', 'mycommand', Result("mystdout"))
assert mc.run('mycommand').stdout == "mystdout"
```

`set_result_for` is mostly useful for modifying an already-instantiated `MockContext`, such as one created by test setup or helper methods.

New in version 1.0.

4.5 exceptions

Custom exception classes.

These vary in use case from "we needed a specific data structure layout in exceptions used for message-passing" to simply "we needed to express an error condition in a way easily told apart from other, truly unexpected errors".

exception invoke.exceptions.**AmbiguousEnvVar**

Raised when loading env var config keys has an ambiguous target.

New in version 1.0.

__weakref__

list of weak references to the object (if defined)

exception invoke.exceptions.**AuthFailure**(*result*, *prompt*)

An authentication failure, e.g. due to an incorrect `sudo` password.

Note: `Result` objects attached to these exceptions typically lack exit code information, since the command was never fully executed - the exception was raised instead.

New in version 1.0.

exception invoke.exceptions.**CommandTimedOut**(*result*, *timeout*)

Raised when a subprocess did not exit within a desired timeframe.

exception invoke.exceptions.**Exit**(*message=None*, *code=None*)

Simple custom stand-in for SystemExit.

Replaces scattered sys.exit calls, improves testability, allows one to catch an exit request without intercepting real SystemExits (typically an unfriendly thing to do, as most users calling `sys.exit` rather expect it to truly exit.)

Defaults to a non-printing, exit-0 friendly termination behavior if the exception is uncaught.

If `code` (an int) given, that code is used to exit.

If `message` (a string) given, it is printed to standard error, and the program exits with code `1` by default (unless overridden by also giving `code` explicitly.)

New in version 1.0.

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

exception `invoke.exceptions.`**`Failure`**(*result*, *reason=None*)
Exception subclass representing failure of a command execution.

"Failure" may mean the command executed and the shell indicated an unusual result (usually, a non-zero exit code), or it may mean something else, like a `sudo` command which was aborted when the supplied password failed authentication.

Two attributes allow introspection to determine the nature of the problem:

- `result`: a *`Result`* instance with info about the command being executed and, if it ran to completion, how it exited.
- `reason`: a wrapped exception instance if applicable (e.g. a *`StreamWatcher`* raised *`WatcherError`*) or `None` otherwise, in which case, it's probably a *`Failure`* subclass indicating its own specific nature, such as *`UnexpectedExit`* or *`CommandTimedOut`*.

This class is only rarely raised by itself; most of the time *`Runner.run`* (or a wrapper of same, such as *`Context.sudo`*) will raise a specific subclass like *`UnexpectedExit`* or *`AuthFailure`*.

New in version 1.0.

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

`streams_for_display`()
Return stdout/err streams as necessary for error display.

Subject to the following rules:

- If a given stream was *not* hidden during execution, a placeholder is used instead, to avoid printing it twice.
- Only the last 10 lines of stream text is included.
- PTY-driven execution will lack stderr, and a specific message to this effect is returned instead of a stderr dump.

Returns Two-tuple of stdout, stderr strings.

New in version 1.3.

exception `invoke.exceptions.`**`ParseError`**(*msg*, *context=None*)
An error arising from the parsing of command-line flags/arguments.

Ambiguous input, invalid task names, invalid flags, etc.

New in version 1.0.

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

exception `invoke.exceptions.`**`PlatformError`**
Raised when an illegal operation occurs for the current platform.

E.g. Windows users trying to use functionality requiring the `pty` module.

Typically used to present a clearer error message to the user.

New in version 1.0.

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

exception `invoke.exceptions.`**`ResponseNotAccepted`**
A responder/watcher class noticed a 'bad' response to its submission.

Mostly used by *`FailingResponder`* and subclasses, e.g. "oh dear I autosubmitted a sudo password and it was incorrect."

New in version 1.0.

exception `invoke.exceptions.`**`SubprocessPipeError`**
Some problem was encountered handling subprocess pipes (stdout/err/in).

Typically only for corner cases; most of the time, errors in this area are raised by the interpreter or the operating system, and end up wrapped in a *`ThreadException`*.

New in version 1.3.

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

exception `invoke.exceptions.`**`ThreadException`**(*exceptions*)
One or more exceptions were raised within background threads.

The real underlying exceptions are stored in the *`exceptions`* attribute; see its documentation for data structure details.

Note: Threads which did not encounter an exception, do not contribute to this exception object and thus are not present inside *`exceptions`*.

New in version 1.0.

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

`exceptions = ()`
A tuple of *`ExceptionWrappers`* containing the initial thread constructor kwargs (because `threading.Thread` subclasses should always be called with kwargs) and the caught exception for that thread as seen by `sys.exc_info` (so: type, value, traceback).

Note: The ordering of this attribute is not well-defined.

Note: Thread kwargs which appear to be very long (e.g. IO buffers) will be truncated when printed, to avoid huge unreadable error display.

exception `invoke.exceptions.`**`UncastableEnvVar`**
Raised on attempted env var loads whose default values are too rich.

E.g. trying to stuff `MY_VAR="foo"` into `{'my_var': ['uh', 'oh']}` doesn't make any sense until/if we implement some sort of transform option.

New in version 1.0.

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

exception `invoke.exceptions.`**`UnexpectedExit`**(*result*, *reason=None*)

A shell command ran to completion but exited with an unexpected exit code.

Its string representation displays the following:

- Command executed;
- Exit code;
- The last 10 lines of stdout, if it was hidden;
- The last 10 lines of stderr, if it was hidden and non-empty (e.g. pty=False; when pty=True, stderr never happens.)

New in version 1.0.

exception `invoke.exceptions.`**`UnknownFileType`**

A config file of an unknown type was specified and cannot be loaded.

New in version 1.0.

`__weakref__`

list of weak references to the object (if defined)

exception `invoke.exceptions.`**`UnpicklableConfigMember`**

A config file contained module objects, which can't be pickled/copied.

We raise this more easily catchable exception instead of letting the (unclearly phrased) TypeError bubble out of the pickle module. (However, to avoid our own fragile catching of that error, we head it off by explicitly testing for module members.)

New in version 1.0.2.

`__weakref__`

list of weak references to the object (if defined)

exception `invoke.exceptions.`**`WatcherError`**

Generic parent exception class for *`StreamWatcher`*-related errors.

Typically, one of these exceptions indicates a *`StreamWatcher`* noticed something anomalous in an output stream, such as an authentication response failure.

`Runner` catches these and attaches them to *`Failure`* exceptions so they can be referenced by intermediate code and/or act as extra info for end users.

New in version 1.0.

`__weakref__`

list of weak references to the object (if defined)

4.6 `executor`

class `invoke.executor.`**`Executor`**(*collection*, *config=None*, *core=None*)

An execution strategy for Task objects.

Subclasses may override various extension points to change, add or remove behavior.

New in version 1.0.

`__init__`(*collection*, *config=None*, *core=None*)

Initialize executor with handles to necessary data structures.

Parameters

- **collection** – A *Collection* used to look up requested tasks (and their default config data, if any) by name during execution.
- **config** – An optional *Config* holding configuration state. Defaults to an empty *Config* if not given.
- **core** – An optional *ParseResult* holding parsed core program arguments. Defaults to `None`.

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

dedupe (*calls*)
Deduplicate a list of *tasks*.

Parameters calls – An iterable of *Call* objects representing tasks.

Returns A list of *Call* objects.

New in version 1.0.

execute (**tasks*)
Execute one or more `tasks` in sequence.

