RiotKit Do Documentation

Release 1

Wolnosciowiec Team

May 06, 2022

Contents:

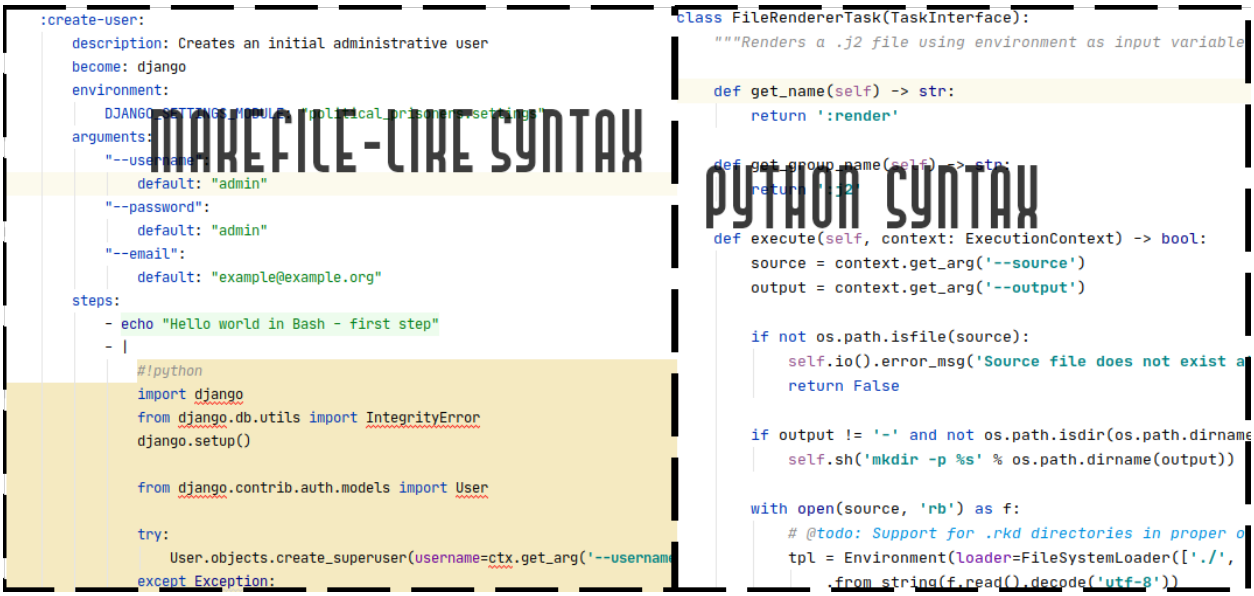
RKD is a stable, open-source, multi-purpose automation tool which balance flexibility with simplicity. The primary language is Python and YAML syntax.

RiotKit-Do can be compared to **Gradle** and to **GNU Make**, by allowing both Python and Makefile-like YAML syntax.

What I can do with RKD?

- Simplify the scripts
- Put your Python and Bash scripts inside a YAML file (like in GNU Makefile)
- Do not reinvent the wheel (argument parsing, logs, error handling for example)
- Share the code across projects and organizations, use native Python Packaging to share tasks (like in Gradle)
- Natively integrate scripts with .env files
- Automatically generate documentation for your scripts
- Maintain your scripts in a good standard

RKD can be used on PRODUCTION, for development, for testing, to replace some of Bash scripts inside docker containers, and for many more, where Makefile was used.



CHAPTER 1

Example use cases

- Docker based production environment with multiple configuration files, procedures (see: Harbor project)
- Database administrator workspace (importing dumps, creating new user accounts, plugging/unplugging databases)
- Development environment (executing migrations, importing test database, splitting tests and running parallel)
- On CI (prepare project to run on eg. Jenkins or Gitlab CI) - RKD is reproducible on local computer which makes inspection easier
- Kubernetes/OKD deployment workspace (create shared YAML parts with JINJA2 between multiple environments and deploy from RKD)
- Automate things like certificate regeneration on production server, RKD can generate any application configs using JINJA2
- Installers (RKD has built-in commands for replacing lines in files, modifying .env files, asking user questions and validating answers)

CHAPTER 2

Install RKD

RiotKit-Do is delivered as a Python package that can be installed system-wide or in a virtual environment. The virtual environment installation is similar in concept to the Gradle wrapper (gradlew)

```
# 1) via PIP
pip install rkd

# 2) Create project (will create a virtual env and commit files to GIT)
rkd :rkd:create-structure --commit
```

CHAPTER 3

Getting started in freshly created structure

The “Quick start” section ends up with a **.rkd** directory, a requirements.txt and ./rkdw

1. Call RKD using a wrapper in project directory **./rkdw**
2. Each time you install anything from **pip** in your project - add it to requirements.txt (or Pipfile), you can install additional RKD tasks from pip
3. In **.rkd/makefile.yaml** add your first tasks, pipelines and imports

CHAPTER 4

Create your first task with Beginners guide - on YAML syntax example

CHAPTER 5

Check how to use commandline to run tasks in RKD with Commandline basics

CHAPTER 6

See how to import existing tasks to your Makefile with Importing tasks page

CHAPTER 7

Keep learning

- YAML syntax is described also in *Tasks development* section
- Writing Python code in makefile.yaml requires to lookup *Tasks API*
- Learn how to import installed tasks via pip - *Importing tasks*
- You can also write tasks code in pure Python and redistribute those tasks via Python's PIP - see *Tasks development*
- With RKD you can create interactive installers - check the *Creating installer wizards with RKD* section

7.1 Beginners guide - on YAML syntax example

7.1.1 Where to place files

`.rkd` directory must always exists in your project. Inside `.rkd` directory you should place your makefile.yaml that will contain all of the required tasks.

Just like in UNIX/Linux, and just like in Python - there is an environment variable `RKD_PATH` that allows to define multiple paths to `.rkd` directories placed in other places - for example outside of your project. This gives a flexibility and possibility to build system-wide tools installable via Python's PIP.

7.1.2 Environment variables

RKD natively reads .env (called also "dot-env files") at startup. You can define default environment values in .env, or in other .env-some-name files that can be included in `env_files` section of the YAML.

Scope of environment variables

`env_files` and `environment` blocks can be defined globally, which will end in including that fact in each task, second possibility is to define those blocks per task. Having both global and per-task block merges those values together and makes per-task more important.

Example

```
version: org.riotkit.rkd/yaml/v1
environment:
    PYTHONPATH: "/project"
tasks:
    :hello:
        description: Prints variables
        environment:
            SOME_VAR: "HELLO"
        steps: |
            echo "SOME_VAR is ${SOME_VAR}, PYTHONPATH is ${PYTHONPATH}"
```

7.1.3 Arguments parsing

Arguments parsing is a strong side of RKD. Each task has it's own argument parsing, it's own generated –help command. Python's argparse library is used, so Python programmers should feel like in home.

