Dodo Commands

Release 0.43.1

Maarten Nieber

Aug 06, 2021

INTRODUCTION

Source code: https://github.com/mnieber/dodo-commands.

CHAPTER

ONE

DODO COMMANDS, WHO AND WHAT IS IT FOR?

The main goal of Dodo Commands is to make it easier to run a variety of scripts and commands from the command line. Below we will describe two scenarios in which Dodo Commands can be useful. In the next chapter, we will explain how Dodo Commands can be set up to support these scenarios.

1.1 Scenario: working with micro-services running in containers

Imagine that you have a Frontend and a Backend microservice running in Docker containers. How would you open a shell in one of these containers? With Dodo Commands you would type *dodo backend.shell* or *dodo frontend.shell*. How about running the backend Python tests and saving the output to an html file? That would be *dodo backend.pytest*.

In general, Dodo Commands tries to reduce typing and mental overhead by compressing commands in the shortest possible alias. These aliases expand to scripts that are either ready-made or custom-made (by you). The scripts access the project-specific dodo configuration to read parameters (such as the pytest html file location).

1.2 Scenario: getting colleagues up-to-speed

Becoming productive in a development environment means being able to easily execute all the tasks that are commonly required. This involves setting up software and developing an efficient workflow. If every developer writes their own set of aliases then you waste of potential for sharing best practices. With dodo commands you can share scripts between projects and between developers, maximizing reuse and promoting a uniform workflow. When a new developer bootstraps their environment by cloning the Dodo Commands configuration, then they will have all the main tasks available to them as convenient and easy to discover aliases. You can group commands into menus that can be accessed by calling *dodo menu*. Moreover, when developers run these commands with the *–confirm* flag then they will be asked for confirmation at each step in the script, giving them the chance to follow what happens in each script.

CHAPTER

TWO

INSTALLATION

2.1 Step 1: Install Dodo Commands

```
# install dodo commands
sudo pip install dodo-commands
```

Tip: On Mac you may need to create a file `~/.distutils.cfg` that sets an empty prefix to prevent errors stating "must supply either home or prefix/exec-prefix – not both".

Tip: Autocompletion is provided for bash, fish and zsh (you need to restart the shell after installation). For activating autocompletion in zsh you need to install the *argcomplete* python package

```
 # deactivate any existing python environment
 deactivate
 pip install argcomplete

and add the code below to .zshrc.
```

```
autoload bashcompinit
bashcompinit
eval "$(register-python-argcomplete dodo)"
```

2.2 Step 2: (Optional) Install virtualenv and git

Some commands depend on the python-virtualenv package. In addition, some of the Dodo commands use git.

```
# install prerequisites
sudo apt-get install python-virtualenv git
```

2.3 Step 3: (Optional) Activate the latest project automatically

To automatically activate the last used Dodo Commands project, call `dodo autostart on`. This creates a `~/.config/fish/conf.d/dodo_autostart.fish` and a `~/.config/bash/conf.d/dodo_autostart.bash` script. Call `dodo autostart off` to turn automatic activation off, this will delete the `dodo_autostart` script. The Fish shell will automatically find the `dodo_autostart.fish` script and run it when the shell starts. To have the same behaviour in Bash, add this line to your `~/.bashrc` file:

```
if [ -f ~/.config/bash/conf.d/dodo_autostart.bash ]; then
    . ~/.config/bash/conf.d/dodo_autostart.bash
fi
```

2.4 Step 4: (Optional) Add fish shell key-bindings for the dial command

If you are using the fish shell then it's highly recommended to add the key-bindings for the (*dial [–group=default] number*) command (click the link for instructions). These key-binding allow you to change the current directory to one of the preset directories (that are configured in the project configuration file), which can really speed up your work-flow in the terminal.

CHAPTER

THREE

SCENARIO: WORKING WITH MICRO-SERVICES

In this scenerio we'll see how Dodo Commands can be used to work with two micro-services. It's probably over-kill to use Dodo Commmands in this simple scenario, but as the project grows bigger, it will start to be worth it. To keep it simple the services are not Dockerized.

Tip: In the tutorials we'll assume that Bash is used. In some cases we will source the output of a Dodo command using `$(dodo <command>)`. In case you are curious what is being sourced, you can run the command without `$()` and source it manually on the command line. If you are using the Fish shell, then you can use `dodo <command> | source` instead of `$(dodo <command>)`. In that case, be sure to also run *dodo global-config setting.shell fish* to tell Dodo Commands that you are using the fish shell.

Tip: Parts of the tutorial that give technical details or explain more advanced features are kept separate from the main text. To not be overwhelmed, it's probably better to skip them on a first read.

3.1 Two simple micro-services

The first micro-service writes the time to a file in the `/tmp` directory, whereas the second micro-service runs a `tail` command that tracks the contents of this file. The source code for these services is found in `part1/before` of the dodo-commands-tutorial <https://github.com/mnieber/dodo-commands-tutorial>`_ repository. We will go ahead and clone the code for this part of the tutorial:

Step 1: Clone the tutorial files

Step 2: Run the example servers

```
cd /tmp
git clone https://github.com/mnieber/dodo-commands-tutorial.git

# Copy part 1 of the tutorial so that we can work with a short path
cp -rf ./dodo-commands-tutorial/part1/before ./tutorial
```

Let's try out the services:

```
cd /tmp/tutorial/writer

# let this run for a few seconds... (or run it in a separate tab and keep it running)
make runserver

cd /tmp/tutorial/reader
```

(continues on next page)

(continued from previous page)

```
# this will print some timestamps that were generated by the writer service
make runserver
```

3.2 Setting up an environment

The next step is to create a Dodo Commands environment for working with our project:

Step 1: Create the environment

Step 2: Inspect the environment

Troubleshooting

```
cd /tmp/tutorial
$(dodo env --init tutorial)

# Check that we are in the "tutorial" environment
dodo which

    tutorial
```

```
# The project dir is where your project lives.
# In this case, it's the directory where we called 'dodo env --init'.
dodo which --project-dir

    /tmp/tutorial

# The configuration directory is where the Dodo Commands configuration files
# for the environment are stored.
dodo which --config-dir

    /tmp/tutorial/.dodo_commands

# The environment directory is where Dodo Commands stores all other information
# about your environment. Usually, you don't need to work with this directory␣
↪directly.
dodo which --env-dir

    ~/.dodo_commands/envs/tutorial

# The (optional) python_env directory contains the virtual Python environment for␣
↪your project.
# In this case, we don't have any
dodo which --python-env-dir

    (nothing here)
```

Tip: If something goes wrong during the creation of the Dodo Commands environment then you can delete the /tmp/tutorial directory and try again. In this case, you should also run *dodo env –forget tutorial* and run the clean up steps that it prints out (for reasons of safety Dodo Commands does not run these cleanup steps automatically).

3.3 Working with the configuration

Each environment contains a set of configuration files:

Step 1: Inspect the configuration files

(Details) Inspect the run-time configuration values

```
# The main configuration file is called config.yaml
dodo which --config

    /tmp/tutorial/.dodo_commands/config.yaml

# Let's take a look at the configuration file:
cat $(dodo which --config)

    ROOT:
      command_path:
      - ~/.dodo_commands/default_project/commands/*
      version: 1.0.0
```

When we print the contents of the configuration, we see that some extra values were added automatically. These values do not appear in the configuration file but they are available at run-time.

```
dodo print-config

    ROOT:
      env_name: tutorial
      command_path:
      - ~/.dodo_commands/default_project/commands/*
      - /some/path/to/dodo_commands/dodo_system_commands
      project_dir: /tmp/tutorial/part1
      config_dir: /tmp/tutorial/part1/.dodo_commands
      version: 1.0.0
```

3.4 Extending the configuration

You can extend the configuration with new keys:

Step 1: Add a new key

Step 2: Inspect

(Details) Using *dodo edit-config*

```
# (bottom of) /tmp/tutorial/.dodo_commands/config.yaml
MAKE:
  cwd: ${/ROOT/project_dir}/writer
```

Now, when we print the contents of the `MAKE` section, we get:

```
dodo print-config MAKE

    cwd: /tmp/tutorial/writer
```

We see that we can interpolate values. In this case `${/ROOT/project_dir}/writer` was interpolated to `/tmp/tutorial/writer`.