Parameters tasks – An all-purpose iterable of "tasks to execute", each member of which may take one of the following forms:

A string naming a task from the Executor's *Collection*. This name may contain dotted syntax appropriate for calling namespaced tasks, e.g. `subcollection.taskname`. Such tasks are executed without arguments.

A two-tuple whose first element is a task name string (as above) and whose second element is a dict suitable for use as `**kwargs` when calling the named task. E.g.:

```
[
    ('task1', {}),
    ('task2', {'arg1': 'val1'}),
    ...
]
```

is equivalent, roughly, to:

```
task1()
task2(arg1='val1')
```

A '.ParserContext' instance, whose `.name` attribute is used as the task name and whose `.as_kwargs` attribute is used as the task kwargs (again following the above specifications).

Note: When called without any arguments at all (i.e. when `*tasks` is empty), the default task from `self.collection` is used instead, if defined.

Returns

A dict mapping task objects to their return values.

This dict may include pre- and post-tasks if any were executed. For example, in a collection with a `build` task depending on another task named `setup`, executing `build` will result in a dict with two keys, one for `build` and one for `setup`.

New in version 1.0.

expand_calls(*calls*)
Expand a list of *Call* objects into a near-final list of same.

The default implementation of this method simply adds a task's pre/post-task list before/after the task itself, as necessary.

Subclasses may wish to do other things in addition (or instead of) the above, such as multiplying the *calls* by argument vectors or similar.

New in version 1.0.

normalize(*tasks*)
Transform arbitrary task list w/ various types, into *Call* objects.

See docstring for *execute* for details.

New in version 1.0.

4.7 loader

class invoke.loader.**Loader**(*config=None*)
Abstract class defining how to find/import a session's base *Collection*.

New in version 1.0.

__init__(*config=None*)
Set up a new loader with some *Config*.

Parameters config – An explicit *Config* to use; it is referenced for loading-related config options. Defaults to an anonymous `Config()` if none is given.

find(*name*)
Implementation-specific finder method seeking collection `name`.

Must return a 4-tuple valid for use by `imp.load_module`, which is typically a name string followed by the contents of the 3-tuple returned by `imp.find_module` (`file`, `pathname`, `description`.)

For a sample implementation, see *FilesystemLoader*.

New in version 1.0.

load(*name=None*)
Load and return collection module identified by `name`.

This method requires a working implementation of *find* in order to function.

In addition to importing the named module, it will add the module's parent directory to the front of `sys.path` to provide normal Python import behavior (i.e. so the loaded module may load local-to-it modules or packages.)

Returns Two-tuple of `(module, directory)` where `module` is the collection-containing Python module object, and `directory` is the string path to the directory the module was found in.

New in version 1.0.

class invoke.loader.**FilesystemLoader**(*start=None*, ***kwargs*)
Loads Python files from the filesystem (e.g. `tasks.py`.)

Searches recursively towards filesystem root from a given start point.

New in version 1.0.

4.8 `parser`

The command-line parsing framework is split up into a handful of sub-modules:

- `parser.argument`
- `parser.context` (not to be confused with the top level `context`!)
- `parser.parser`

API docs for all are below.

class `invoke.parser.parser.`**`ParseResult`**`(`**args*, ***kwargs*`)`

List-like object with some extra parse-related attributes.

Specifically, a `.remainder` attribute, which is the string found after a `--` in any parsed argv list; and an `.unparsed` attribute, a list of tokens that were unable to be parsed.

New in version 1.0.

`__weakref__`

list of weak references to the object (if defined)

class `invoke.parser.parser.`**`Parser`**`(`*contexts=()*, *initial=None*, *ignore_unknown=False*`)`

Create parser conscious of `contexts` and optional `initial` context.

`contexts` should be an iterable of `Context` instances which will be searched when new context names are encountered during a parse. These Contexts determine what flags may follow them, as well as whether given flags take values.

`initial` is optional and will be used to determine validity of "core" options/flags at the start of the parse run, if any are encountered.

`ignore_unknown` determines what to do when contexts are found which do not map to any members of `contexts`. By default it is `False`, meaning any unknown contexts result in a parse error exception. If `True`, encountering an unknown context halts parsing and populates the return value's `.unparsed` attribute with the remaining parse tokens.

New in version 1.0.

`__weakref__`

list of weak references to the object (if defined)

`parse_argv``(`*argv*`)`

Parse an argv-style token list `argv`.

Returns a list (actually a subclass, *`ParseResult`*) of *`ParserContext`* objects matching the order they were found in the `argv` and containing *`Argument`* objects with updated values based on any flags given.

Assumes any program name has already been stripped out. Good:

```
Parser(...).parse_argv(['--core-opt', 'task', '--task-opt'])
```

Bad:

```
Parser(...).parse_argv(['invoke', '--core-opt', ...])
```

Parameters **`argv`** – List of argument string tokens.

Returns A *`ParseResult`* (a `list` subclass containing some number of *`ParserContext`* objects).

New in version 1.0.

class `invoke.parser.context.`**`ParserContext`**(*name=None*, *aliases=()*, *args=()*)

Parsing context with knowledge of flags & their format.

Generally associated with the core program or a task.

When run through a parser, will also hold runtime values filled in by the parser.

New in version 1.0.

`__init__`(*name=None*, *aliases=()*, *args=()*)

Create a new `ParserContext` named `name`, with `aliases`.

`name` is optional, and should be a string if given. It's used to tell ParserContext objects apart, and for use in a Parser when determining what chunk of input might belong to a given ParserContext.

`aliases` is also optional and should be an iterable containing strings. Parsing will honor any aliases when trying to "find" a given context in its input.

May give one or more `args`, which is a quick alternative to calling `for arg in args: self.add_arg(arg)` after initialization.

`__weakref__`

list of weak references to the object (if defined)

`add_arg`(**args*, ***kwargs*)

Adds given `Argument` (or constructor args for one) to this context.

The Argument in question is added to the following dict attributes:

- `args`: "normal" access, i.e. the given names are directly exposed as keys.
- `flags`: "flaglike" access, i.e. the given names are translated into CLI flags, e.g. `"foo"` is accessible via `flags['--foo']`.
- `inverse_flags`: similar to `flags` but containing only the "inverse" versions of boolean flags which default to True. This allows the parser to track e.g. `--no-myflag` and turn it into a False value for the `myflag` Argument.

New in version 1.0.

`as_kwargs`

This context's arguments' values keyed by their `.name` attribute.

Results in a dict suitable for use in Python contexts, where e.g. an arg named `foo-bar` becomes accessible as `foo_bar`.

New in version 1.0.

`flag_names`()

Similar to *`help_tuples`* but returns flag names only, no helpstrs.

Specifically, all flag names, flattened, in rough order.

New in version 1.0.

`help_for`(*flag*)

Return 2-tuple of `(flag-spec, help-string)` for given `flag`.

New in version 1.0.

`help_tuples`()

Return sorted iterable of help tuples for all member Arguments.

Sorts like so:

- General sort is alphanumerically
- Short flags win over long flags
- Arguments with *only* long flags and *no* short flags will come first.
- When an Argument has multiple long or short flags, it will sort using the most favorable (lowest alphabetically) candidate.

This will result in a help list like so:

```
--alpha, --zeta # 'alpha' wins
--beta
-a, --query # short flag wins
-b, --argh
-c
```

New in version 1.0.

`invoke.parser.context.`**`flag_key`**(*x*)

Obtain useful key list-of-ints for sorting CLI flags.

New in version 1.0.