Example

```
version: org.riotkit.rkd/yaml/v1
environment:
    PYTHONPATH: "/project"
tasks:
    :hello:
        description: Prints your name
        arguments:
            "--name":
                required: true
                #option: store_true # for booleans/flags
                #default: "Unknown" # for default values
        steps: |
            echo "Hello ${ARG_NAME}"
```

```
rkd :hello --name Peter
```

7.1.4 Defining tasks in Python code

Defining tasks in Python gives wider possibilities - to access Python's libraries, better handle errors, write less tricky code. RKD has a similar concept to hashbangs in UNIX/Linux.

There are two supported hashbangs + no hashbang:

- #!python
- #!bash
- (just none there)

What can I do in such Python code? Everything! Import, print messages, execute shell commands, everything.

Example

```
version: org.riotkit.rkd/yaml/v1
environment:
    PYTHONPATH: "/project"
```

(continues on next page)

(continued from previous page)

```
tasks:
    :hello:
        description: Prints your name
        arguments:
            "--name":
                required: true
                #option: store_true # for booleans/flags
                #default: "Unknown" # for default values
        steps: |
            #!python
            print('Hello %s' % ctx.get_arg('--name'))
```

Special variables

- *this* - instance of current TaskInterface implementation
- *ctx* - instance of ExecutionContext

Please check *Tasks API* for those classes reference.

7.1.5 YAML syntax reference

Let's at the beginning start from analyzing an example.

```
version: org.riotkit.rkd/yaml/v1

# optional: Import tasks from Python packages
# This gives a possibility to publish tasks and share across projects, teams,␣
↪organizations
imports:
    - rkt_utils.db.WaitForDatabaseTask

# optional environment section would append those variables to all tasks
# of course the tasks can overwrite those values in per-task syntax
environment:
    PYTHONPATH: "/project/src"

# optional env files loaded there would append loaded variables to all tasks
# of course the tasks can overwrite those values in per-task syntax
#env_files:
#    - .some-dotenv-file

tasks:
    :check-is-using-linux:
        description: Are you using Linux?
        # use sudo to become a other user, optional
        become: root
        steps:
            # steps can be defined as single step, or multiple steps
            # each step can be in a different language
            # each step can be a multiline string
            - "[[ $(uname -s) == \"Linux\" ]] && echo \"You are using Linux, cool\""
            - echo "step 2"
            - |
                #!python
                print('Step 3')
```

(continues on next page)

(continued from previous page)

```
    :hello:
        description: Say hello
        arguments:
            "--name":
                help: "Your name"
                required: true
                #default: "Peter"
                #option: "store_true" # for booleans
        steps: |
            echo "Hello ${ARG_NAME}"

            if [[ $(uname -s) == "Linux" ]]; then
                echo "You are a Linux user"
            fi
```

imports - Imports external tasks installed via Python' PIP. That's the way to easily share code across projects

environment - Can define default values for environment variables. Environment section can be defined for all tasks, or per task

env_files - Includes .env files, can be used also per task

tasks - List of available tasks, each task has a name, descripton, list of steps (or a single step), arguments

Running the example:

1. Create a .rkd directory
2. Create .rkd/makefile.yaml file
3. Paste/rewrite the example into the .rkd/makefile.yaml
4. Run `rkd :tasks` from the directory where the .rkd directory is placed
5. Run defined tasks `rkd :hello :check-is-using-linux`

Example projects using Makefile YAML syntax:

- Taiga docker image
- Taiga Events docker image
- K8S Workspace

7.2 Extended usage - Makefile in Python syntax

Not only tasks can be written in Python code, but Makefile too - such makefile is called `makefile.py`, and placed in `.rkd` directory.

Example:

```
import os
from rkd.core.api.syntax import TaskAliasDeclaration as Task  # RKD API (for defining␣
↪shortcuts/aliases for whole tasks lists)
from rkd.core.api.syntax import TaskDeclaration               # RKD API (for␣
↪declaring usage of given task, importing it)
from rkd.core.api.contract import ExecutionContext            # RKD API (one of␣
↪dependencies - context gives us access to commandline arguments and environment␣
↪variables)
```

(continues on next page)

(continued from previous page)

```
from rkd.pythonic import imports as PythonImports           # group of imports (not␣
↪all packages supports it, but most of them)
from rkd.core.standardlib.jinja import FileRendererTask       # single task
from rkd.core.standardlib import CallableTask                 # Basic Python callable␣
↪task for a little bit advanced usage
# from .mypackage import MyTask                             # import your task from␣
↪local package

os.putenv('LANG', 'c')


def example_method(ctx: ExecutionContext, task: CallableTask) -> bool:
    print('I AM EXECUTED')
    raise Exception('test123')
    return True


IMPORTS = [
    # We declare that we will use this task.
    # Declaration can take some additional arguments like args= or env=, to always␣
↪append environment and/or commandline switches
    # regardless of if user used it
    TaskDeclaration(FileRendererTask()),
    # remember about the "," between tasks, it's an array/list ;)
    # TaskDeclaration(MyTask())

    TaskDeclaration(CallableTask(':read-real-history', example_method, description=
↪'Example task with simple Python code'))
]

IMPORTS += PythonImports()

TASKS = [
    # declared task-aliases. A Task Alias is a shortcut eg. ":release" that will␣
↪expands to ":py:build :py:publish --username= (...)"
    # the best feature in task-aliases is that you can append and overwrite last␣
↪commandline arguments, those will be added
    # at the end of the command
    Task(':release', description='Release to PyPI (snapshot when on master, release␣
↪on tag)',
         to_execute=[
            ':py:build', ':py:publish', '--username=__token__', '--password=${PYPI_
↪TOKEN}'
         ]),

    Task(':test', [':py:unittest'], description='Run unit tests'),
    Task(':docs', [':sh', '-c', ''' set -x
        cd docs
        rm -rf build
        sphinx-build -M html "source" "build"
    '''])
]
```

- The Python syntax is very flexible
- You can create your own local packages and import them here, create own advanced structure
- Possibility to declare aliases and adjust TaskDeclarations for advanced usage (YAML syntax does not offer this)

Example projects using Makefile.py syntax:

- TunMan
- RiotKit Harbor building scripts
- RiotKit CI Utils

7.2.1 Check ADVANCED usage page for description of all environment variables, mechanisms, good practices and more

7.3 Commandline basics

RKD command-line usage is highly inspired by GNU Make and Gradle, but it has its own extended possibilities to make your scripts smaller and more readable.