Tip: Run the `dodo edit-config` command to open all files in the configuration directory in an editor. Set the `config_editor` field in the global configuration file (`~/.dodo_commands/config`) to the editor you wish to use (we recommend using gedit with the Side Panel enabled).

Note: From here on, we will use the notation ${/FOO/bar} to refer to the `bar` key in the `FOO` section of the configuration file.

3.5 Adding an alias to run the writer service

We'll now create a `mk.py` script that can be used as an alias for running the writer service. This alias will serve as a shortcut to running *make* in the directory of the writer service.

Step 1: Add the mk.py script

Step 2: Extend the command path

Step 3: Inspect

```
cd /tmp/tutorial
mkdir ./commands
touch ./commands/mk.py
```

Add the following code to `mk.py`:

```
from dodo_commands import Dodo

Dodo.parser.add_argument("what")
Dodo.run(["make", Dodo.args.what], cwd=Dodo.get("/MAKE/cwd"))
```

Open `/tmp/tutorial/.dodo_commands/config.yaml` and edit `${/ROOT/command_path}` so it looks like this:

```
ROOT:
  command_path:
  - ~/.dodo_commands/default_project/commands/*
  - ${/ROOT/project_dir}/commands
```

Now when we run `dodo` (without passing any arguments) we get a list of all available commands, and `mk` should be somewhere in that list.

To run the command, let's use the `--confirm` flag so we can check that everything is looking good:

```
dodo mk runserver --confirm

    (/tmp/tutorial/writer) make runserver

    confirm? [Y/n]
```

We see that the command will run `make runserver` in the `/tmp/tutorial/writer` directory, great!

3.6 Using layers to run the reader and writer service

At the moment, the *mk* command operates on the writer service. What if we instead want to run the Makefile of the reader service?

Step 1: Add the mk.py script

Step 2: Add a LAYERS_GROUP

Step 3: Inspect

As a first step to generalize our *mk* command we will move the `${/MAKE}` section to a new configuration file: `server.writer.yaml`. This file should therefore look like this:

```
# /tmp/tutorial/.dodo_commands/server.writer.yaml
MAKE:
  cwd: ${/ROOT/project_dir}/writer
```

Then we add a similar file for the reader:

```
# /tmp/tutorial/.dodo_commands/server.reader.yaml
MAKE:
  cwd: ${/ROOT/project_dir}/reader
```

Tip: Don't forget to remove the *MAKE* section from the main Dodo configuration file. To edit this file, you can run (substituting your favourite editor) *nano $(dodo which –config)*.

Next, we will add a `LAYERS_GROUP` in the main configuration file:

```
# (bottom of) /tmp/tutorial/.dodo_commands/config.yaml
LAYER_GROUPS:
  server:
  - writer
  - reader
```

Now when we call `dodo writer.mk runserver` then Dodo Commands will look for a layer that has the name `writer`. It will find this layer in the `server` group and load the `server.writer.yaml` layer:

```
dodo writer.mk runserver --confirm

    (/tmp/tutorial/writer) make runserver

    confirm? [Y/n]
```

Of course, to run the reader, we can use `dodo reader.mk runserver`.

3.7 Detail sections

Details

The –trace option

Running the services in tmux

Open the adjacent tabs for more detail sections

We saw above the Dodo Commands applies some magic to find out what command you want to run based on the prefixes that you use before the name of the command. To find out what is going on below the surface, use the `--trace` option to print the result of this translation process (without running any commands). For example:

```
dodo reader.mk runserver --trace

    ['/usr/local/bin/dodo', 'mk', 'runserver', '--layer=server.reader.yaml']
```

This tells us that we are actually invoking the command `dodo mk runserver --layer=server.reader.yaml`.

We can group commands in a menu so we can easily run them in a tmux session. First, make sure that tmux is installed on your system. Then, add a `MENU` section to the configuration file like this:

```
# (bottom of) /tmp/tutorial/.dodo_commands/config.yaml
MENU:
  commands:
    server:
    - dodo writer.mk runserver
    - dodo reader.mk runserver
```

When we run `dodo menu --tmux` we'll open a tmux session that show the menu:

```
dodo menu --tmux

    1 [server] - dodo writer.mk runserver
    2 [server] - dodo reader.mk runserver

    Select one or more commands (e.g. 1,3-4) or type 0 to exit:
```

Type `1,2` to run both commands. They will open in separate windows inside the tmux screen.

CHAPTER

FOUR

SCENARIO: LOCAL DEVELOPMENT WITH DOCKER

We will continue the previous scenario (*Scenario: working with micro-services*) by Dockerizing the two services. If you haven't done the steps of the previous scenario, run this to get started:

```
cd /tmp
git clone git@github.com:mnieber/dodo-commands-tutorial.git

# Copy the end state of part 1 of the tutorial
cp -rf ./dodo-commands-tutorial/part1/after ./tutorial

# Create and activate a dodo environment for our project
cd ./tutorial
$(dodo env --init tutorial)
```

4.1 Running the docker-compose file

Our services come with Docker files that we have not used so far.

Step 1: Inspect the docker compose file

Step 2: Run docker-compose up

```
cat /tmp/tutorial/docker-compose.yml

    version: "3"
    services:
      writer:
        image: dodo_tutorial
        build:
          dockerfile: ./Dockerfile
          context: .
        volumes:
          - ./writer:/app
          - ./time.log:/time.log
        working_dir: /app
        command: make runserver
      reader:
        depends_on: [writer]
        image: dodo_tutorial
        volumes:
          - ./reader:/app
          - ./time.log:/time.log
        working_dir: /app
```

(continues on next page)

(continued from previous page)

```
        command: make runserver

cat /tmp/tutorial/Dockerfile

    FROM python:3.7-alpine

    RUN apk add make
```

Let's test if it works:

```
cd /tmp/tutorial
docker-compose up

    Creating network "tutorial_default" with the default driver
    Creating tutorial_writer_1 ... done
    Creating tutorial_reader_1 ... done
    Attaching to tutorial_writer_1, tutorial_reader_1
    reader_1  | echo "Starting reader service"
    reader_1  | Starting reader service
    writer_1  | echo "Starting writer service"
    reader_1  | tail -f ../time.log
    writer_1  | Starting writer service
    reader_1  | 1586270720.460995
```

4.2 Using the docker-compose command

We'd like to be able to bring this docker system up from any directory with the `dodo docker-compose up` command. To facilitate this, we'll create a new configuration layer in `/tmp/tutorial/.dodo_commands/docker.yaml`.

Step 1: Create the layer

Step 2: Enable it

Step 3: Try it out

```
# Create a new file /tmp/tutorial/.dodo_commands/docker.yaml

DOCKER_COMPOSE:
  cwd: ${/ROOT/project_dir}
```

To enable the layer we add it to the `LAYERS` of the main configuration file. Note that this layer is always loaded.

```
# In: /tmp/tutorial/.dodo_commands/config.yaml

LAYERS:
- docker.yaml
```

We can now run `dodo docker-compose up` to bring the stack up. Remember that you can use the `--confirm` flag to see the command before it's executed. You can also use the `--echo` flag for this purpose.

The `docker-compose` command comes standard with Dodo Commands. If you want to see its location and inspect its contents, you can use the `dodo which` command:

```
dodo which docker-compose

    /some/path/to/dodo_docker_commands/docker-compose.py
```

Tip: We could also have added the `DOCKER_COMPOSE` section directly to `config.yaml`. It's up to you to decide when parts of the configuration should be moved to a separate layer file.