class `invoke.parser.argument.`**`Argument`**(*name=None*, *names=()*, *kind=<class 'str'>*, *default=None*, *help=None*, *positional=False*, *optional=False*, *incrementable=False*, *attr_name=None*)

A command-line argument/flag.

Parameters

- **`name`** – Syntactic sugar for `names=[<name>]`. Giving both `name` and `names` is invalid.
- **`names`** – List of valid identifiers for this argument. For example, a "help" argument may be defined with a name list of `['-h', '--help']`.
- **`kind`** – Type factory & parser hint. E.g. `int` will turn the default text value parsed, into a Python integer; and `bool` will tell the parser not to expect an actual value but to treat the argument as a toggle/flag.
- **`default`** – Default value made available to the parser if no value is given on the command line.
- **`help`** – Help text, intended for use with `--help`.
- **`positional`** – Whether or not this argument's value may be given positionally. When `False` (default) arguments must be explicitly named.
- **`optional`** – Whether or not this (non-`bool`) argument requires a value.
- **`incrementable`** – Whether or not this (`int`) argument is to be incremented instead of overwritten/assigned to.
- **`attr_name`** – A Python identifier/attribute friendly name, typically filled in with the underscored version when `name`/`names` contain dashes.

New in version 1.0.

`__weakref__`

list of weak references to the object (if defined)

`got_value`

Returns whether the argument was ever given a (non-default) value.

For most argument kinds, this simply checks whether the internally stored value is non-`None`; for others, such as `list` kinds, different checks may be used.

New in version 1.3.

name

The canonical attribute-friendly name for this argument.

Will be `attr_name` (if given to constructor) or the first name in `names` otherwise.

New in version 1.0.

set_value (*value*, *cast=True*)

Actual explicit value-setting API call.

Sets `self.raw_value` to `value` directly.

Sets `self.value` to `self.kind(value)`, unless:

- `cast=False`, in which case the raw value is also used.
- `self.kind==list`, in which case the value is appended to `self.value` instead of cast & over-written.
- `self.incrementable==True`, in which case the value is ignored and the current (assumed int) value is simply incremented.

New in version 1.0.

4.9 program

class `invoke.program.`**Program**(*version=None*, *namespace=None*, *name=None*, *binary=None*, *loader_class=None*, *executor_class=None*, *config_class=None*, *binary_names=None*)

Manages top-level CLI invocation, typically via `setup.py` entrypoints.

Designed for distributing Invoke task collections as standalone programs, but also used internally to implement the `invoke` program itself.

See also:

Reusing Invoke's CLI module as a distinct binary for a tutorial/walkthrough of this functionality.

New in version 1.0.

__init__(*version=None*, *namespace=None*, *name=None*, *binary=None*, *loader_class=None*, *executor_class=None*, *config_class=None*, *binary_names=None*)

Create a new, parameterized `Program` instance.

Parameters

- **version** (`str`) – The program's version, e.g. `"0.1.0"`. Defaults to `"unknown"`.
- **namespace** – A `Collection` to use as this program's subcommands.

  If `None` (the default), the program will behave like `invoke`, seeking a nearby task namespace with a `Loader` and exposing arguments such as `--list` and `--collection` for inspecting or selecting specific namespaces.

  If given a `Collection` object, will use it as if it had been handed to `--collection`. Will also update the parser to remove references to tasks and task-related options, and display the subcommands in `--help` output. The result will be a program that has a static set of subcommands.

- **name** (*str*) – The program's name, as displayed in `--version` output.

  If `None` (default), is a capitalized version of the first word in the `argv` handed to *run*. For example, when invoked from a binstub installed as `foobar`, it will default to `Foobar`.

- **binary** (*str*) – Descriptive lowercase binary name string used in help text.

  For example, Invoke's own internal value for this is `inv[oke]`, denoting that it is installed as both `inv` and `invoke`. As this is purely text intended for help display, it may be in any format you wish, though it should match whatever you've put into your `setup.py`'s `console_scripts` entry.

  If `None` (default), uses the first word in `argv` verbatim (as with `name` above, except not capitalized).

- **binary_names** – List of binary name strings, for use in completion scripts.

  This list ensures that the shell completion scripts generated by *--print-completion-script* instruct the shell to use that completion for all of this program's installed names.

  For example, Invoke's internal default for this is `["inv", "invoke"]`.

  If `None` (the default), the first word in `argv` (in the invocation of *--print-completion-script*) is used in a single-item list.

- **loader_class** – The *Loader* subclass to use when loading task collections.

  Defaults to *FilesystemLoader*.

- **executor_class** – The *Executor* subclass to use when executing tasks.

  Defaults to *Executor*; may also be overridden at runtime by the *configuration system* and its `tasks.executor_class` setting (anytime that setting is not `None`).

- **config_class** – The *Config* subclass to use for the base config object.

  Defaults to *Config*.

Changed in version 1.2: Added the `binary_names` argument.

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

`args`
Obtain core program args from `self.core` parse result.

New in version 1.0.

`binary`
Derive program's help-oriented binary name(s) from init args & argv.

New in version 1.0.

`binary_names`
Derive program's completion-oriented binary name(s) from args & argv.

New in version 1.2.

`called_as`
Returns the program name we were actually called as.

Specifically, this is the (Python's os module's concept of a) basename of the first argument in the parsed argument vector.

New in version 1.2.

core_args()
: Return default core `Argument` objects, as a list.

  New in version 1.0.

create_config()
: Instantiate a `Config` (or subclass, depending) for use in task exec.

  This Config is fully usable but will lack runtime-derived data like project & runtime config files, CLI arg overrides, etc. That data is added later in `update_config`. See `Config` docstring for lifecycle details.

  Returns `None`; sets `self.config` instead.

  New in version 1.0.

execute()
: Hand off data & tasks-to-execute specification to an `Executor`.

  Note: Client code just wanting a different `Executor` subclass can just set `executor_class` in `__init__`, or override `tasks.executor_class` anywhere in the *config system* (which may allow you to avoid using a custom Program entirely).

  New in version 1.0.

initial_context
: The initial parser context, aka core program flags.

  The specific arguments contained therein will differ depending on whether a bundled namespace was specified in `__init__`.

  New in version 1.0.

load_collection()
: Load a task collection based on parsed core args, or die trying.

  New in version 1.0.

name
: Derive program's human-readable name based on `binary`.

  New in version 1.0.

normalize_argv(*argv*)
: Massages `argv` into a useful list of strings.

  If None (the default), uses `sys.argv`.

  If a non-string iterable, uses that in place of `sys.argv`.

  If a string, performs a `str.split` and then executes with the result. (This is mostly a convenience; when in doubt, use a list.)

  Sets `self.argv` to the result.

  New in version 1.0.

parse_cleanup()
: Post-parsing, pre-execution steps such as –help, –list, etc.

  New in version 1.0.

parse_collection()
: Load a tasks collection & project-level config.

New in version 1.0.

parse_core_args()

Filter out core args, leaving any tasks or their args for later.

Sets `self.core` to the `ParseResult` from this step.

New in version 1.0.

parse_tasks()

Parse leftover args, which are typically tasks & per-task args.

Sets `self.parser` to the parser used, `self.tasks` to the parsed per-task contexts, and `self.core_via_tasks` to a context holding any core flags seen within the task contexts.

Also modifies `self.core` to include the data from `core_via_tasks` (so that it correctly reflects any supplied core flags regardless of where they appeared).

New in version 1.0.

print_columns(*tuples*)

Print tabbed columns from (name, help) `tuples`.

Useful for listing tasks + docstrings, flags + help strings, etc.