- Tasks are prefixed always with ":".
- Each task can handle it's own arguments (unique in RKD)
- "@" allows to propagate arguments to next tasks (unique in RKD)

7.3.1 Tasks arguments usage in shell and in scripts

Executing multiple tasks in one command:

```
rkd :task1 :task2
```

Multiple tasks with different switches:

```
rkd :task1 --hello  :task2 --world --become=root
```

Second task will run as root user, additionally with `--world` parameter.

Tasks sharing the same switches

Both tasks will receive switch "–hello"

```
# expands to:
#  :task1 --hello
#  :task2 --hello
rkd @ --hello :task1 :task2

# handy, huh?
```

Advanced usage of shared switches

Operator "@" can set switches anytime, it can also clear or replace switches in **NEXT TASKS**.

```
# expands to:
#   :task1 --hello
#   :task2 --hello
#   :task3
#   :task4 --world
#   :task5 --world
rkd @ --hello :task1 :task2 @ :task3 @ --world :task4 :task5
```

Written as a pipeline (regular bash syntax)

It's exactly the same example as above, but written multiline. It's recommended to write multiline commands if they are longer.

```
rkd @ --hello \
    :task1 \
    :task2 \
    @
    :task3 \
    @ --world \
    :task4 \
    :task5
```

7.4 Importing tasks

Tasks can be defined as installable Python's packages that you can import in your Makefile

Please note:

- To import a group, the package you try to import need to hvve a defined **imports()** method inside of the package.
- The imported group does not need to import automatically dependend tasks (but it can, it is recommended), you need to read into the docs of specific package if it does so

7.4.1 1) Install a package

RKD defines dependencies using Python standards.

Example: Given we want to import tasks from package "rkt_armutils".

```
echo "rkt_armutils==3.0" >> requirements.txt
pip install -r requirements.txt
```

Good practices:

- Use fixed versions eg. 3.0 or even 3.0.0 and upgrade only intentionally to reduce your work. Automatic updates, especially of major versions

could be unpredictable and possibly can break something time-to-time

How do I check latest version?:

- Simply install a package eg. `pip install rkt_armutils`, then do a `pip show rkt_armutils` and write the version

to the requirements.txt, or lookup a package first at https://pypi.org/project/rkt_armutils/ (where rkt_armutils is an example package)

7.4.2 2) In YAML syntax

Example: Given we want to import task "InjectQEMUBinaryIntoContainerTask", or we want to import whole "rkt_armutils.docker" group

```
imports:
    # Import whole package, if the package defines a group import (method imports())
    - rkt_armutils.docker

    # Or import single task
    - rkt_armutils.docker.InjectQEMUBinaryIntoContainerTask
```

7.4.3 2) In Python syntax

Example: Given we want to import task "InjectQEMUBinaryIntoContainerTask", or we want to import whole "rkt_armutils.docker" group

```
from rkd.core.api.syntax import TaskDeclaration
from rkt_armutils.docker import InjectQEMUBinaryIntoContainerTask

# ... (use "+" operator to append, remove "+" if you didn't define any import yet)
IMPORTS += [TaskDeclaration(InjectQEMUBinaryIntoContainerTask)]
```

7.4.4 3) Inline syntax

Tasks could be imported also in shell, for quick check, handy scripts, or for embedding inside other applications.

```
# note: Those examples requires "rkt_utils" package from PyPI
RKD_IMPORTS="rkt_utils.docker" rkd :docker:tag
RKD_IMPORTS="rkt_utils.docker:rkt_ciutils.boatci:rkd_python" rkd :tasks

# via commandline switch "--imports"
rkd --imports "rkt_utils.docker:rkt_ciutils.boatci:rkd_python" :tasks
```

Note: The significant difference between environment variable and commandline switch is that the environment variable will be inherited into subshells of RKD, commandline argument not.

For more information about this environment variable check it's documentation page: *RKD_IMPORTS*

7.4.5 Ready to go? Check Built-in tasks that you can import in your Makefile

7.5 ADVANCED usage

7.5.1 Troubleshooting

1. Output is corrupted or there is no output from a shell command executed inside of a task

The output capturing is under testing. The Python's subprocess module is skipping "sys.stdout" and "sys.stderr" by writing directly to /dev/stdout and /dev/stderr, which makes output capturing difficult.

Run rkd in compat mode to turn off output capturing from shell commands:

```
RKD_COMPAT_SUBPROCESS=true rkd :some-task-here
```

7.5.2 Loading priority

7.5.2.1 Environment variables loading order from .env and from .rkd

Legend: Top is most important, the variables loaded on higher level are not overridden by lower level

1. Operating system environment
2. Current working directory .env file
3. .env files from directories defined in RKD_PATH

7.5.2.2 Environment variables loading order in YAML syntax

Legend: Top - is most important

1. Operating system environment
2. .env file
3. Per-task "environment" section
4. Per-task "env_file" imports
5. Global "environment" section
6. Global "env_file" imports

7.5.2.3 Order of loading of makefile files in same .rkd directory

Legend: Lower has higher priority (next is appending changes to previous)

1. *.py
2. *.yaml
3. *.yml

7.5.2.4 Paths and inheritance

RKD by default search for .rkd directory in current execution directory - *./.rkd*.

The search order is following (from lower to higher load priority):

1. RKD's internals (we provide a standard tasks like *:tasks*, *:init*, *:sh*, *:exec* and more)
2. */usr/lib/rkd*
3. User's home *~/.rkd*
4. Current directory *./.rkd*
5. *RKD_PATH*

Custom path defined via environment variable

RKD_PATH allows to define multiple paths that would be considered in priority.

```
export RKD_PATH="/some/path:/some/other/path:/home/user/riotkit/.rkd-second"
```

How the makefiles are loaded?

Each makefile is loaded in order, next makefile can override tasks of previous. That's why we at first load internals, then your tasks.

7.5.2.5 Tasks execution

Tasks are executed one-by-one as they are specified in commandline or in TaskAlias declaration (commandline arguments).

```
rkd :task-1 :task-2 :task-3
```

1. task-1
2. task-2
3. task-3

A –keep-going can be specified after given task eg. :task-2 –keep-going, to ignore a single task failure and in consequence allow to go to the next task regardless of result.

7.5.3 Tasks development

RKD has multiple approaches to define a task. The first one is simpler - in makefile in YAML or in Python. The second one is a set of tasks as a Python package.

7.5.3.1 Option 1) Simplest - in YAML syntax

Definitely the simplest way to define a task is to use YAML syntax, it is recommended for beginning users.