4.3 Detail sections

Details

Adding an alias for docker-compose up

Preset docker commands

Open the adjacent tabs for more detail sections

We can add an alias for `docker-compose up` so we don't have to type too much. With this alias we can start the Docker system with `dodo up`:

```
# /tmp/tutorial/.dodo_commands/config.yaml
ROOT:
  # other stuff
  aliases:
    up: docker-compose up
```

Aliases that should be available in any environment can be added to the global configuration file. To find out where this file lives run `dodo which --global-config`. Let's add an alias there for `docker-compose up --detach`:

```
# ~/.dodo_commands/config

[alias]
upd = docker-compose up --detach
```

When we try out the command with `dodo upd` it will start both containers.

Dodo Commands comes with various useful commands to work with Docker containers. For example, `dodo docker-kill` will show you a menu in which you can select the container that you want to kill:

```
dodo docker-kill

    1 - tutorial_writer_1
    2 - tutorial_reader_1
    Select a container:
```

The `dodo docker-exec` command lets you execute a command in a selected docker container.

```
dodo docker-exec --cmd ls

    0 - exit
    1 - tutorial_reader_1
    2 - tutorial_writer_1
```

(continues on next page)

(continued from previous page)

```
    Select a container:
    2
    Makefile                write_periodically.py
```

4.4 Running a command inside a container

We'll now take a look at how we can add a command script that runs inside a Docker container. We'll first add a new command (`mk-greet`) without worrying about Docker. This new command script will run the `greeting` Makefile command.

Step 1: Extend the Makefile

Step 2: Add the command

Step 3: Try the command

```
# In: /tmp/tutorial/writer/Makefile

greeting:
  echo "Hello ${GREETING}"
```

We'll add a `mk-greet.py` script to `/tmp/tutorial/commands` that sets the `GREETING` environment variable and then runs `make greeting`:

```
# In: /tmp/tutorial/commands/mk-greet.py

from dodo_commands import Dodo

Dodo.parser.add_argument("greeting")
Dodo.run(
    ["make", "greeting", "GREETING=%s" % Dodo.args.greeting],
    cwd=Dodo.get("/MAKE/cwd")
)
```

Remember that we have to run this as `dodo writer.mk-greet` so that the `server.writer.yaml` layer is loaded. Let's see what it currently looks like:

```
dodo writer.mk-greet hi --confirm

    (/tmp/tutorial/writer) make greeting GREETING=hi

    confirm? [Y/n]
```

We have a new command script but it is not yet running inside the `tutorial_writer_1` Docker container. Let's fix this.

Step 1: Decorate the command

Step 2: Update ${MAKE}/cwd

Step 3: Inspect

We first need to tell Dodo Commands that the `mk-greet` command is dockerized:

```
# /tmp/tutorial/.dodo_commands/server.writer.yaml
ROOT:
  # other stuff
  decorators:
    docker: [mk-greet]
```

We also need to specify in which container the `mk-greet` command should run:

```
# /tmp/tutorial/.dodo_commands/writer.yaml
DOCKER_OPTIONS:
  mk-greet:
    container: tutorial_writer_1
```

Tip: The keys in the `DOCKER_OPTIONS` take wild-cards, so instead of `mk-greet` we could have used `*`. In our example, this would mean that *any* dockerized script uses the `tutorial_writer_1` container.

Next, we need to update the value of `${/MAKE/cwd}` because it should point to a location in the container (and not in the host machine):

```
# In: /tmp/tutorial/.dodo_commands/writer.yaml

MAKE:
  cwd: /app
```

When we try again we see that the command is prefixed with the proper Docker arguments:

```
dodo writer.mk-greet hi --confirm

    (/tmp/tutorial) docker exec  \
      --interactive --tty  \
      --workdir=/app  \
      tutorial_writer_1  \
      make greeting GREETING=hi

    confirm? [Y/n]

    echo "Hello hi"
    Hello hi
```

4.5 Detail sections

Details

Inferred commands

Open the adjacent tabs for more detail sections

If the `mk-greet` command is only used in combination with the `writer` layer then there is a way to make the call even shorter. We can tell Dodo Commands that the `writer` layer is inferred by the `mk-greet` command:

```
# /tmp/tutorial/.dodo_commands/config.yaml

LAYER_GROUPS:
  server:
```

(continues on next page)

(continued from previous page)

```
  - writer:
      inferred_by: [mk-greet]
  - reader
```

Now we can run `dodo mk-greet hi` instead of `dodo writer.mk-greet hi`:

```
dodo mk-greet hi --trace

    ['/usr/local/bin/dodo', 'mk-greet', 'hi', '--layer=server.writer.yaml']
```

Warning: Because inferred commands are magical, they are also a bit dangerous. For this reason, it's only allowed to use them in the main `config.yaml` configuration file. Using them in layers has no effect. This makes it easier to reason about the configuration.

CHAPTER

FIVE

SCENARIO: USING PROJECT-SPECIFIC SETS OF ALIASES

We will again continue where we left off in part 2 (*Scenario: local development with Docker*). This time we will see how commands that were created in one Dodo Commands environment can be reused in a different environemnt. If you haven't done the steps of the previous scenario, run this to get started:

```
cd /tmp
git clone git@github.com:mnieber/dodo_commands_tutorial.git

# Copy the end state of part 2 of the tutorial
cp -rf ./dodo_commands_tutorial/part2/after ./tutorial

# Create and activate a dodo environment for our project
cd ./tutorial
$(dodo env --init tutorial)
```

5.1 Activating the default Dodo Commands environment

Let's start by deactivating the current `tutorial` environment. You do this by activating the `default` environment:

Step 1: Activate the default environment

Step 2: Inspect

```
# activate default environment
$(dodo env default)

dodo which

    default
```

The default environment is similar to all other environment. Let's check it out:

```
# Print the Dodo Commands environment directory
dodo which --env-dir

    ~/.dodo_commands/envs/default

# Print the project directory
dodo which --project-dir

    ~/.dodo_commands/default_project

# Print the configuration
```

(continues on next page)

(continued from previous page)

```
dodo print-config

    ROOT:
      env_name: default
      version: 1.0.0
      config_dir: ~/.dodo_commands/default_project
      command_path:
      - ~/.dodo_commands/default_project/commands/*
      - /some/path/to/dodo_commands/dodo_system_commands
      project_dir: ~/.dodo_commands/default_project
```

Tip: In the output of *dodo print-config* we see that all command directories in `~/.dodo_commands/default_project/commands/*` are in the command path. This is true for all Dodo Commands environments (unless if you explitly remove this path from the command path).

5.2 Installing more commands

We will now see how additional default commands can be installed.

Step 1: Use `dodo install-commands`

Step 2: Inspect

```
# If you haven't activated the default environment yet, do so now
$(dodo env default)

# Install the dodo-git-commands pip package
dodo install-commands --pip dodo-git-commands --to-defaults --confirm

    (/) python3.5 -m pip install --upgrade --target ~/.dodo_commands/commands dodo_
↪git_commands

    confirm? [Y/n]

    Collecting dodo-git-commands
    Successfully installed dodo-git-commands-0.3.0

    (/) ln -s \
      ~/.dodo_commands/commands/dodo_git_commands \
      ~/.dodo_commands/default_project/commands/dodo_git_commands

    confirm? [Y/n]
```

We see that the commands are installed into the `~/.dodo_commands/commands` directory. Because we passed the `to-default` flag, a symlink to dodo_git_commands is created in `~/.dodo_commands/default_project/commands`:

```
# Print the command path
dodo print-config /ROOT/command_path

    - ~/.dodo_commands/default_project/commands/*
    - /some/path/to/dodo_commands/dodo_system_commands
```

(continues on next page)

(continued from previous page)

```
dodo which git-multi-status

    ~/.dodo_commands/commands/dodo_git_commands/git-multi-status.py
```

5.3 Creating a new environment

Now we'll create a new project in the `~/projects` directory. The new project will have a python virtual environment.