New in version 1.0.

print_task_help(*name*)

Print help for a specific task, e.g. `inv --help <taskname>`.

New in version 1.0.

run(*argv=None*, *exit=True*)

Execute main CLI logic, based on `argv`.

Parameters

- **argv** – The arguments to execute against. May be `None`, a list of strings, or a string. See `normalize_argv` for details.
- **exit** (`bool`) – When `False` (default: `True`), will ignore `ParseError`, `Exit` and `Failure` exceptions, which otherwise trigger calls to `sys.exit`.

Note: This is mostly a concession to testing. If you're setting this to `False` in a production setting, you should probably be using `Executor` and friends directly instead!

New in version 1.0.

task_args()

Return default task-related `Argument` objects, as a list.

These are only added to the core args in "task runner" mode (the default for `invoke` itself) - they are omitted when the constructor is given a non-empty `namespace` argument ("bundled namespace" mode).

New in version 1.0.

update_config(*merge=True*)

Update the previously instantiated `Config` with parsed data.

For example, this is how `--echo` is able to override the default config value for `run.echo`.

Parameters **merge** (`bool`) – Whether to merge at the end, or defer. Primarily useful for subclassers. Default: `True`.

New in version 1.0.

4.10 `runners`

class `invoke.runners.`**`Runner`**(*context*)

Partially-abstract core command-running API.

This class is not usable by itself and must be subclassed, implementing a number of methods such as *`start`*, *`wait`* and *`returncode`*. For a subclass implementation example, see the source code for *`Local`*.

New in version 1.0.

`__init__`(*context*)

Create a new runner with a handle on some *`Context`*.

Parameters `context` – a *`Context`* instance, used to transmit default options and provide access to other contextualized information (e.g. a remote-oriented *`Runner`* might want a *`Context`* subclass holding info about hostnames and ports.)

Note: The *`Context`* given to *`Runner`* instances **must** contain default config values for the *`Runner`* class in question. At a minimum, this means values for each of the default *`Runner.run`* keyword arguments such as `echo` and `warn`.

Raises `exceptions.ValueError` – if not all expected default values are found in `context`.

`context = None`

The *`Context`* given to the same-named argument of *`__init__`*.

`program_finished = None`

A `threading.Event` signaling program completion.

Typically set after *`wait`* returns. Some IO mechanisms rely on this to know when to exit an infinite read loop.

`read_chunk_size = 1000`

How many bytes (at maximum) to read per iteration of stream reads.

`input_sleep = 0.01`

How many seconds to sleep on each iteration of the stdin read loop and other otherwise-fast loops.

`warned_about_pty_fallback = None`

Whether pty fallback warning has been emitted.

`watchers = None`

A list of *`StreamWatcher`* instances for use by *`respond`*. Is filled in at runtime by *`run`*.

`run`(*command*, ***kwargs*)

Execute `command`, returning an instance of *`Result`* once complete.

By default, this method is synchronous (it only returns once the subprocess has completed), and allows interactive keyboard communication with the subprocess.

It can instead behave asynchronously (returning early & requiring interaction with the resulting object to manage subprocess lifecycle) if you specify `asynchronous=True`. Furthermore, you can completely disassociate the subprocess from Invoke's control (allowing it to persist on its own after Python exits) by saying `disown=True`. See the per-kwarg docs below for details on both of these.

Note: All kwargs will default to the values found in this instance's *context* attribute, specifically in its configuration's `run` subtree (e.g. `run.echo` provides the default value for the `echo` keyword, etc). The base default values are described in the parameter list below.

Parameters

- **command** (*str*) – The shell command to execute.
- **asynchronous** (*bool*) – When set to `True` (default `False`), enables asynchronous behavior, as follows:
  - Connections to the controlling terminal are disabled, meaning you will not see the subprocess output and it will not respond to your keyboard input - similar to `hide=True` and `in_stream=False` (though explicitly given `(out|err|in)_stream` file-like objects will still be honored as normal).
  - *run* returns immediately after starting the subprocess, and its return value becomes an instance of *Promise* instead of *Result*.
  - *Promise* objects are primarily useful for their *join* method, which blocks until the subprocess exits (similar to threading APIs) and either returns a final *Result* or raises an exception, just as a synchronous `run` would.
    - As with threading and similar APIs, users of `asynchronous=True` should make sure to `join` their *Promise* objects to prevent issues with interpreter shutdown.
    - One easy way to handle such cleanup is to use the *Promise* as a context manager - it will automatically `join` at the exit of the context block.

  New in version 1.4.
- **disown** (*bool*) – When set to `True` (default `False`), returns immediately like `asynchronous=True`, but does not perform any background work related to that subprocess (it is completely ignored). This allows subprocesses using shell backgrounding or similar techniques (e.g. trailing `&`, `nohup`) to persist beyond the lifetime of the Python process running Invoke.

  Note: If you're unsure whether you want this or `asynchronous`, you probably want `asynchronous`!

  Specifically, `disown=True` has the following behaviors:
  - The return value is `None` instead of a *Result* or subclass.
  - No I/O worker threads are spun up, so you will have no access to the subprocess' stdout/stderr, your stdin will not be forwarded, `(out|err|in)_stream` will be ignored, and features like `watchers` will not function.
  - No exit code is checked for, so you will not receive any errors if the subprocess fails to exit cleanly.
  - `pty=True` may not function correctly (subprocesses may not run at all; this seems to be a potential bug in Python's `pty.fork`) unless your command line includes tools such as `nohup` or (the shell builtin) `disown`.

  New in version 1.4.

- **`dry`** (*`bool`*) – Whether to dry-run instead of truly invoking the given command. See *`--dry`* (which flips this on globally) for details on this behavior.

  New in version 1.3.

- **`echo`** (*`bool`*) – Controls whether *`run`* prints the command string to local stdout prior to executing it. Default: `False`.

  Note: `hide=True` will override `echo=True` if both are given.

- **`echo_format`** – A string, which when passed to Python's inbuilt `.format` method, will change the format of the output when `run.echo` is set to true.

  Currently, only `{command}` is supported as a parameter.

  Defaults to printing the full command string in ANSI-escaped bold.

- **`echo_stdin`** (*`bool`*) – Whether to write data from `in_stream` back to `out_stream`.

  In other words, in normal interactive usage, this parameter controls whether Invoke mirrors what you type back to your terminal.

  By default (when `None`), this behavior is triggered by the following:

  – Not using a pty to run the subcommand (i.e. `pty=False`), as ptys natively echo stdin to stdout on their own;

  – And when the controlling terminal of Invoke itself (as per `in_stream`) appears to be a valid terminal device or TTY. (Specifically, when *`isatty`* yields a `True` result when given `in_stream`.)

    Note: This property tends to be `False` when piping another program's output into an Invoke session, or when running Invoke within another program (e.g. running Invoke from itself).

  If both of those properties are true, echoing will occur; if either is false, no echoing will be performed.

  When not `None`, this parameter will override that auto-detection and force, or disable, echoing.

- **`encoding`** (*`str`*) – Override auto-detection of which encoding the subprocess is using for its stdout/stderr streams (which defaults to the return value of *`default_encoding`*).

- **`err_stream`** – Same as `out_stream`, except for standard error, and defaulting to `sys.stderr`.

- **`env`** (*`dict`*) – By default, subprocesses receive a copy of Invoke's own environment (i.e. `os.environ`). Supply a dict here to update that child environment.

  For example, `run('command', env={'PYTHONPATH': '/some/virtual/env/maybe'})` would modify the `PYTHONPATH` env var, with the rest of the child's env looking identical to the parent.