Example 1:

```
version: org.riotkit.rkd/yaml/v1
imports:
  - rkd.standardlib.jinja.RenderDirectoryTask

tasks:
  # see this task in "rkd :tasks"
  # run with "rkd :examples:bash-test"
  :examples:bash-test:
      description: Execute an example command in bash - show only python related tasks
      steps: |
             echo "RKD_DEPTH: ${RKD_DEPTH} # >= 2 means we are running rkd-in-rkd"
             echo "RKD_PATH: ${RKD_PATH}"
             rkd --silent :tasks | grep ":py"

  # try "rkd :examples:arguments-test --text=Hello --test-boolean"
  :examples:arguments-test:
      description: Show example usage of arguments in Bash
      arguments:
          "--text":
              help: "Adds text message"
              required: True
          "--test-boolean":
              help: "Example of a boolean flag"
              action: store_true # or store_false
```

(continues on next page)

(continued from previous page)

```
    steps:
      - |
        #!bash
        echo " ==> In Bash"
        echo " Text: ${ARG_TEXT}"
        echo " Boolean test: ${ARG_TEST_BOOLEAN}"
      - |
        #!python
        print(' ==> In Python')
        print(' Text: %s ' % ctx.args['text'])
        print(' Text: %s ' % str(ctx.args['test_boolean']))
        return True

# run with "rkd :examples:list-standardlib-modules"
:examples:list-standardlib-modules:
    description: List all modules in the standardlib
    steps:
      - |
        #!python
        ctx: ExecutionContext
        this: TaskInterface

        import os

        print('Hello world')
        print(os)
        print(ctx)
        print(this)

        return True

:examples:with-other-workdir:
    description: "This task runs in /tmp"
    workdir: "/tmp"
    steps: |
        echo "I run in"
        pwd
```

Example 2:

```
version: org.riotkit.rkd/yaml/v1

environment:
    GLOBALLY_DEFINED: "16 May 1966, seamen across the UK walked out on a nationwide␣
↪strike for the first time in half a century. Holding solid for seven weeks, they␣
↪won a reduction in working hours from 56 to 48 per week "

env_files:
    - env/global.env

tasks:
    :hello:
        description: |
            #1 line: 11 June 1888 Bartolomeo Vanzetti, Italian-American anarchist who␣
↪was framed & executed alongside Nicola Sacco, was born.
            #2 line: This is his short autobiography:
            #3 line: https://libcom.org/library/story-proletarian-life
```

(continues on next page)

(continued from previous page)

```
        environment:
            INLINE_PER_TASK: "17 May 1972 10,000 schoolchildren in the UK walked out␣
↪on strike in protest against corporal punishment. Within two years, London state␣
↪schools banned corporal punishment. The rest of the country followed in 1987."
        env_files: ['env/per-task.env']
        steps: |
            echo " >> ENVIRONMENT VARIABLES DEMO"
            echo "Inline defined in this task: ${INLINE_PER_TASK}\n\n"
            echo "Inline defined globally: ${GLOBALLY_DEFINED}\n\n"
            echo "Included globally - global.env: ${TEXT_FROM_GLOBAL_ENV}\n\n"
            echo "Included in task - per-task.env: ${TEXT_PER_TASK_FROM_FILE}\n\n"
```

Explanation of examples:

1. "arguments" is an optional dict of arguments, key is the argument name, subkeys are passed directly to argparse
2. "steps" is a mandatory list or text with step definition in Bash or Python language
3. "description" is an optional text field that puts a description visible in ":tasks" task
4. "workdir" allows to optionally specify a working directory for a task
5. "environment" is a dict of environment variables that can be defined
6. "env_files" is a list of paths to .env files that should be included
7. "imports" imports a Python package that contains tasks to be used in the makefile and in shell usage

7.5.3.2 Option 2) For Python developers - task as a class

This way allows to create tasks in a structure of a Python module. Such task can be packaged, then published to eg. PyPI (or other private repository) and used in multiple projects.

Each task should implement methods of **rkd.core.api.contract.TaskInterface** interface, that's the basic rule.

Following example task could be imported with path **rkd.standardlib.ShellCommandTask**, in your own task you would have a different package name instead of **rkd.standardlib**.

Example task from RKD standardlib:

Explanation of example:

1. The docstring in Python class is what will be shown in **:tasks as description**. You can also define your description by implementing **def get_description() -> str**
2. Name and group name defines a full name eg. :your-project:build
3. **def configure_argparse()** allows to inject arguments, and –help description for a task - it's a standard Python's argparse object to use
4. **def execute()** provides a context of execution, please read *Tasks API* chapter about it. In short words you can get commandline arguments, environment variables there.
5. **self.io()** is providing input-output interaction, please use it instead of print, please read *Tasks API* chapter about it.

7.5.3.3 Option 3) Quick and elastic way in Python code of Makefile.py

Multiple Makefile files can be used at one time, you don't have to choose between YAML and Python. This opens a possibility to define more advanced tasks in pure Python, while you have most of the tasks in YAML. It's elastic - use YAML, or Python or both.

Let's define then a task in Python in a simplest method.

Makefile.py

```
import os
from rkd.core.api.syntax import TaskDeclaration
from rkd.core.api.contract import ExecutionContext
from rkd.standardlib import CallableTask

def union_method(context: ExecutionContext) -> bool:
    os.system('xdg-open https://iwa-ait.org')
    return True

IMPORTS = [
    # just declare a task with a name + code as function! Yay, simple!
    TaskDeclaration(CallableTask(':create-union', union_method), workdir='/var')
]

TASKS = []
```

Extended usage - Makefile in Python syntax.

7.5.3.4 Please check Tasks API for interfaces description

7.5.4 Global environment variables

Global switches designed to customize RKD per project. Put environment variables into your **.env** file, so you will no have to prepend them in the commandline every time.

Read also about *Environment variables loading order from .env and from .rkd*

7.5.4.1 RKD_WHITELIST_GROUPS

Allows to show only selected groups in the ":tasks" list. All tasks from hidden groups are still callable.

Examples:

```
RKD_WHITELIST_GROUPS=:rkd, rkd :tasks
RKD_WHITELIST_GROUPS=:rkd rkd :tasks
```

7.5.4.2 RKD_ALIAS_GROUPS

Alias group names, so it can be shorter, or even group names could be not typed at all.

Notice: :tasks will rename a group with a first defined alias for this group

Examples:

```
RKD_ALIAS_GROUPS=":rkd->:r" rkd :tasks :r:create-structure
RKD_ALIAS_GROUPS=":rkd->" rkd :tasks :create-structure
```

7.5.4.3 RKD_UI

Allows to toggle (true/false) the UI - messages like "Executing task X" or "Task finished", leaving only tasks stdout, stderr and logs.

7.5.4.4 RKD_AUDIT_SESSION_LOG

Logs output of each executed task, when set to "true".