Step 1: Create a new environment

Step 2: Inspect

```
# create a new project with python virtual environment
$(dodo env --create --create-python-env foo)

    Creating project directory ~/projects/foo ... done
```

```
# check that we've switched to the foo environment
dodo which

    foo

# check that we're using the new python virtual environment
which python

    ~/projects/foo/.env/bin/python
```

Tip: You can change the standard location for creating new projects in the `~/.dodo_commands/config` file. You can edit this file or call

```
dodo global-config settings.projects_dir /path/to/projects
```

5.4 Using the mk.py script in the new environment

To use the `mk` command script that we created in the `tutorial` environment, we need to have `/tmp/tutorial/commands` in our command_path. Surely, we can simply add this path to `${/ROOT/command_path}`. The problem with this approach is that we may move the `tutorial` project to a new location, and then the hard-coded path will no longer be correct. In the steps below, we will use an alternative option that is a bit more robust.

Step 1: Run `dodo install-commands`

Step 2: Extend command path

Step 3: Add ${MAKE}

In this step, we use `dodo install-commands` to create a symlink in the global commands directory that points to `/tmp/tutorial/commands`. Note that we use the `--as` option to give a more recognizable name (`dodo_tutorial_commands`) to the new command path.

```
dodo install-commands /tmp/tutorial/commands --as dodo_tutorial_commands --confirm

    (/tmp) ln -s \
      /tmp/tutorial/commands \
      ~/.dodo_commands/commands/dodo_tutorial_commands

    confirm? [Y/n]
```

Now, if we add `~/.dodo_commands/commands/dodo_tutorial_commands` to `${/ROOT/command_path}` then the `mk` command will be found.

```
ROOT:
  # other stuff
  command_path:
  - ~/.dodo_commands/default_project/commands/*
  - ~/.dodo_commands/commands/dodo_tutorial_commands
```

Note: Of course, if the original location of /tmp/tutorial/commands changes, then you still need to update the symlink in the global commands directory, but you won't have to update the command path in every project.

Before we can successfully call `mk`, we should add a `MAKE` section to `config.yaml`, otherwise the command will fail (it expects to find a `MAKE` configuration key):

```
# ~/projects/foo/.dodo_commands/config.yaml
MAKE:
  cwd: /tmp/tutorial/writer
```

5.5 Importing symbols from a command script

So far, we've kept our `mk` script deliberately very simple. Let's refactor it by extracting a function for running `make`. We can then use this function in our `mk-greet` script.

Step 1: Update `mk.py`

Step 2: Update `mk-greet.py`

Step 3: Inspect

```
# In: /tmp/tutorial/commands/mk.py

from dodo_commands import Dodo

def run_make(*what):
    Dodo.run(["make", *what], cwd=Dodo.get("/MAKE/cwd"))

if Dodo.is_main(__name__):
    Dodo.parser.add_argument("what")
    run_make(Dodo.args.what)
```

```
# In: /tmp/tutorial/commands/mk-greet.py

from dodo_commands import Dodo
from dodo_tutorial_commands.mk import run_make
```

(continues on next page)

(continued from previous page)

```
if Dodo.is_main(__name__):
    Dodo.parser.add_argument("greeting")
    run_make("greeting", "GREETING=%s" % Dodo.args.greeting)
```

Note: The import of `run_make` from the `dodo_tutorial_commands` package will succeed because all packages in the `${/ROOT/command_path}` are added to `sys.path` during execution of the command.

Note: You see that we added a line that says `if Dodo.is_main(__name__)`. This replaces the standard line `if __name__ == "__main__"` which doesn't work when executing the script with `dodo mk`. The reason is that during the execution of `dodo ``mk.py` is not the main module.

Note: If the caller of the script uses the `-confirm` flag then they expect to be notified of any action before it's taken. If your script violates this assumption, then you should use `Dodo.is_main(__name__, safe=False)`. This has the effect that the script will not run in combination with `--confirm` (instead, it will stop with an error message).

```
dodo mk-greet stranger

    echo "Hello stranger"
    Hello stranger
```

5.6 Detail sections

Details

Using environments directly

Specifying dependencies in the .meta file

Open the adjacent tabs for more detail sections

In some cases we may want to call a command in a different environment without switching to that environment. For example, we may only want to print its configuration. We can do this by calling its entry-point in `~/.dodo_commands/bin`:

```
# Directly call the entry point of the tutorial environment
~/.dodo_commands/bin/dodo-tutorial which

    tutorial

# We can extend the path to make this easier
export PATH=$PATH:~/.dodo_commands/bin

# Directly call the dodo entry point in the foo environment
dodo-tutorial which

    tutorial
```

Each Dodo command should ideally run out-of-the-box. If the `mk` command needs additional Python packages, you can describe them in a `mk.meta` file:

```
# /tmp/tutorial/commands/mk.meta
requirements:
- dominate==2.2.0
```

We can try this out by importing `dominate` in `mk.py`:

```
# /tmp/tutorial/commands/mk.py

import dominate
from dodo_commands import Dodo

# ... rest of the script stays the same
```

Calling the `mk` command will ask the user for confirmation to install the `dominate` package into the current Python environment:

```
dodo mk greeting

    This command wants to install dominate==2.2.0:

    Install (yes), or abort (no)? [Y/n]

    Collecting dominate==2.2.0
    Successfully installed dominate-2.2.0
    --- Done ---

    (/tmp) make runserver

    confirm? [Y/n]
```

CHAPTER

SIX

SCENARIO: GETTING COLLEAGUES UP-TO-SPEED

If someone joins your project, then it makes sense to share your working environment with them. At the same time, you want working environments to be independent, so that each project member can arrange it to their preferences. This is achieved by having a shared configuration from which you cherry-pick the parts you need.

We will continue where we left off in part 3 (*Scenario: using project-specific sets of aliases*). If you haven't done the steps of the previous scenario, run this to get started:

```
cd /tmp
git clone git@github.com:mnieber/dodo-commands-tutorial.git

# Copy the end state of part 3 of the tutorial
cp -rf ./dodo-commands-tutorial/part3/after ./tutorial

# Create and activate a dodo environment for our project
cd ./tutorial
$(dodo env --init tutorial)
```

6.1 Preparing the configuration files for sharing

Let's start by activating the *tutorial* environment:

```
$(dodo env tutorial)
```

If you followed the tutorial so far (or if you've cloned the latest starting point) then we have a working Dodo Commands configuration in */tmp/tutorial/.dodo_commands*, and some scripts in `/tmp/tutorial/commands`. Let's see how we can share this configuration with colleagues.

Step 1: Add files to git

Step 2: Add shared configuration files

Step 3: Diff

The first step we will take is to move some files into a `src` directory that we can add to git:

```
cd /tmp/tutorial
mkdir src
mv commands/ docker-compose.yml Dockerfile reader/ writer/ time.log src/

cd src
git init
git add *
git commit -m "First commit"
```

(continues on next page)

(continued from previous page)

```
    [master (root-commit) 56f79a1] First commit
    9 files changed, 77741 insertions(+)
    create mode 100644 Dockerfile
    create mode 100644 commands/greet.py
    create mode 100644 commands/mk.meta
    create mode 100644 commands/mk.py
    create mode 100644 docker-compose.yml
    create mode 100644 reader/Makefile
    create mode 100644 time.log
    create mode 100644 writer/Makefile
    create mode 100644 writer/write_periodically.py
```

We're not adding the `.dodo_commands` directory to git, because then the ownership of these files will become a problem. It's better if each developer can tweak the configuration to their liking.