  See also:

  `replace_env` for changing 'update' to 'replace'.

- **`fallback`** (*`bool`*) – Controls auto-fallback behavior re: problems offering a pty when `pty=True`. Whether this has any effect depends on the specific *`Runner`* subclass being invoked. Default: `True`.
- **`hide`** – Allows the caller to disable `run`'s default behavior of copying the subprocess' stdout and stderr to the controlling terminal. Specify `hide='out'` (or `'stdout'`) to hide only the stdout stream, `hide='err'` (or `'stderr'`) to hide only stderr, or `hide='both'` (or `True`) to hide both streams.

  The default value is `None`, meaning to print everything; `False` will also disable hiding.

  Note: Stdout and stderr are always captured and stored in the `Result` object, regardless of `hide`'s value.

  Note: `hide=True` will also override `echo=True` if both are given (either as kwargs or via config/CLI).

- **`in_stream`** – A file-like stream object to used as the subprocess' standard input. If `None` (the default), `sys.stdin` will be used.

  If `False`, will disable stdin mirroring entirely (though other functionality which writes to the subprocess' stdin, such as autoresponding, will still function.) Disabling stdin mirroring can help when `sys.stdin` is a misbehaving non-stream object, such as under test harnesses or headless command runners.
- **`out_stream`** – A file-like stream object to which the subprocess' standard output should be written. If `None` (the default), `sys.stdout` will be used.
- **`pty`** (*`bool`*) – By default, `run` connects directly to the invoked process and reads its stdout/stderr streams. Some programs will buffer (or even behave) differently in this situation compared to using an actual terminal or pseudoterminal (pty). To use a pty instead of the default behavior, specify `pty=True`.

  Warning: Due to their nature, ptys have a single output stream, so the ability to tell stdout apart from stderr is **not possible** when `pty=True`. As such, all output will appear on `out_stream` (see below) and be captured into the `stdout` result attribute. `err_stream` and `stderr` will always be empty when `pty=True`.

- **`replace_env`** (*`bool`*) – When `True`, causes the subprocess to receive the dictionary given to `env` as its entire shell environment, instead of updating a copy of `os.environ` (which is the default behavior). Default: `False`.
- **`shell`** (*`str`*) – Which shell binary to use. Default: `/bin/bash` (on Unix; `COMSPEC` or `cmd.exe` on Windows.)
- **`timeout`** – Cause the runner to submit an interrupt to the subprocess and raise *`CommandTimedOut`*, if the command takes longer than `timeout` seconds to execute. Defaults to `None`, meaning no timeout.

  New in version 1.3.
- **`warn`** (*`bool`*) – Whether to warn and continue, instead of raising *`UnexpectedExit`*, when the executed command exits with a nonzero status. Default: `False`.

Note: This setting has no effect on exceptions, which will still be raised, typically bundled in *ThreadException* objects if they were raised by the IO worker threads.

Similarly, *WatcherError* exceptions raised by *StreamWatcher* instances will also ignore this setting, and will usually be bundled inside *Failure* objects (in order to preserve the execution context).

Ditto *CommandTimedOut* - basically, anything that prevents a command from actually getting to "exited with an exit code" ignores this flag.

- **watchers** – A list of *StreamWatcher* instances which will be used to scan the program's `stdout` or `stderr` and may write into its `stdin` (typically `str` or `bytes` objects depending on Python version) in response to patterns or other heuristics.

  See *Automatically responding to program output* for details on this functionality.

  Default: `[]`.

Returns *Result*, or a subclass thereof.

Raises *UnexpectedExit*, if the command exited nonzero and `warn` was `False`.

Raises *Failure*, if the command didn't even exit cleanly, e.g. if a *StreamWatcher* raised *WatcherError*.

Raises *ThreadException* (if the background I/O threads encountered exceptions other than *WatcherError*).

New in version 1.0.

make_promise()

Return a *Promise* allowing async control of the rest of lifecycle.

New in version 1.4.

create_io_threads()

Create and return a dictionary of IO thread worker objects.

Caller is expected to handle persisting and/or starting the wrapped threads.

generate_result(***kwargs*)

Create & return a suitable *Result* instance from the given `kwargs`.

Subclasses may wish to override this in order to manipulate things or generate a *Result* subclass (e.g. ones containing additional metadata besides the default).

New in version 1.0.

read_proc_output(*reader*)

Iteratively read & decode bytes from a subprocess' out/err stream.

Parameters **reader** – A literal reader function/partial, wrapping the actual stream object in question, which takes a number of bytes to read, and returns that many bytes (or `None`).

`reader` should be a reference to either *read_proc_stdout* or *read_proc_stderr*, which perform the actual, platform/library specific read calls.

Returns

A generator yielding Unicode strings (`unicode` on Python 2; `str` on Python 3).

Specifically, each resulting string is the result of decoding `read_chunk_size` bytes read from the subprocess' out/err stream.

New in version 1.0.

`write_our_output`(*stream*, *string*)

Write `string` to `stream`.

Also calls `.flush()` on `stream` to ensure that real terminal streams don't buffer.

Parameters

- **`stream`** – A file-like stream object, mapping to the `out_stream` or `err_stream` parameters of `run`.
- **`string`** – A Unicode string object.

Returns `None`.

New in version 1.0.

`handle_stdout`(*buffer_*, *hide*, *output*)

Read process' stdout, storing into a buffer & printing/parsing.

Intended for use as a thread target. Only terminates when all stdout from the subprocess has been read.

Parameters

- **`buffer`** – The capture buffer shared with the main thread.
- **`hide`** (`bool`) – Whether or not to replay data into `output`.
- **`output`** – Output stream (file-like object) to write data into when not hiding.

Returns `None`.

New in version 1.0.

`handle_stderr`(*buffer_*, *hide*, *output*)

Read process' stderr, storing into a buffer & printing/parsing.

Identical to `handle_stdout` except for the stream read from; see its docstring for API details.

New in version 1.0.

`read_our_stdin`(*input_*)

Read & decode bytes from a local stdin stream.

Parameters **`input`** – Actual stream object to read from. Maps to `in_stream` in `run`, so will often be `sys.stdin`, but might be any stream-like object.

Returns A Unicode string, the result of decoding the read bytes (this might be the empty string if the pipe has closed/reached EOF); or `None` if stdin wasn't ready for reading yet.

New in version 1.0.

`handle_stdin`(*input_*, *output*, *echo*)

Read local stdin, copying into process' stdin as necessary.

Intended for use as a thread target.

Note: Because real terminal stdin streams have no well-defined "end", if such a stream is detected (based on existence of a callable `.fileno()`) this method will wait until `program_finished` is set, before terminating.

When the stream doesn't appear to be from a terminal, the same semantics as `handle_stdout` are used - the stream is simply `read()` from until it returns an empty value.

Parameters

- **input** – Stream (file-like object) from which to read.
- **output** – Stream (file-like object) to which echoing may occur.
- **echo** (`bool`) – User override option for stdin-stdout echoing.

Returns `None`.

New in version 1.0.

should_echo_stdin(*input_*, *output*)

Determine whether data read from `input_` should echo to `output`.

Used by `handle_stdin`; tests attributes of `input_` and `output`.

Parameters

- **input** – Input stream (file-like object).
- **output** – Output stream (file-like object).

Returns A `bool`.

New in version 1.0.

respond(*buffer_*)

Write to the program's stdin in response to patterns in `buffer_`.

The patterns and responses are driven by the `StreamWatcher` instances from the `watchers` kwarg of `run` - see *Automatically responding to program output* for a conceptual overview.