Example structure of logs:

```
# Note: This example requires "rkd-harbor" package to be installed from PyPI
RKD_AUDIT_SESSION_LOG=true harbor :service:list   # RiotKit Harbor is another project␣
↪based on RKD

# ls .rkd/logs/2020-06-11/11\:06\:02.068556/
task-1-init.log  task-2-harbor_service_list.log
```

7.5.4.5 RKD_BIN

Defines a command that invokes RKD eg. `rkd`. When a custom distribution is present, then this value can different. For example project RiotKit Harbor has it's own command `harbor`, which is based on RKD, so the RKD_BIN=harbor would be defined in such project.

RKD_BIN is automatically generated, when executing task in a separate process, but it can be also set globally.

7.5.4.6 RKD_SYS_LOG_LEVEL

Use for debugging. The variable is read in very early stage of RKD initialization, before `:init` task, and before context preparation.

```
RKD_SYS_LOG_LEVEL=debug rkd :tasks
```

7.5.4.7 RKD_IMPORTS

Allows to import a task, or group of tasks (module) inline, without need to create a Makefile. Useful in daily tasks to create handy shortcuts, also very useful for testing tasks and embedding them inside other applications.

"**:**" character is a separator for multiple imports.

```
# note: Those examples requires "rkt_utils" package from PyPI
RKD_IMPORTS="rkt_utils.docker" rkd :docker:tag
RKD_IMPORTS="rkt_utils.docker:rkt_ciutils.boatci:rkd_python" rkd :tasks
```

7.5.5 Custom distribution

RiotKit Do can be used as a transparent framework for writing tasks for various usage, especially for specialized usage. To simplify usage for end-user RKD allows to create a custom distribution.

Custom distribution allows to:

- Define custom 'binary' name eg. "harbor" instead of "rkd"

- Hide unnecessary tasks in custom 'binary' (filter by groups - whitelist)
- Make shortcuts to tasks: Skip writing group name, make a group name to be appended by default

7.5.5.1 Example

```
import os
from rkd import main as rkd_main

def env_or_default(env_name: str, default: str):
    return os.environ[env_name] if env_name in os.environ else default

def main():
    os.environ['RKD_WHITELIST_GROUPS'] = env_or_default('RKD_WHITELIST_GROUPS', ':env,
↪:harbor,')
    os.environ['RKD_ALIAS_GROUPS'] = env_or_default('RKD_ALIAS_GROUPS', '->:harbor')
    os.environ['RKD_UI'] = env_or_default('RKD_UI', 'false')
    rkd_main()


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

```
$ harbor :tasks
[global]
:sh                                                 # Executes shell scripts
:exec                                               # Spawns a shell process
:init                                               # :init task is executing ALWAYS.␣
↪That's a technical, core task.
:tasks                                              # Lists all enabled tasks
:version                                            # Shows version of RKD and of all␣
↪loaded tasks

[harbor]
:compose:ps                                         # List all containers
:start                                              # Create and start containers
:stop                                               # Stop running containers
:remove                                             # Forcibly stop running containers␣
↪and remove (keeps volumes)
:service:list                                       # Lists all defined containers in␣
↪YAML files (can be limited by --profile selector)
:service:up                                         # Starts a single service
:service:down                                       # Brings down the service without␣
↪deleting the container
:service:rm                                         # Stops and removes a container and␣
↪it's images
:pull                                               # Pull images specified in␣
↪containers definitions
:restart                                            # Restart running containers
:config:list                                        # Gets environment variable value
:config:enable                                      # Enable a configuration file - YAML
:config:disable                                     # Disable a configuration file -␣
↪YAML
:prod:gateway:reload                                # Reload gateway, regenerate␣
↪missing SSL certificates
:prod:gateway:ssl:status                            # Show status of SSL certificates
:prod:gateway:ssl:regenerate                        # Regenerate all certificates with␣
↪force
```

(continues on next page)

(continued from previous page)

```
:prod:maintenance:on                                    # Turn on the maintenance mode
:prod:maintenance:off                                   # Turn on the maintenance mode
:git:apps:update                                        # Fetch a git repository from the␣
↪remote
:git:apps:update-all                                    # List GIT repositories
:git:apps:set-permissions                               # Make sure that the application␣
↪would be able to write to allowed directories (eg. upload directories)
:git:apps:list                                          # List GIT repositories

[env]
:env:get                                                # Gets environment variable value
:env:set                                                # Sets environment variable in the .
↪env file


Use --help to see task environment variables and switches, eg. rkd :sh --help, rkd --
↪help
```

Notices for above example:

- No need to type eg. :harbor:config:list - just :config:list (RKD_ALIAS_GROUPS used)
- No "rkd" group is displayed (RKD_WHITELIST_GROUPS used)
- There is no information about task name (RKD_UI used)

7.5.5.2 Read more in Global environment variables

7.5.5.3 Customizing RKD resource files

Files like banner, internal Makefiles can be overridden in user's home directory, or in operating system-wide directory.

Here is the priority list, first matching result stops the search:

7.5.6 Tasks API

7.5.6.1 Each task must implement a TaskInterface

7.5.6.2 To include a task, wrap it in a declaration

7.5.6.3 To create an alias for task or multiple tasks

7.5.6.4 Execution context provides parsed shell arguments and environment variables

7.5.6.5 Interaction with input and output

7.5.6.6 Storing temporary files

7.5.6.7 Parsing RKD syntax

7.5.6.8 Testing

7.5.7 Working with YAML files

Makefile.yaml has checked syntax before it is parsed by RKD. A **jsonschema** library was used to validate YAML files against a JSON formatted schema file.

This gives the early validation of typing inside of YAML files, and a clear message to the user about place where the typo is.

7.5.7.1 YAML parsing API

Schema validation is a part of YAML parsing, the preferred way of working with YAML files is to not only parse the schema but also validate. In result of this there is a class that wraps **yaml** library - **rkd.yaml_parser.YamlFileLoader**, use it instead of plain **yaml** library.

Notice: The YAML and schema files are automatically searched in .rkd, .rkd/schema directories, including RKD_PATH

Example usage:

```
from rkd.yaml_parser import YamlFileLoader

parsed = YamlFileLoader([]).load_from_file('deployment.yml', 'org.riotkit.harbor/
↪deployment/v1')
```

7.5.7.2 FAQ

1. *FileNotFoundError: Schema "my-schema-name.json" cannot be found, looked in: ['.../riotkit-harbor', '/.../riotkit-harbor/schema', '/.../riotkit-harbor/.rkd/schema', '/home/.../.rkd/schema', '/usr/lib/rkd/schema', '/usr/lib/python3.8/site-packages/rkd/internal/schema']*

The schema file cannot be found, the name is invalid or file missing. The schema should be placed somewhere in the .rkd/schema directory - in global, in home directory or in project.

2. *rkd.exception.YAMLFileValidationError: YAML schema validation failed at path "tasks" with error: [] is not of type 'object'*

It means you created a list (starts with "-") instead of dictionary at "tasks" path.

Example what went wrong:

```
tasks:
    - description: first
    - description: second
```

Example how it should be as an 'object':

```
tasks:
    first:
        description: first

    second:
        description: second
```

7.5.7.3 API

7.5.8 Creating installer wizards with RKD

Wizard is a component designed to create comfortable installers, where user has to answer a few questions to get the task done.

7.5.8.1 Concept

- User answers questions invoked by `ask()` method calls
- At the end the `finish()` is called, which acts as a commit, saves answers into `.rkd/tmp-wizard.json` by default and into the `.env` file (depends on if to_env=true was specified)
- Next RKD task executed can read `.rkd/tmp-wizard.json` looking for answers, the answers placed in .env are already loaded automatically as part of standard mechanism of environment variables support

7.5.8.2 Example Wizard

```
from rkd.core.api.inputoutput import Wizard

# self is the TaskInterface instance, in Makefile.yaml it would be "this", in Python
↪code it is "self"
Wizard(self)\
    .ask('Service name', attribute='service_name', regexp='([A-Za-z0-9_]+)', default=
↪'redis')\
    .finish()
```

```
Service name [([A-Za-z0-9_]+)] [default: redis]:
    -> redis
```

Example of application that is using Wizard to ask interactive questions

7.5.8.3 Using Wizard results internally

Wizard is designed to keep the data on the disk, so you can access it in any other task executed, but this is not mandatory. You can skip committing changes to disk by not using `finish()` which **is flushing data to json and to .env files.**

Use `wizard.answers` to see all answers that would be put into json file, and `wizard.to_env` to browse all environment variables that would be set in .env if `finish()` would be used.

7.5.8.4 Example of loading stored values by other task

Wizard stores values into file and into .env file, so it can read it from file after it was stored there. This allows you to separate Wizard questions into one RKD task, and the rest of logic/steps into other RKD tasks.

```
from rkd.core.api.inputoutput import Wizard

# ... assuming that previously the Wizard was completed by user and the finish()␣
↪method was called ...

wizard = Wizard(self)
wizard.load_previously_stored_values()

print(wizard.answers, wizard.to_env)
```

7.5.8.5 API

7.5.9 Good practices

7.5.9.1 Do not use os.getenv()

Note: Only in Python code

The ExecutionContext is providing processed environment variables. Variables could be overridden on some levels eg. in makefile.py - `rkd.core.api.syntax.TaskAliasDeclaration` can take a dict of environment variables to force override.

Use `context.get_env()` instead.

7.5.9.2 Define your environment variables

Note: Only in Python code

By using `context.get_env()` you are enforced to implement a `TaskInterface.get_declared_envs()` returning a list of all environment variables used in your task code.

All defined environment variables will land in –help, which is considered as a task self-documentation.

7.5.9.3 Use sh(), exec(), rkd() and silent_sh()

Using raw `subprocess` will make your commands output invisible in logs, as the subprocess is writting directly to stdout/stderr skipping sys.stdout and sys.stderr. The methods provided by RKD are buffering the output and making it possible to save to both file and to console.

7.5.9.4 Do not print if you do not must, use io()

`rkd.core.api.inputoutput.IO` provides a standardized way of printing messages. The class itself distinct importance of messages, writing them to proper stdout/stderr and to log files.

`print` is also captured by IO, but should be used only eventually.

7.5.10 Process isolation and permissions changing with sudo

Alternatively called "forking" is a feature of RKD similar to Gradle's JVM forking - the task can be run in a separate Python's process. This gives a possibility to run specific task as a specific user (eg. upgrade permissions to ROOT or downgrade to regular user)

7.5.10.1 Mechanism

RKD uses serialization to transfer data between processes - a standard `pickle` library is used. Pickle has limitations on what can be serialized - any inner-methods and lambdas cannot be returned by task.

To test if your task is compatible with running as a separate process simply add `--become=USER-NAME` to the commandline of your task. If it will fail due to serialization issue, then you will be notified with a nice stacktrace.

Technically the mechanism works on the task executor level, it means that process isolation is independent of the programming language as whole task's execute() is ran in a separate process, even if task is declared in YAML and has Bash steps.

7.5.10.2 Permissions changing with sudo

YAML syntax allows to define additional attribute `become`, that if defined then makes whole task to execute inside a separate Python process ran with sudo.

Additionally the RKD commandline supports a per-task parameter `--become`

7.5.10.3 Future usage

The mechanism is universal, it can be possibly used to sandbox, or even to execute tasks remotely. Currently we do not support such features but we do not say its impossible in the future.

7.5.11 Docker entrypoints under control

RKD has enough small footprint so that it can be used as an entrypoint in docker containers. There are a few features that are making RKD very attractive to use in this role.

7.5.11.1 Environment variables

Defined commandline `--my-switch` can have optionally overridden value with environment variable. In docker it can help easily adjusting default values.

Task needs to create an explicit declaration of environment variable:

```
def get_declared_envs(self) -> Dict[str, ArgumentEnv]:
    return {
        'MY_SWITCH': ArgumentEnv(name='MY_SWITCH', switch='--switch-name', default='
↪'),
    }
```

```
def execute(self, ctx: ExecutionContext) -> bool:
    # this one will look for a switch value, if switch has default value, then it␣
↪will look for an environment variable
    ctx.get_arg_or_env('--my-switch')
```

7.5.11.2 Arguments propagation

When setting `ENTRYPOINT ["rkd", ":entrypoint"]` everything that will be passed as docker's CMD will be passed to rkd, so additional tasks and arguments can be appended.

7.5.11.3 Tasks customization

It is a good practice to split your entrypoint into multiple tasks executed one-by-one. This gives you a possibility to create new `makefile.yaml/py` in any place and modify `RKD_PATH` environment variable to add additional tasks or replace existing. The RKD_PATH has always higher priority than current `.rkd` directory.

Possible options:

- Create a bind-mount volume with additional `.rkd/makefile.yaml`, add `.rkd/makefile.yaml` into container and set RKD_PATH to point to `.rkd` directory
- Create new docker image having original in `FROM`, add `.rkd/makefile.yaml` into container and set RKD_PATH to point to `.rkd` directory

7.5.11.4 Massive files rendering with JINJA2

`:j2:directory-to-directory` is a specially designed task to render JINJA2 templates recursively preserving a directory structure. You can create for example `templates/etc/nginx/nginx.conf.j2` and render `./templates/etc` into `/etc` with all files being copied on the fly.

All jinja2 templates will have access to environment variables - with templating syntax you can define very advanced configuration files

7.5.11.5 Privileges dropping

Often in entrypoint there are cache/uploads permissions corrected, so the `root` user is used. To migrate the application, to run the webserver the privileges could be dropped.

Solutions:

- In YAML syntax each task have a possible field to use: `become: user-name-here`
- In Python class TaskInterface has method `get_become_as()` that should return empty string or a username to use sudo with
- In commandline there is a switch `--become=user-name-here` that can be used with most of the tasks

7.5.12 Testing with PyTest

`rkd.core.api.testing` provides methods for running tasks with output capturing, a well as mocking RKD classes for unit testing of your task methods. To use our API just extend one of base classes.

7.5.12.1 Example: Running a task on a fully featured RKD executor