Instead of adding *.dodo_commands* directly to git, we copy these configuration files to a special location (called `extra`) inside the repository and consider this copy to be the shared configuration.

```
cd /tmp/tutorial
mkdir -p ./src/extra/dodo_commands
cp -rf .dodo_commands ./src/extra/dodo_commands/config

cd src
git add *
git commit -m "Add shared configuration files"

    [master de33a3f] Add shared configuration files
    4 files changed, 39 insertions(+), 5 deletions(-)
    create mode 100644 extra/dodo_commands/config/config.yaml
    create mode 100644 extra/dodo_commands/config/docker.yaml
    create mode 100644 extra/dodo_commands/config/server.reader.yaml
    create mode 100644 extra/dodo_commands/config/server.writer.yaml
```

We also add the `${/ROOT/config/shared_config_dir}` key to tell Dodo Commands where the shared configuration files are:

```
ROOT:
  # ...
  shared_config_dir: ${/ROOT/src_dir}/extra/dodo_commands/config
```

Now, we can compare our local configuration files to the shared files as follows:

```
dodo diff --confirm

    (/tmp) meld \
      /tmp/tutorial/src/extra/dodo_commands/config \
      /tmp/tutorial/.dodo_commands/.
```

When you run this command then `meld` will tell us that the `config.yaml` file has changed (remember, we added a `shared_config_dir` key to the `ROOT` section). You can double click on this file to get a detailed view of the differences, and copy the local changes over to the shared file. Since we are updating the shared configuration, we should also bump the `${/ROOT/version}` key in both files to indicate the version change. Finally, you can add the changes in `/tmp/tutorial/src/extra/dodo_commands/config/config.yaml` to git and commit them:

```
cd /tmp/tutorial/src
git add *
git commit -m "Update shared configuration files"

    [master 256a23b] Update shared configuration files
    1 file changed, 3 insertions(+), 1 deletion(-)
```

Note: The purpose of `${/ROOT/version}` is to track the version of the configuration file. If the version in the local file is smaller than the version in the shared file, then it means that your colleague added something to the shared file. In this case, use `dodo diff` to synchronize your local file with the shared one. When you are done, make sure to edit the `${/ROOT/version}` value so that it's the same as the value in the shared file (this marks the fact that you are up-to-date with the shared configuration).

6.2 Bootstrapping a Dodo Commands environment

We are now ready to let a colleague work on our project. To similate the steps that our colleague will take on their computer, we will create an environment named *colleague* and pretend that this is the "tutorial" environment that they will use. We will use the `bootstrap` command to initialize it. This will provide our colleague with a copy of the configuration files that we just added to git.

```
$(dodo env --create colleague)
dodo bootstrap --git-url=/tmp/tutorial/src src extra/dodo_commands/config --confirm

    (/tmp) mkdir -p /home/maarten/projects/colleague

    confirm? [Y/n]

    (/tmp) cp -rf \
      ~/projects/colleague/src/extra/dodo_commands/config/config.yaml
      ~/projects/colleague/.dodo_commands/config.yaml

    Copying shared environment files to your local environment...

    Warning, destination path already exists: ~/projects/colleague/.dodo_commands/
↪config.yaml. Overwrite it?
    confirm? [Y/n]

    (/tmp) cp -rf
      ~/projects/colleague/src/extra/dodo_commands/config/server.writer.yaml
      ~/projects/colleague/.dodo_commands/server.writer.yaml
    confirm? [Y/n]

    (/tmp) cp -rf
      ~/projects/colleague/src/extra/dodo_commands/config/server.reader.yaml
      ~/projects/colleague/.dodo_commands/server.reader.yaml
    confirm? [Y/n]

    (/tmp) cp -rf
      ~/projects/colleague/src/extra/dodo_commands/config/docker.yaml
      ~/projects/colleague/.dodo_commands/docker.yaml
    confirm? [Y/n]
```

Because we used the `--confirm` flag, the command asks permission to copy the shared configuration files to our

local configuration directory. Let's look at the arguments that were supplied in the call to `bootstrap`:

- We used a `--git-url` that points to our local git repository. Usually you would use a remote git url.
- The repository is cloned to the `src` subdirectory of colleague's project directory.
- The shared configuration files are copied from the `extra/dodo_commands/config` location (which is relative to `src`) to the configuration directory of colleague.

6.3 Detail sections

Details

Checking the config version

Checking the Dodo Commands version

Alternatives to git as the starting point

Open the adjacent tabs for more detail sections

When your colleague changes their local configuration files, they may decide at some point to contribute these changes to the shared configuration files. Hopefully, they will also bump the `${/ROOT/version}` value when they do. Whenever you pull the git repository on which you both work, you can run the `dodo check-version --config` command to find out if the shared configuration has changed. This command compares the `${/ROOT/version}` value in your local configuration with the value in the shared configuration. Then, use `dodo diff` to synchronize any changes.

If you are using the `autostart` script to enable the last used environment automatically when opening a shell, then this checks happens automatically.

The (optional) `${/ROOT/required_dodo_commands_version}` value is used to check that you have the right version of Dodo Commands. The call `dodo check-version --dodo` verifies this.

If you are using the `autostart` script to enable the last used environment automatically when opening a shell, then this checks happens automatically.

In the steps above, we cloned a git repository to obtain a `src` directory that has the shared configuration files. However, there are other ways to obtain these files. First of all, you can obtain the `src` directory from a cookiecutter template:

```
dodo bootstrap --cookiecutter-url https://github.com/foo/foobar.git src extra/dodo_
↪commands/config
```

Note that the cookiecutter url can also point to a directory on the local filesystem. Second, when you already have a checked out monolithic source tree, then you can use any subdirectory of this tree as the `src` directory of your new project:

```
dodo bootstrap --link-dir ~/sources/monolith/foobar src extra/dodo_commands/config
```

Note that both examples look very similar to the case where git was used.

CHAPTER

SEVEN

GLOBAL CONFIGURATION

The global configuration is stored in `~/.dodo_commands/config`.

7.1 The settings section

```
[settings]
# the location where your projects are stored (defaults to ~/projects)
project_dir = ~/projects

# the python interpreter that is used
# a) in virtual environments created by Dodo Commands and
# b) to install new Dodo Commands pip packages into the commands directory
python = python3.5

# the diff tool used to show changes to your project configuration files.
# It's recommended to install and use ``meld`` for this option.
diff_tool = meld
```

7.2 The alias section

This section contains aliases for any dodo command, e.g.

```
[alias]
wh = which
whpp = which --projects
```

CHAPTER

EIGHT

GLOBALLY USED DIRECTORIES AND FILES

The following files are accessible by all Dodo Commands environments:

8.1 global configuration file

Value: `~/.dodo_commands/config`. Ini file that contains the global configuration settings

8.2 global commands directory

Value: `~/.dodo_commands/commands`. The directory that contains all globally installed command packages

8.3 default project directory

Value: `~/.dodo_commands/default_project`. The directory that contains the project files for the default environment.

8.4 default commands directory

Value: `~/.dodo_commands/default_project/commands`. The directory that contains the default command packages (these are symlinks to packages in the global commands directory)

8.5 global bin directory: ~/.dodo_commands/bin

A directory that contains an executable per environment (e.g. dodo-foo) for using that environment directly. Calling `dodo-<name> which` will always return `<name>`.

CHAPTER

NINE

CONFIGURATION RULES

The configuration file uses the yaml format, with a few extra rules:

1. environment variables (such as `$PATH`) in values are expanded automatically. Note that unresolved variables of the shape `$FOO` are ignored, whereas variables of the shape `${FOO}` will result in an error if they are not resolved.