Parameters **buffer** – The capture buffer for this thread's particular IO stream.

Returns `None`.

New in version 1.0.

generate_env(*env*, *replace_env*)

Return a suitable environment dict based on user input & behavior.

Parameters

- **env** (`dict`) – Dict supplying overrides or full env, depending.
- **replace_env** (`bool`) – Whether `env` updates, or is used in place of, the value of `os.environ`.

Returns A dictionary of shell environment vars.

New in version 1.0.

should_use_pty(*pty*, *fallback*)

Should execution attempt to use a pseudo-terminal?

Parameters

- **pty** (`bool`) – Whether the user explicitly asked for a pty.
- **fallback** (`bool`) – Whether falling back to non-pty execution should be allowed, in situations where `pty=True` but a pty could not be allocated.

New in version 1.0.

has_dead_threads
Detect whether any IO threads appear to have terminated unexpectedly.

Used during process-completion waiting (in `wait`) to ensure we don't deadlock our child process if our IO processing threads have errored/died.

Returns `True` if any threads appear to have terminated with an exception, `False` otherwise.

New in version 1.0.

wait()
Block until the running command appears to have exited.

Returns `None`.

New in version 1.0.

write_proc_stdin(*data*)
Write encoded `data` to the running process' stdin.

Parameters **data** – A Unicode string.

Returns `None`.

New in version 1.0.

decode(*data*)
Decode some `data` bytes, returning Unicode.

New in version 1.0.

process_is_finished
Determine whether our subprocess has terminated.

Note: The implementation of this method should be nonblocking, as it is used within a query/poll loop.

Returns `True` if the subprocess has finished running, `False` otherwise.

New in version 1.0.

start(*command*, *shell*, *env*)
Initiate execution of `command` (via `shell`, with `env`).

Typically this means use of a forked subprocess or requesting start of execution on a remote system.

In most cases, this method will also set subclass-specific member variables used in other methods such as `wait` and/or `returncode`.

Parameters

- **command** (`str`) – Command string to execute.
- **shell** (`str`) – Shell to use when executing `command`.
- **env** (`dict`) – Environment dict used to prep shell environment.

New in version 1.0.

start_timer(*timeout*)
Start a timer to `kill` our subprocess after `timeout` seconds.

read_proc_stdout(*num_bytes*)
Read `num_bytes` from the running process' stdout stream.

Parameters **num_bytes** (*int*) – Number of bytes to read at maximum.

Returns A string/bytes object.

New in version 1.0.

read_proc_stderr(*num_bytes*)
Read `num_bytes` from the running process' stderr stream.

Parameters **num_bytes** (*int*) – Number of bytes to read at maximum.

Returns A string/bytes object.

New in version 1.0.

close_proc_stdin()
Close running process' stdin.

Returns `None`.

New in version 1.3.

default_encoding()
Return a string naming the expected encoding of subprocess streams.

This return value should be suitable for use by encode/decode methods.

New in version 1.0.

send_interrupt(*interrupt*)
Submit an interrupt signal to the running subprocess.

In almost all implementations, the default behavior is what will be desired: submit to the subprocess' stdin pipe. However, we leave this as a public method in case this default needs to be augmented or replaced.

Parameters **interrupt** – The locally-sourced `KeyboardInterrupt` causing the method call.

Returns `None`.

New in version 1.0.

returncode()
Return the numeric return/exit code resulting from command execution.

Returns `int`

New in version 1.0.

stop()
Perform final cleanup, if necessary.

This method is called within a `finally` clause inside the main *run* method. Depending on the subclass, it may be a no-op, or it may do things such as close network connections or open files.

Returns `None`

New in version 1.0.

stop_timer()
Cancel an open timeout timer, if required.

kill()

Forcibly terminate the subprocess.

Typically only used by the timeout functionality.

This is often a "best-effort" attempt, e.g. remote subprocesses often must settle for simply shutting down the local side of the network connection and hoping the remote end eventually gets the message.

timed_out

Returns `True` if the subprocess stopped because it timed out.

New in version 1.3.

__weakref__

list of weak references to the object (if defined)

class invoke.runners.**Local**(*context*)

Execute a command on the local system in a subprocess.

Note: When Invoke itself is executed without a controlling terminal (e.g. when `sys.stdin` lacks a useful `fileno`), it's not possible to present a handle on our PTY to local subprocesses. In such situations, `Local` will fallback to behaving as if `pty=False` (on the theory that degraded execution is better than none at all) as well as printing a warning to stderr.

To disable this behavior, say `fallback=False`.

New in version 1.0.

class invoke.runners.**Result**(*stdout=", stderr=", encoding=None, command=", shell=", env=None, exited=0, pty=False, hide=()*)

A container for information about the result of a command execution.

All params are exposed as attributes of the same name and type.

Parameters

- **stdout** (*str*) – The subprocess' standard output.
- **stderr** (*str*) – Same as `stdout` but containing standard error (unless the process was invoked via a pty, in which case it will be empty; see `Runner.run`.)
- **encoding** (*str*) – The string encoding used by the local shell environment.
- **command** (*str*) – The command which was executed.
- **shell** (*str*) – The shell binary used for execution.
- **env** (*dict*) – The shell environment used for execution. (Default is the empty dict, `{}`, not `None` as displayed in the signature.)
- **exited** (*int*) – An integer representing the subprocess' exit/return code.

  Note: This may be `None` in situations where the subprocess did not run to completion, such as when auto-responding failed or a timeout was reached.

- **pty** (*bool*) – A boolean describing whether the subprocess was invoked with a pty or not; see `Runner.run`.
- **hide** (*tuple*) – A tuple of stream names (none, one or both of `('stdout', 'stderr')`) which were hidden from the user when the generating command executed; this is a normalized value derived from the `hide` parameter of `Runner.run`.

For example, `run('command', hide='stdout')` will yield a `Result` where `result.hide == ('stdout',)`; `hide=True` or `hide='both'` results in `result.hide == ('stdout', 'stderr')`; and `hide=False` (the default) generates `result.hide == ()` (the empty tuple.)

Note: `Result` objects' truth evaluation is equivalent to their `ok` attribute's value. Therefore, quick-and-dirty expressions like the following are possible:

```
if run("some shell command"):
    do_something()
else:
    handle_problem()
```

However, remember Zen of Python #2.

New in version 1.0.

`return_code`

An alias for `.exited`.

New in version 1.0.

`ok`

A boolean equivalent to `exited == 0`.

New in version 1.0.

`failed`

The inverse of `ok`.

I.e., `True` if the program exited with a nonzero return code, and `False` otherwise.

New in version 1.0.

`tail`(*stream*, *count=10*)

Return the last `count` lines of `stream`, plus leading whitespace.

Parameters

- **stream** (*str*) – Name of some captured stream attribute, eg `"stdout"`.
- **count** (*int*) – Number of lines to preserve.

New in version 1.3.

`__weakref__`

list of weak references to the object (if defined)

class `invoke.runners.`**`Promise`**(*runner*)

A promise of some future `Result`, yielded from asynchronous execution.

This class' primary API member is `join`; instances may also be used as context managers, which will automatically call `join` when the block exits. In such cases, the context manager yields `self`.

`Promise` also exposes copies of many `Result` attributes, specifically those that derive from `run` kwargs and not the result of command execution. For example, `command` is replicated here, but `stdout` is not.

New in version 1.4.

`__init__`(*runner*)

Create a new promise.

Parameters **runner** – An in-flight `Runner` instance making this promise.