```
#!/usr/bin/env python3

import os
from rkd.core.api.testing import FunctionalTestingCase

SCRIPT_DIR_PATH = os.path.dirname(os.path.realpath(__file__))


class TestFunctional(FunctionalTestingCase):
    """
    Functional tests case of the whole application.
    Runs application like from the shell, captures output and performs assertions on
↪the results.
    """

    def test_tasks_listing(self):
        """ :tasks """

        full_output, exit_code = self.run_and_capture_output([':tasks'])

        self.assertIn(' >> Executing :tasks', full_output)
        self.assertIn('[global]', full_output)
        self.assertIn(':version', full_output)
        self.assertIn('succeed.', full_output)
        self.assertEqual(0, exit_code)
```

7.5.12.2 Example: Mocking RKD-specific dependencies in TaskInterface

```
from rkd.core.api.inputoutput import BufferedSystemIO
from rkd.core.api.testing import FunctionalTestingCase

# ...

class SomeTestCase(FunctionalTestingCase):

    # ...

    def test_something_important(self):
        task = LineInFileTask()  # put your task class there
        io = BufferedSystemIO()

        BasicTestingCase.satisfy_task_dependencies(task, io=io)

        self.assertEqual('something', task.some_method())
```

7.5.12.3 Documentation

7.6 Built-in tasks

7.6.1 Shell

Provides tasks for shell commands execution - mostly used in YAML syntax and in Python modules.

7.6.1.1 :sh

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib.shell	rkd.core.standardlib.shell.ShellCommandTask	pip install rkd==SELECT VERSION	

Executes a Bash script. Can be multi-line.

Notice: phrase %RKD% is replaced with an rkd binary name

Example of plain usage:

```
rkd :sh -c "ps aux"
rkd :sh --background -c "some-heavy-task"
```

Example of task alias usage:

```
from rkd.core.api.syntax import TaskAliasDeclaration as Task

#
# Example of Makefile-like syntax
#

IMPORTS = []

TASKS = [
    Task(':find-images', [
        ':sh', '-c', 'find ../../ -name \'*.png\''
    ]),

    Task(':build', [':sh', '-c', ''' set -x;
        cd ../../../

        chmod +x setup.py
        ./setup.py build

        ls -la
    ''']),

    # https://github.com/riotkit-org/riotkit-do/issues/43
    Task(':hello', [':sh', '-c', 'echo "Hello world"']),
    Task(':alias-in-alias-test', [':hello'])
]
```

7.6.1.2 :exec

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib.shell	rkd.core.standardlib.shell.ExecCommand	pip install rkd== SELECT VERSION	

Works identically as **:sh**, but for spawns a single process. Does not allow a multi-line script syntax.

7.6.1.3 Class to import: BaseShellCommandWithArgumentParsingTask

Creates a command that executes bash script and provides argument parsing using Python's argparse. Parsed arguments are registered as ARG_{{argument_name}} eg. –activity-type would be exported as ARG_ACTIVITY_TYPE.

```
IMPORTS += [
    BaseShellCommandWithArgumentParsingTask(
        name=":protest",
        group=":activism",
        description="Take action!",
        arguments_definition=lambda argparse: (
            argparse.add_argument('--activity-type', '-t', help='Select an activity␣
↪type')
        ),
        command='''
            echo "Let's act! Let's ${ARG_ACTIVITY_TYPE}!"
        '''
    )
]
```

7.6.2 Technical/Core

7.6.2.1 :init

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib	rkd.core.standardlib.InitTask	pip install rkd== SELECT VERSION	

This task runs ALWAYS. :init implements a possibility to inherit global settings to other tasks

7.6.2.2 :tasks

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib	rkd.core.standardlib.TasksListTask	pip install rkd== SELECT VERSION	

Lists all tasks that are loaded by all chained makefile.py configurations.

Environment variables:

- RKD_WHITELIST_GROUPS: (Optional) Comma separated list of groups to only show on the list
- RKD_ALIAS_GROUPS: (Optional) Comma separated list of groups aliases eg. ":international-workers-association->:iwa,:anarchist-federation->:fa"

7.6.2.3 :version

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib	rkd.core.standardlib.VersionTask	pip install rkd== SELECT VERSION	

Shows version of RKD and lists versions of all loaded tasks, even those that are provided not by RiotKit. The version strings are taken from Python modules as RKD strongly rely on Python Packaging.

7.6.2.4 CallableTask

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib	rkd.core.standardlib.CallableTask	pip install rkd== SELECT VERSION	

This is actually not a task to use directly, it is a template of a task to implement yourself. It's kind of a shortcut to create a task by defining a simple method as a callback.

```
import os
from rkd.core.api.syntax import TaskDeclaration
from rkd.api.contract import ExecutionContext
from rkd.core.standardlib import CallableTask


def union_method(context: ExecutionContext) -> bool:
    os.system('xdg-open https://iwa-ait.org')
    return True


IMPORTS = [
    TaskDeclaration(CallableTask(':create-union', union_method))
]