2. values may reference other values in the configuration file:

```
BUILD:
    nr_of_threads: 2
FOO:
    bar: ${/BUILD/nr_of_threads}       # value will be the number 2
    ${/BUILD/nr_of_threads}_foo: baz   # key will be the string "2_foo"
```

3. the following magic values are automatically added: `${/ROOT/project_name}`, `${/ROOT/project_dir}`, `${/ROOT/res_dir}`. Finally the dodo_system_commands directory is automatically added to `${/ROOT/command_path}`.

4. `${/ROOT/LAYERS}` lists additional yaml files that are layered on top of the root configuration file. If a key exists both in the root configuration and in the layer, then values replace values, lists are concatenated, and dictionaries are merged. Layers are found relative to the resources directory that holds all configuration files. However, layers may also be prefixed with an absolute path. Moreover, wildcards are allowed. To list all active layers, use `dodo which --layers`.

```
ROOT:
  layers:
    # contents of this file are layered on top of this configuration
    - buildtype.debug.yaml
    # layer with an absolute path
    - ~/.dodo_commands/default_layer.yaml
    # example of using wildcards
    - ~/.dodo_commands/default_layers/*.yaml
```

Layers can be switched on and off with the `dodo layer` command (except for the ones with absolute paths). In the above example, to replace the layer `buildtype.debug.yaml` with `buildtype.release.yaml` call:

```
dodo layer buildtype release
```

5. All files in `{/ROOT/dotenv_files}` are loaded with `python-dotenv` and used in the expansion of environment variables in the Dodo configuration. Note that these values are not added to the environment during the execution of a command script.

CHAPTER

TEN

CONFIGURATION FILE

10.1 ROOT section

10.1.1 command_path (list)

Search path for finding command scripts. The `dodo_system_commands` module is added to the command path by default.

10.1.2 command_path_exclude (filename pattern)

Exclude matching paths from the command path

10.2 LAYER_GROUPS section

Contains a map from group-name to layer properties, e.g.

```
LAYER:GROUPS:
  server:
  - reader:
      inferred_commands: [greet]
      name: rdr
      target_path: /foo/bar.yaml
  - writer: {}
```

CHAPTER

ELEVEN

THE DODO ENTRY POINT

The `dodo` command runs a Dodo Commands script.

11.1 dodo –layer=<name> –trace –traceback

11.1.1 –layer=<name>

Adds <name> as a layer to the configuration.

Note: The layers `foo.bar.yaml` and `foo.baz.yaml` are considered to be mutually exclusive variations of the `foo` layer. Therefore, the use of `--layer foo.baz.yaml` will nullify any layer such as `foo.bar.yaml` in `${/ROOT/layers}`.

11.1.2 –trace

Instead of running the command, prints an array that contains the final form (after interpretation) of all the arguments

11.1.3 –traceback

Instead of writing a short error when a command fails, writes the full stacktrace.

CHAPTER

TWELVE

THE DODO SINGLETON

The Dodo singleton class gives Dodo Commands scripts access to the core Dodo Commands functions.

12.1 Methods

12.1.1 The is_main function

Using `if Dodo.is_main(__name__)` instead of the usual `if __name__ == '__main__'` allows Dodo Commands to execute your script when its invoked through calling `dodo`.

```
from dodo_commands.framework import Dodo

if Dodo.is_main(__name__):
    print("Hello world")
```

12.1.2 The get function

Calling `Dodo.get('/ROOT/my/key', 'default-value')` will retrieve a value from the project's configuration. Use `Dodo.get()` to get direct access to the entire configuration dictionary.

12.1.3 The parse_args function (–confirm and –echo)

The `Dodo.parse_args(parser)` function uses `parser` to parse the arguments in `sys.argv`. It adds an `--echo` and `--confirm` flag to the command line arguments of your script:

1. the `--echo` flag changes the behaviour of `run` so that it only prints a command line instead of executing the command.
2. the `--confirm` flag changes the behaviour of `run` so that it prints a command line and asks for confirmation before executing the command.
3. if you use the `--confirm` flag twice then also nested `dodo` calls (i.e. any calls to `dodo` that are executed inside the Dodo Command script) will ask for confirmation.

If you call `Dodo.parse_args` then you should do so before any calls to `Dodo.run` so that `--echo` and `--confirm` can take effect:

```
from dodo_commands.framework import Dodo
from argparse import ArgumentParser

```

(continues on next page)

(continued from previous page)

```
def _args():
    parser = ArgumentParser()
    parser.add_argument('--verbose')
    return Dodo.parse_args(parser)

if Dodo.is_main(__name__):
    args = _args()

    if args.verbose:
        Dodo.run(
            ['echo', 'hello world'],
            cwd=Dodo.get('/ROOT/src_dir')
        )
```

12.1.4 The run function

The `Dodo.run` function takes a list of arguments (and a current working directory) and runs them on the command line. It also adds all variables in `${/ENVIRONMENT/variable_map}` to the system environment for the duration of running the command.

```
if Dodo.is_main(__name__):
    Dodo.run(['echo', 'hello'], cwd='/tmp')
```

12.2 Config arguments

Although it's possible to use `Dodo.get` directly inside the `Dodo.run` invocation, doing this work in the `_args()` helper function yields a better separation of concerns:

```
def _args():
    Dodo.parser.add_argument('--verbose')
    args.src_dir = Dodo.get('/ROOT/src_dir')
    return Dodo.args

if Dodo.is_main(__name__):
    args = _args()
    # You can now use args.src_dir
```

This approach opens up an interesting possibility: if the requested configuration key is absent then we could still ask the user for a value on the command line. This can be achieved through the `ConfigArg` helper class:

```
from dodo_commands.framework import Dodo, ConfigArg
from argparse import ArgumentParser

def _args():
    parser = ArgumentParser()
    parser.add_argument('--verbose')
    return Dodo.parse_args(parser, config_args=[
      '/ROOT/src_dir', 'src_dir', help="Location of the source files"
    ])
```

The `ConfigArg` is constructed with the configuration key, followed by any (keyword) arguments that `parser.add_argument` accepts. If the key is found in the configuration, then the corresponding value will be inserted into

the return value of `Dodo.parse_args`. Otherwise, an extra *argument* will be added to the command line syntax. This ensures that the value is either read from the configuration or from the command line.

You may also use an expression such as `${/ROOT/project_dir}/foo/bar` as the first argument of ConfigArg. In this case, the config argument is considered to exist if the expression can be fully expanded.

12.3 Using pipes and redirection

Since pipes and redirection are handled by the shell, you need to explicitly mention the shell executable to use them, e.g.

```
if Dodo.is_main(__name__):
    args = _args()
    Dodo.run(['sh', '-c', 'echo \'Hello world\' > /tmp.out'])
```

The `sh_cmd` function offers a shortcut for this:

```
from dodo_commands import Dodo, sh_cmd

if Dodo.is_main(__name__):
    args = _args()
    Dodo.run(sh_cmd('echo \'Hello world\' > /tmp.out'))
```

12.4 Marking a script as unsafe

Since command scripts are written in Python, the script can in principle perform any operation without explicitly asking your permission. In other words, it may choose to ignore the `--confirm` and `--echo` options. This sitation should of course be avoided. However, if a Command script does not completely honor the `--confirm` and `--echo` flags, it should pass `safe=False` when it calls `Dodo.is_main`, as shown in the example below. Unsafe commands will not run with the –echo flag, and will pause with a warning when run with the –confirm flag.

```
if Dodo.is_main(__name__, safe=False): # NOTE: setting the _safe flag here
    # Do destructive things without asking permission. Having this call
    # is the reason we used safe=False to mark the script as unsafe.
    # Running the script with ``--echo`` is not possible, because that would
    # lead to unpleasant surprises. Running with ``--confirm`` will inform
    # you that unpleasant surprises can be expected if you continue.
    os.unlink('/tmp/foo.text')

    # Delete the /tmp directory. Since this time we are using Dodo.run,
    # the user can use the --confirm flag to inspect and cancel it.
    # This makes this call *relatively* safe, but if you blindly run this script␣
↪(without
    # using ``--confirm``) you may still be unpleasantly surprised.
    Dodo.run(['rm', '-rf', '/tmp'])
```

CHAPTER

THIRTEEN

DECORATORS

A Decorator is a class that alters the workings of a Dodo Command script by extending or modifying the arguments that are passed to `Dodo.run`. For this purpose, the arguments for `Dodo.run` are constructed as a tree. Initially, this tree only has a root node that holds all arguments. Each decorator may restructure the argument tree by adding new tree nodes and changing existing nodes. Dodo Commands obtains the final list of arguments by flattening the tree in a pre-order fashion. This allows each decorator to prepend or append arguments, or completely rewrite the tree.