Must already have started the subprocess and spun up IO threads.

join()

Block until associated subprocess exits, returning/raising the result.

This acts identically to the end of a synchronously executed `run`, namely that:

- various background threads (such as IO workers) are themselves joined;
- if the subprocess exited normally, a `Result` is returned;
- in any other case (unforeseen exceptions, IO sub-thread `ThreadException`, `Failure`, `WatcherError`) the relevant exception is raised here.

See `run` docs, or those of the relevant classes, for further details.

`invoke.runners.`**default_encoding**()

Obtain apparent interpreter-local default text encoding.

Often used as a baseline in situations where we must use SOME encoding for unknown-but-presumably-text bytes, and the user has not specified an override.

4.11 tasks

This module contains the core `Task` class & convenience decorators used to generate new tasks.

class `invoke.tasks.`**Call**(*task*, *called_as=None*, *args=None*, *kwargs=None*)

Represents a call/execution of a `Task` with given (kw)args.

Similar to `partial` with some added functionality (such as the delegation to the inner task, and optional tracking of the name it's being called by.)

New in version 1.0.

__init__(*task*, *called_as=None*, *args=None*, *kwargs=None*)

Create a new `Call` object.

Parameters

- **task** – The `Task` object to be executed.
- **called_as** (`str`) – The name the task is being called as, e.g. if it was called by an alias or other rebinding. Defaults to `None`, aka, the task was referred to by its default name.
- **args** (`tuple`) – Positional arguments to call with, if any. Default: `None`.
- **kwargs** (`dict`) – Keyword arguments to call with, if any. Default: `None`.

__weakref__

list of weak references to the object (if defined)

clone(*into=None*, *with_=None*)

Return a standalone copy of this Call.

Useful when parameterizing task executions.

Parameters

- **into** – A subclass to generate instead of the current class. Optional.

- **with** (*dict*) – A dict of additional keyword arguments to use when creating the new clone; typically used when cloning `into` a subclass that has extra args on top of the base class. Optional.

  Note: This dict is used to `.update()` the original object's data (the return value from its *clone_data*), so in the event of a conflict, values in `with_` will win out.

New in version 1.0.

Changed in version 1.1: Added the `with_` kwarg.

clone_data()

Return keyword args suitable for cloning this call into another.

New in version 1.1.

make_context(*config*)

Generate a *Context* appropriate for this call, with given config.

New in version 1.0.

invoke.tasks.**NO_DEFAULT = <object object>**

Sentinel object representing a truly blank value (vs `None`).

class invoke.tasks.**Task**(*body*, *name=None*, *aliases=()*, *positional=None*, *optional=()*, *default=False*, *auto_shortflags=True*, *help=None*, *pre=None*, *post=None*, *autoprint=False*, *iterable=None*, *incrementable=None*)

Core object representing an executable task & its argument specification.

For the most part, this object is a clearinghouse for all of the data that may be supplied to the *@task* decorator, such as `name`, `aliases`, `positional` etc, which appear as attributes.

In addition, instantiation copies some introspection/documentation friendly metadata off of the supplied `body` object, such as `__doc__`, `__name__` and `__module__`, allowing it to "appear as" `body` for most intents and purposes.

New in version 1.0.

__weakref__

list of weak references to the object (if defined)

argspec(*body*)

Returns two-tuple:

- First item is list of arg names, in order defined.
  - I.e. we *cannot* simply use a dict's `keys()` method here.
- Second item is dict mapping arg names to default values or *NO_DEFAULT* (an 'empty' value distinct from None, since None is a valid value on its own).

New in version 1.0.

get_arguments(*ignore_unknown_help=None*)

Return a list of Argument objects representing this task's signature.

Parameters **ignore_unknown_help** (*bool*) – Controls whether unknown help flags cause errors. See the config option by the same name for details.

New in version 1.0.

Changed in version 1.7: Added the `ignore_unknown_help` kwarg.

`invoke.tasks.call(task, *args, **kwargs)`

Describes execution of a `Task`, typically with pre-supplied arguments.

Useful for setting up *pre/post task invocations*. It's actually just a convenient wrapper around the `Call` class, which may be used directly instead if desired.

For example, here's two build-like tasks that both refer to a `setup` pre-task, one with no baked-in argument values (and thus no need to use `call`), and one that toggles a boolean flag:

```
@task
def setup(c, clean=False):
    if clean:
        c.run("rm -rf target")
    # ... setup things here ...
    c.run("tar czvf target.tgz target")

@task(pre=[setup])
def build(c):
    c.run("build, accounting for leftover files...")

@task(pre=[call(setup, clean=True)])
def clean_build(c):
    c.run("build, assuming clean slate...")
```

Please see the constructor docs for `Call` for details - this function's `args` and `kwargs` map directly to the same arguments as in that method.

New in version 1.0.

`invoke.tasks.task(*args, **kwargs)`

Marks wrapped callable object as a valid Invoke task.

May be called without any parentheses if no extra options need to be specified. Otherwise, the following keyword arguments are allowed in the parenthese'd form:

- `name`: Default name to use when binding to a `Collection`. Useful for avoiding Python namespace issues (i.e. when the desired CLI level name can't or shouldn't be used as the Python level name.)
- `aliases`: Specify one or more aliases for this task, allowing it to be invoked as multiple different names. For example, a task named `mytask` with a simple `@task` wrapper may only be invoked as `"mytask"`. Changing the decorator to be `@task(aliases=['myothertask'])` allows invocation as `"mytask"` *or* `"myothertask"`.
- `positional`: Iterable overriding the parser's automatic "args with no default value are considered positional" behavior. If a list of arg names, no args besides those named in this iterable will be considered positional. (This means that an empty list will force all arguments to be given as explicit flags.)
- `optional`: Iterable of argument names, declaring those args to have *optional values*. Such arguments may be given as value-taking options (e.g. `--my-arg=myvalue`, wherein the task is given `"myvalue"`) or as Boolean flags (`--my-arg`, resulting in `True`).
- `iterable`: Iterable of argument names, declaring them to *build iterable values*.
- `incrementable`: Iterable of argument names, declaring them to *increment their values*.
- `default`: Boolean option specifying whether this task should be its collection's default task (i.e. called if the collection's own name is given.)
- `auto_shortflags`: Whether or not to automatically create short flags from task options; defaults to True.

- `help`: Dict mapping argument names to their help strings. Will be displayed in `--help` output. For arguments containing underscores (which are transformed into dashes on the CLI by default), either the dashed or underscored version may be supplied here.
- `pre`, `post`: Lists of task objects to execute prior to, or after, the wrapped task whenever it is executed.
- `autoprint`: Boolean determining whether to automatically print this task's return value to standard output when invoked directly via the CLI. Defaults to False.
- `klass`: Class to instantiate/return. Defaults to *`Task`*.

If any non-keyword arguments are given, they are taken as the value of the `pre` kwarg for convenience's sake. (It is an error to give both `*args` and `pre` at the same time.)

New in version 1.0.

Changed in version 1.1: Added the `klass` keyword argument.

4.12 terminals

Utility functions surrounding terminal devices & I/O.

Much of this code performs platform-sensitive branching, e.g. Windows support.

This is its own module to abstract away what would otherwise be distracting logic-flow interruptions.

`invoke.terminals.`**`WINDOWS = False`**

Whether or not the current platform appears to be Windows in nature.

Note that Cygwin's Python is actually close enough to "real" UNIXes that it doesn't need (or want!) to use PyWin32 – so we only test for literal Win32 setups (vanilla Python, ActiveState etc) here.

New in version 1.0.