TASKS = []
```

7.6.2.5 :rkd:create-structure

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib	rkd.core.standardlib.CreateStructureTask	pip install rkd== SELECT VERSION	

Creates a template structure used by RKD in current directory.

API for developers:

This task is extensible by class inheritance, you can override methods to implement your own task with changed behavior. It was designed to allow to create customized installers for tools based on RKD (custom RKD distributions), the example is RiotKit Harbor.

Look for "interface methods" in class code, those methods are guaranteed to not change from minor version to minor version.

7.6.2.6 :file:line-in-file

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib	rkd.core.standardlib.LineInFileTask	pip install rkd==SELECT VERSION	

Similar to the Ansible's lineinfile, replaces/creates/deletes a line in file.

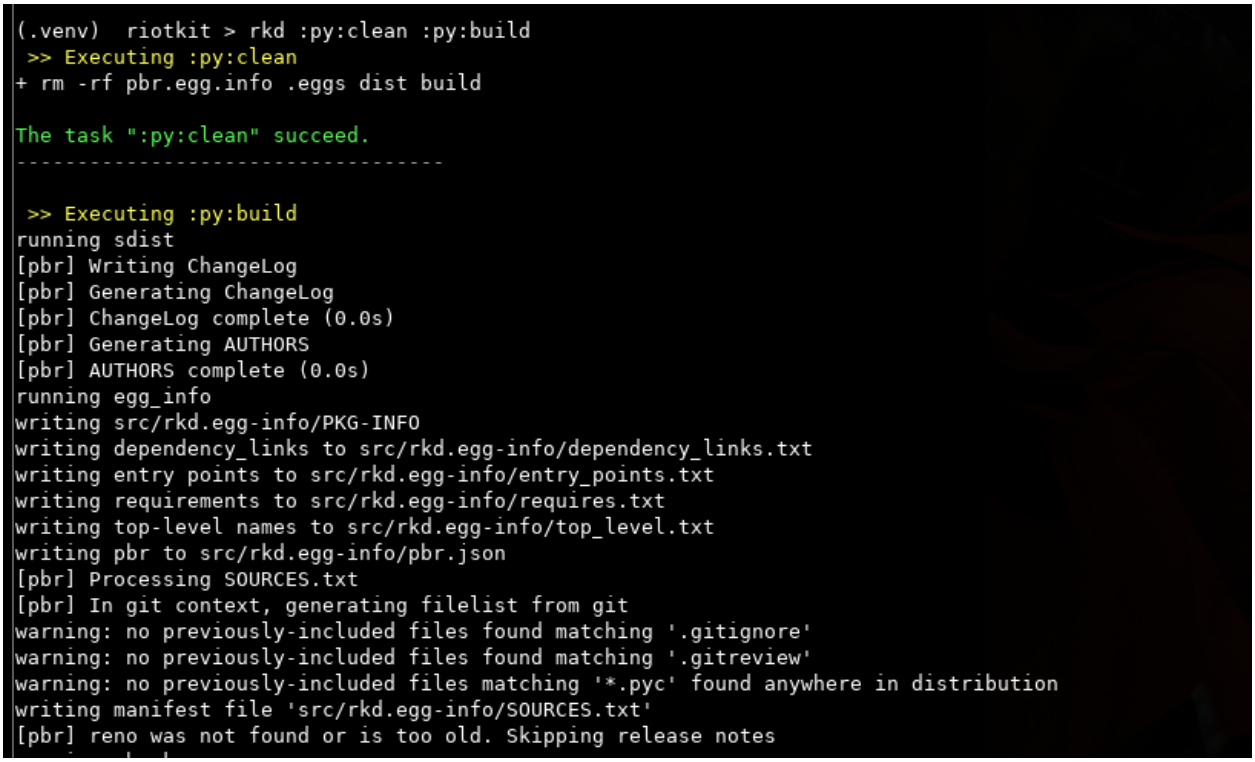
Example usage:

```
echo "Number: 10" > test.txt

rkd -rl debug :file:line-in-file test.txt --regexp="Number: ([0-9]+)?(.*)" --insert=
↪'Number: $match[0] / new: 10'
cat test.txt

rkd -rl debug :file:line-in-file test.txt --regexp="Number: ([0-9]+)?(.*)" --insert=
↪'Number: $match[0] / new: 6'
cat test.txt

rkd -rl debug :file:line-in-file test.txt --regexp="Number: ([0-9]+)?(.*)" --insert=
↪'Number: 50'
cat test.txt

rkd -rl debug :file:line-in-file test.txt --regexp="Number: ([0-9]+)?(.*)" --insert=
↪'Number: $match[0] / new: 90'
cat test.txt
```

7.6.3 Python

This package was extracted from standardlib to rkd_python, but is maintained together with RKD as part of RKD core.

Set of Python-related tasks for building, testing and publishing Python packages.



7.6.3.1 :py:publish

Package to import	Single task to import	PIP package to install	Stable version
rkd_python	rkd_python.PublishTask	pip install rkd_python== SELECT VERSION	

Publish a package to the PyPI.

Example of usage:

```
rkd :py:publish --username=__token__ --password=.... --skip-existing --test
```

7.6.3.2 :py:build

Package to import	Single task to import	PIP package to install	Stable version
rkd_python	rkd_python.BuildTask	pip install rkd_python== SELECT VERSION	

Runs a build through setuptools.

7.6.3.3 :py:install

Package to import	Single task to import	PIP package to install	Stable version
rkd_python	rkd_python.InstallTask	pip install rkd_python== SELECT VERSION	

Installs the project as Python package using setuptools. Calls ./setup.py install.

7.6.3.4 :py:clean

Package to import	Single task to import	PIP package to install	Stable version
rkd_python	rkd_python.CleanTask	pip install rkd_python== SELECT VERSION	

Removes all files related to building the application.

7.6.3.5 :py:unittest

Package to import	Single task to import	PIP package to install	Stable version
rkd_python	rkd_python.UnitTestTask	pip install rkd_python== SELECT VERSION	

Runs Python's built'in unittest module to execute unit tests.

Examples:

```
rkd :py:unittest
rkd :py:unittest -p some_test
rkd :py:unittest --tests-dir=../test
```

7.6.4 ENV

Manipulates the environment variables stored in a .env file

RKD is always loading an .env file on startup, those tasks in this package allows to manage variables stored in .env file in the scope of a project.

7.6.4.1 :env:get

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib.env	rkd.core.standardlib.env.GetTask	pip install rkd== SELECT VERSION	

Example of usage:

```
rkd :env:get --name COMPOSE_PROJECT_NAME
```

7.6.4.2 :env:set

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib.env	rkd.core.standardlib.env.GetTask	pip install rkd==SELECT VERSION	

Example of usage:

```
rkd :env:set --name COMPOSE_PROJECT_NAME --value hello
rkd :env:set --name COMPOSE_PROJECT_NAME --ask
rkd :env:set --name COMPOSE_PROJECT_NAME --ask --ask-text="Please enter your name:"
```

7.6.5 JINJA

Renders JINJA2 files, and whole directories of files. Allows to render by pattern.

All includes and extends are by default looking in current working directory path.

7.6.5.1 :j2:render

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib.jinja	rkd.core.standardlib.jinja.RenderSETask	pip install rkd==SELECT VERSION	

Renders a single file from JINJA2.

Example of usage:

```
rkd :j2:render -s SOURCE-FILE.yaml.j2 -o OUTPUT-FILE.yaml
```

7.6.5.2 :j2:directory-to-directory

Package to import	Single task to import	PIP package to install	Stable version
rkd.core.standardlib.jinja	rkd.core.standardlib.jinja.RenderSETask	pip install rkd==SELECT VERSION	

Renders all files recursively in given directory to other directory. Can remove source files after rendering them to the output files.

Note: Pattern is a regexp pattern that matches whole path, not only file name

Note: Exclude pattern is matching on SOURCE files, not on target files

Example usage:

```
rkd :j2:directory-to-directory \
    --source="/some/path/templates" \
    --target="/some/path/rendered" \
    --delete-source-files \
    --pattern="(.*).j2"
```