13.1 Constructing a tree

Each node in the tree is of type `ArgsTreeNode`. A node has attributes `args` (these are the command line arguments) and `is_horizontal` (this flag determines how the arguments are printed when the `--echo` or `--confirm` flag is used). To add a child node, call `node.add_child()`.

13.2 Prepending an argument

The following example shows a decorator that prepends the arguments with the path to a debugger executable. The decorator should be placed in a `decorators` directory inside a commands directory:

```
# file: my_commands/decorators/debugger.py

class Decorator:  # noqa
    def add_arguments(self, parser):  # noqa
        parser.add_argument(
            '--use-debugger',
            action='store_true',
            default=False,
            help="Run the command through the debugger"
        )

    def modify_args(self, dodo_args, root_node, cwd):  # noqa
        if not getattr(dodo_args, 'use_debugger', False):
            return root_node, cwd

        # Create a new root node with just the path to the debugger
        # Note that "debugger" is a name tag which is only used internally
        # to identify nodes in the tree.
        debugger_node = ArgsTreeNode(
            "debugger", args=[Dodo.get_config('/BUILD/debugger')]
        )
```

(continues on next page)

(continued from previous page)

```
        # Since we want to make the debugger path a prefix, we add the
        # original arguments as a child node. When the tree is flattened
        # in a pre-order fashion, this will give the correct result.
        debugger_node.add_child(root_node)
        return debugger_node, cwd
```

Note that the decorator returns both the new argument tree and a current working directory. This means that it's possible to change the current working directory for the decorated command as well.

13.3 Appending an argument

This is similar to prepending, except that we do not need to create a new node

```
# file: my_commands/decorators/foo.py

class Decorator:  # noqa
    def modify_args(self, dodo_args, root_node, cwd):  # noqa
        root_node.args.append('--foo')
        return root_node, cwd
```

13.4 Mapping decorators to commands

Not all decorators are compatible with all commands. For example, only some commands can be run inside a debugger. Therefore, the configuration contains a list of decorated command for each decorator. In this list, wildcards are allowed, and you can exclude commands by prefixing them with an exclamation mark:

```
ROOT:
  decorators:
    # Use a wildcard to decorate all commands, but exclude the foo command
    debugger: ['*', '!foo']
    # The cmake and runserver scripts can be run inside docker
    docker: ['cmake', 'runserver']
```

13.5 Using DecoratorScope in scripts

The `DecoratorScope` context manager makes it possible to enforce the use of a decorator inside a section of the the command script:

```
from dodo_commands import DecoratorScope

with DecoratorScope("docker"):
    # This command will run inside docker
    Dodo.run(["ls"])
```

It's also possible to force a decorator to be disabled. In this case, even if the command was decorated, the `Dodo.run` calls inside `DecoratorScope` will not use the decorator:

```
from dodo_commands import DecoratorScope

with DecoratorScope("docker", remove=True):
    # This command will not run inside docker
    Dodo.run(["ls"])
```

13.6 Printing arguments

The structure of the argument tree determines how arguments are printed when the `--echo` or `--confirm` flag is used. We've seen above that nodes in the tree are created with the `ArgsTreeNode` constructor. The arguments in this node are indented in correspondence to the node's depth in the tree. The `ArgsTreeNode` constructor takes an optional argument `is_horizontal` that determines if arguments are printed horizontally or vertically, e.g.

```
docker_node = ArgsTreeNode("docker", args=['docker', 'run'])
tty_node = ArgsTreeNode(
    ["tty", args=['--rm', '--interactive', '--tty'],
    is_horizontal=True
)
docker_node.add_child(tty_node)

# add more nodes to the tree...
```

```
# assume cmake is decorated with the docker decorator
dodo cmake --echo
```

produces

```
docker run  \
  --rm --interactive --tty  \
  --name=cmake  \
  dodo_tutorial:1604  \
  cmake -DCMAKE_BUILD_TYPE=release /home/maarten/projects/dodo_tutorial/src
```

CHAPTER

FOURTEEN

SHELL COMMANDS

A shell command file should have `dodo.` as a prefix, e.g. `dodo.my-command.sh`. For the rest, the command file should be a normal shell script.

CHAPTER

FIFTEEN

ENV [–INIT/–CREATE/–FORGET] –CREATE-PYTHON-ENV [–LATEST <NAME>]

This command returns a string that - when sourced - activates the <name> environment.

Arguments:

15.1 –create

Creates a new project directory in the global projects directory. Also creates a new Dodo Commands environment in the global environments directory.

15.2 –init

Creates a new Dodo Commands environment in the global environments directory. The current directory is taken as the project directory in this environment.

15.3 –forget

Removes <name> from the global environments directory

15.4 –create-python-env

Create a new Python virtual environment inside the project directory.

15.5 –use-python-env=<path>

Register the existing Python virtual environment inside the Dodo Commands environment.

15.6 –latest

Activates the latest used environment

15.7 <name>

Name of the environment. If − is used then the previously used environment is activated.

CHAPTER

SIXTEEN

GLOBAL-CONFIG

Used to change a setting in the global configuration, e.g. `dodo global-config settings.diff_tool meld`

CHAPTER

SEVENTEEN

INSTALL-COMMANDS

Installs Python packages with command scripts into the global commands directory.

17.1 –to-defaults

Adds the installed packages to the default commands directory.

17.2 –make-default

Adds an already installed global commands package to the default commands directory

17.3 –remove

Removes a package from the global commands directory

17.4 –pip

Installs commands from a pip package

17.5 –as <dirname>

Installs to a directory with <dirname>

17.6 /path/to/package

Install a local Python package into the global commands directory

CHAPTER

EIGHTEEN

AUTOSTART [ON OFF]

Writes (or removes) a small shell script in `~/.config/fish/conf.d` and `~/.config/bash/conf.d`. When this small script is sourced, it activates the last used environment. In Bash, this requires that you add these lines to the *~/.bashrc`* file:

```
if [ -f ~/.config/bash/conf.d/dodo_autostart.bash ]; then
    . ~/.config/bash/conf.d/dodo_autostart.bash
fi
```

CHAPTER

NINETEEN

COMMIT-CONFIG

Breaking your local configuration can be serious problem, because it stops all Dodo Commands from working. Therefore, it's advisable to store your local configuration in a local git repository so that you can always restore a previous version. The `dodo commit-config` command makes this easy. It initializes a local git repository (if one doesn't exist already) next to your configuration files, and stages and commits all changes to the configuration.