`invoke.terminals.`**`bytes_to_read`**`(`*input_*`)`

Query stream `input_` to see how many bytes may be readable.

Note: If we are unable to tell (e.g. if `input_` isn't a true file descriptor or isn't a valid TTY) we fall back to suggesting reading 1 byte only.

Parameters **`input`** – Input stream object (file-like).

Returns `int` number of bytes to read.

New in version 1.0.

`invoke.terminals.`**`character_buffered`**`(`*stream*`)`

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

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

New in version 1.0.

`invoke.terminals.`**`pty_size`**`()`

Determine current local pseudoterminal dimensions.

Returns A `(num_cols, num_rows)` two-tuple describing PTY size. Defaults to `(80, 24)` if unable to get a sensible result dynamically.

New in version 1.0.

`invoke.terminals.`**`ready_for_reading`**`(`*input_*`)`

Test `input_` to determine whether a read action will succeed.

Parameters **`input`** – Input stream object (file-like).

Returns `True` if a read should succeed, `False` otherwise.

New in version 1.0.

`invoke.terminals.`**`stdin_is_foregrounded_tty`**`(`*stream*`)`

Detect if given stdin `stream` seems to be in the foreground of a TTY.

Specifically, compares the current Python process group ID to that of the stream's file descriptor to see if they match; if they do not match, it is likely that the process has been placed in the background.

This is used as a test to determine whether we should manipulate an active stdin so it runs in a character-buffered mode; touching the terminal in this way when the process is backgrounded, causes most shells to pause execution.

Note: Processes that aren't attached to a terminal to begin with, will always fail this test, as it starts with "do you have a real `fileno`?".

New in version 1.0.

4.13 `util`

`class` `invoke.util.`**`ExceptionHandlingThread`**`(`***kwargs*`)`

Thread handler making it easier for parent to handle thread exceptions.

Based in part on Fabric 1's ThreadHandler. See also Fabric GH issue #204.

When used directly, can be used in place of a regular `threading.Thread`. If subclassed, the subclass must do one of:

- supply `target` to `__init__`
- define `_run()` instead of `run()`

This is because this thread's entire point is to wrap behavior around the thread's execution; subclasses could not redefine `run()` without breaking that functionality.

New in version 1.0.

`__init__``(`***kwargs*`)`

Create a new exception-handling thread instance.

Takes all regular `threading.Thread` keyword arguments, via `**kwargs` for easier display of thread identity when raising captured exceptions.

`exception``()`

If an exception occurred, return an *`ExceptionWrapper`* around it.

Returns An *`ExceptionWrapper`* managing the result of `sys.exc_info`, if an exception was raised during thread execution. If no exception occurred, returns `None` instead.

New in version 1.0.

`is_dead`

Returns `True` if not alive and has a stored exception.

Used to detect threads that have excepted & shut down.

New in version 1.0.

`invoke.util.`**`encode_output`**`(`*string*, *encoding*`)`

Transform string-like object `string` into bytes via `encoding`.

Returns A byte-string (`str` on Python 2, `bytes` on Python 3.)

New in version 1.0.

`invoke.util.`**`has_fileno`**`(`*stream*`)`

Cleanly determine whether `stream` has a useful `.fileno()`.

Note: This function helps determine if a given file-like object can be used with various terminal-oriented modules and functions such as `select`, `termios`, and `tty`. For most of those, a fileno is all that is required; they'll function even if `stream.isatty()` is `False`.

Parameters **`stream`** – A file-like object.

Returns `True` if `stream.fileno()` returns an integer, `False` otherwise (this includes when `stream` lacks a `fileno` method).

New in version 1.0.

`invoke.util.`**`helpline`**`(`*obj*`)`

Yield an object's first docstring line, or None if there was no docstring.

New in version 1.0.

`invoke.util.`**`isatty`**`(`*stream*`)`

Cleanly determine whether `stream` is a TTY.

Specifically, first try calling `stream.isatty()`, and if that fails (e.g. due to lacking the method entirely) fallback to `os.isatty`.

Note: Most of the time, we don't actually care about true TTY-ness, but merely whether the stream seems to have a fileno (per *`has_fileno`*). However, in some cases (notably the use of `pty.fork` to present a local pseudoterminal) we need to tell if a given stream has a valid fileno but *isn't* tied to an actual terminal. Thus, this function.

Parameters **`stream`** – A file-like object.

Returns A boolean depending on the result of calling `.isatty()` and/or `os.isatty`.

New in version 1.0.

`invoke.util.`**`task_name_sort_key`**`(`*name*`)`

Return key tuple for use sorting dotted task names, via e.g. `sorted`.

New in version 1.0.

`class` `invoke.util.`**`ExceptionWrapper`**`(`*kwargs*, *type*, *value*, *traceback*`)`

A namedtuple wrapping a thread-borne exception & that thread's arguments. Mostly used as an intermediate between *`ExceptionHandlingThread`* (which preserves initial exceptions) and *`ThreadException`* (which holds 1..N such exceptions, as typically multiple threads are involved.)

4.14 `watchers`

class `invoke.watchers.`**`FailingResponder`**(*pattern*, *response*, *sentinel*)

Variant of *`Responder`* which is capable of detecting incorrect responses.

This class adds a `sentinel` parameter to `__init__`, and its `submit` will raise *`ResponseNotAccepted`* if it detects that sentinel value in the stream.

New in version 1.0.

class `invoke.watchers.`**`Responder`**(*pattern*, *response*)

A parameterizable object that submits responses to specific patterns.

Commonly used to implement password auto-responds for things like `sudo`.

New in version 1.0.

`__init__`(*pattern*, *response*)

Imprint this *`Responder`* with necessary parameters.

Parameters

- **`pattern`** – A raw string (e.g. `r"\[sudo\] password for .*:"`) which will be turned into a regular expression.
- **`response`** – The string to submit to the subprocess' stdin when `pattern` is detected.

`pattern_matches`(*stream*, *pattern*, *index_attr*)

Generic "search for pattern in stream, using index" behavior.

Used here and in some subclasses that want to track multiple patterns concurrently.

Parameters

- **`stream`** (*`unicode`*) – The same data passed to `submit`.
- **`pattern`** (*`unicode`*) – The pattern to search for.
- **`index_attr`** (*`unicode`*) – The name of the index attribute to use.

Returns An iterable of string matches.

New in version 1.0.

class `invoke.watchers.`**`StreamWatcher`**

A class whose subclasses may act on seen stream data from subprocesses.

Subclasses must exhibit the following API; see *`Responder`* for a concrete example.

- `__init__` is completely up to each subclass, though as usual, subclasses *of* subclasses should be careful to make use of `super` where appropriate.
- *`submit`* must accept the entire current contents of the stream being watched, as a Unicode string, and may optionally return an iterable of Unicode strings (or act as a generator iterator, i.e. multiple calls to `yield <unicode string>`), which will each be written to the subprocess' standard input.

Note: *`StreamWatcher`* subclasses exist in part to enable state tracking, such as detecting when a submitted password didn't work & erroring (or prompting a user, or etc). Such bookkeeping isn't easily achievable with simple callback functions.

Note: *`StreamWatcher`* subclasses `threading.local` so that its instances can be used to 'watch' both subprocess stdout and stderr in separate threads.

New in version 1.0.

`submit`(*stream*)

Act on `stream` data, potentially returning responses.

Parameters **`stream`** (*`unicode`*) – All data read on this stream since the beginning of the session.

Returns An iterable of Unicode strings (which may be empty).

New in version 1.0.

Python Module Index

Index