CHAPTER

TWENTY

DIAL [–GROUP=DEFAULT] NUMBER

Selects directory from a list in `/DIAL` and prints it to the console, e.g.

```
DIAL:
  default:
    # used as a short-cut for using a Dodo Commands layer
    '1': dodo backend.
    '2': dodo frontend.
  shift:
    # used for cd'ing into a directory
    '1': ${/ROOT/src_dir}/backend
    '2': ${/ROOT/src_dir}/frontend
```

```
# prints "/path/to/backend/"
dodo dial 1

# cd's into "/path/to/backend/"
cd $(dodo dial --group=shift 1)
```

This becomes useful when combined with a key binding in the shell. For example, in Fish the following binding allows you to go to a directory (in the `${/DIAL/default}` group) with F1 and F2, and insert a string (in the `${/DIAL/shift}` group) with Shift+F1 and Shift+F2, allowing for a very fast work-flow:

```
# /.config/fish/functions/dial_insert.fish
function dial_insert
    set cmd (dodo dial -g default $argv)
    if test $cmd
        commandline -i $cmd
    end
end

# /.config/fish/functions/dial_cd.fish
function dial_cd
    cd (dodo dial $argv)
    commandline -f repaint
end

# /.config/fish/functions/fish_user_key_bindings.fish
function fish_user_key_bindings
    bind --key f1 "dial_insert 0"
    bind --key f2 "dial_insert 1"

    bind \e\[1\;2P "dial_cd 1"
    bind \e\[1\;2Q "dial_cd 2"
end
```

Tip: Dodo Commands comes with "dial" key bindings for fish. You can install them as follows:

```
# find location of the which.py script, e.g.
dodo which --fish-config

# There you will find subdirectories called ``functions`` and ``conf.d`` that need
# to be copied to ~/.config/fish (merging them with the existing directories there)
# If ~/.config/fish does not yet contain these subdirectories then you can do this␣
↪using:
cd (dodo which --fish-config)
cp -rf functions conf.d ~/.config/fish
```

20.1 group[=default]

The group inside `${/DIAL}` from which to pick.

The number to dial.

20.2 number

The number to dial.

CHAPTER

TWENTYONE

DROP-IN

When you install a package with `dodo install-commands` it may contain more than just command scripts. Some packages contain a so-called "drop-in" directory with configuration files and other resources such as Dockerfiles. Since the Dodo Commands philosophy is that you own your local configuration, the way to use these files is through copying them:

```
dodo install-commands --pip dodo-deploy-commands
# copy drop-in directory to ${/ROOT/res_dir}/drops/dodo_deploy_commands
dodo drop-in dodo_deploy_commands
```

The `dodo drop-in` command copies the package's "drop-in" directory to `${/ROOT/res_dir}/drops/<package_name>`. The default location of the `drop-in` source directory is in the root of the package. Alternatively, the package root may contain a `.drop-in` file that holds the relative path to the actual `drop-in` directory.

You can use a copied configuration file by including it as a layer:

```
# enable layer (drop.on.yaml)
dodo layer drops/dodo_deploy_commands/drop on
# disable layer (drop.off.yaml)
dodo layer drops/dodo_deploy_commands/drop off
```

CHAPTER

TWENTYTWO

LAYER LAYER-GROUP LAYER-VALUE

Selects a layer from the chosen group and adds it to `/LAYERS`. E.g. `dodo layer server foo` will add server.foo.yaml to `LAYERS`.

Note: Note that this makes a change to your main configuration file. Make sure that you do not have any unsaved configuration changes before calling this command.

Arguments:

- layer-group: selected group
- layer-value: selected layer within the selected layer-group

CHAPTER

TWENTYTHREE

THE DOCKER DECORATOR

If the "docker" decorator is used, then all command lines will be prefixed with `/usr/bin/docker run` and related docker arguments found in `$(/DOCKER_OPTIONS/<pattern>`. Here, `<pattern>` matches the name of the current dodo command. For example, consider this configuration:

```
DOCKER_OPTIONS:
  # * will match any name
  '*':
    image: foobar:base
    volume_map:
      ${/ROOT/src_dir}: /srv/foobar/src
  # docker options when running the 'django-manage' command
  'django-manage':
    extra_options:
    - '--publish=127.0.0.1:27017:27017'
```

Running the `django-manage` command will produce something like this:

```
# note that docker options for django-manage are looked up in ${/DOCKER_OPTIONS}
# where it will match with '*' and 'django-manage'
dodo django-manage --echo

# outputs:
docker run
    --rm --interactive --tty
    --name=django-manage
    --volume=/home/maarten/projects/foobar/src:/srv/foobar/src
    --publish=127.0.0.1:80:80
    foobar:base
    python manage.py
```

23.1 ${/DOCKER_OPTIONS/<pattern>/image}

Identifies the docker image.

23.2 ${/DOCKER_OPTIONS/<pattern>/name}

Used to name the docker docker container (defaults to the name of the dodo command).

23.3 ${/DOCKER_OPTIONS/<pattern>/volume_map}

Each key-value pair will be added as a docker volume (where 'key' in the host maps to 'value' in the docker container)

23.4 ${/DOCKER_OPTIONS/<pattern>/volume_map_strict}

Each key-value pair is added as a docker volume. If the key does not exist as a local path, an error is raised.

23.5 ${/DOCKER_OPTIONS/<pattern>/volume_list}

Each item will be added as a docker volume (where 'item' in the host maps to 'item' in the docker container)

23.6 ${/DOCKER_OPTIONS/<pattern>/publish_map}

Each key-value pair will be added as a docker published port

23.7 ${/DOCKER_OPTIONS/<pattern>/publish_list}

Each item will be added as a docker published port

23.8 ${/DOCKER_OPTIONS/<pattern>/volumes_from_list}

Each item will be added as a docker "volumes_from" argument

23.9 $(/DOCKER_OPTIONS/<pattern>/link_list}

Each item will be added as a docker "link" argument

23.10 $(/DOCKER_OPTIONS/<pattern>/variable_list}

Each environment variable will be added as an environment variable in the docker container. Variables in `variable_list` have the same name in the host and in the container.

23.11 $(/DOCKER_OPTIONS/<pattern>/variable_map}

Each key-value pair will be added as an environment variable in the docker container.

23.12 ${/DOCKER_OPTIONS/<pattern>/extra_options}

Key-value pairs are passed as extra options to the docker command line call.

23.13 $(/ENVIRONMENT/variable_map}

Each key-value pair will be added as an environment variable in the docker container.

23.14 $(/DOCKER_OPTIONS/<pattern>/rm}

Decides if the docker container is automatically removed (defaults to True).

23.15 $(/DOCKER_OPTIONS/<pattern>/is_interactive}

Decides if the `-i` and `-t` flags are added.

23.16 Matching multiple names

It's possible to match multiple names using a list:

```
DOCKER_OPTIONS:
  ['django-manage', 'django-runserver']:
    extra_options:
    - '--publish=127.0.0.1:27017:27017'
```

Patterns starting with '!' indicate names that should be excluded:

```
DOCKER_OPTIONS:
  # match django-manage but not django-runserver
  ['django-*', '!django-runserver']:
    extra_options:
    - '--publish=127.0.0.1:27017:27017'
```

CHAPTER

TWENTYFOUR

DOCKER-BUILD

When referring to a docker image in `${/DOCKER_OPTIONS/<pattern>/image}`, you may also need to ensure this image is built. The details for building an image are specified in `${/DOCKER_IMAGES}`:

```
DOCKER_IMAGES:
  'base':
    image: foobar:base
    build_dir: ${/ROOT/src_dir}/docker/base
```

Running `dodo docker-build base` builds the image:

```
dodo docker-build --confirm base

# outputs something like:
(/home/maarten/projects/foobar/src/docker/base) docker build -t foobar:base -f␣
↪Dockerfile .

continue? [Y/n]
```

CHAPTER

TWENTYFIVE

DOCKER-EXEC –CMD

To inspect a running docker container, run `dodo docker-exec`. This will print a list of running containers, allowing you to select one.

25.1 –cmd

The command to run. Defaults to a shell giving you access to the container.

25.2 $(/DOCKER/default_shell

The shell to open when no `--cmd` is supplied in `dodo docker-exec`.

CHAPTER

TWENTYSIX

DOCKER-KILL

Similar to how `dodo docker-exec` works, this command kills a selected docker container.