doit Documentation

Release 0.30.3

Jan Vlčinský

Oct 26, 2017

Table of Contents

CHAPTER 1

tasks => {doit + shell + python} => done

Declare your task(s) in *dodo.py*:

```
def task_compress():
    """Compress input file(s)"""
    return {
        "actions": [
            "7z a %(targets)s %(dependencies)s",
        ],
        "file_dep": ["lauer.dat", "hardy.dat", "elephant.txt"],
        "targets": ["bigarchive.7z"],
        "clean": True
    }
```

Install *doit* and go:

```
$ pip install doit

$ doit list  # see list of tasks with one-line helpstring
compress    Compress input file(s)

$ doit  # run all default tasks
. compress

$ doit  # skip doing tasks with results being already uptodate
-- compress

$ doit clean  # clean created files
compress - removing file 'bigarchive.7z'
```

Use Cases

Here are some use cases, where *doit* can help you with automation of your tasks.

Todo

later add links to related tutorials and how to. But be careful not to distract too much.

Simplify cumbersome command line calls

Do you have to repeatedly call complex command like this?:

```
$ aws s3 sync _built/html s3://buck/et --exclude "*" --include "*.html"
```

Wrap it into *dodo.py* file:

```
def task_publish():
  """Publish to AWS S3"""
  return {
    "actions": [
        'aws s3 sync _built/html s3://buck/et --exclude "*" --include "*.html"'
    ]
  }
```

and next time just:

```
$ doit publish
```

It is easy to include multiple actions into one task or use multiple tasks.

Automate typical project related actions

Do you have to lint your code, run test suite, evaluate coverage, generate documentation incl. spelling?

Create the *dodo.py*, which defines tasks you have to do and next time:

```
$ doit list
coverage          show coverage for all modules including tests
coverage_module   show coverage for individual modules
coverage_src      show coverage for all modules (exclude tests)
package           create/upload package to pypi
pyflakes
pypi              create/upload package to pypi
spell             spell checker for doc files
sphinx            build sphinx docs
tutorial_check    check tutorial sample are at least runnable without error
ut                run unit-tests
website           dodo file create website html files
website_update    update website on SITE_PATH
```

and then decide which task to run:

```
$ doit spell
```

Share unified way of doing things

Do you expect your colleagues perform the same steps before committing changes to repository? What to do with the complains the steps are too complex?

Provide them with the *dodo.py* file doing the things. What goes easy, is more likely to be used.

dodo.py will become easy to use prescription of best practices.

Optimize processing time by skipping tasks already done

You dump your database and convert the data to CSV. It takes minutes, but often the input is the same as before. Why to do things already done and wait?

Wrap the conversion into *doit* task and *doit* will automatically detect, the input and output are already in sync and complete in fraction of a second, when possible.

Manage complex set of depending tasks

The system you code shall do many small actions, which are interdependent.

Split it into small tasks, define (file) dependencies and let *doit* do the planning of what shall be processed first and what next.

Your solution will be clean and modular.

Speed up by parallel task execution

You already have bunch of tasks defined, results are correct, it only takes so much time. But wait, you have multi-core machine!

Just ask for parallel processing:

```
$ doit -n 4
```

and *doit* will take care of planning and make all your CPU cores hot.

No need to rewrite your processing from scratch, properly declared tasks is all what you have to provide.

Extend your project by doit features

Your own python project would need features of *doit*, but you cannot ask your users to call *doit* on command line?

Simply integrate *doit* functionality into your own command line tool and nobody will notice where it comes from.

Create cross-platform tool for processing your stuff

Do you have team members working on MS Windows and others on Linux?

Scripts are great, but all those small shell differences prevent single reusable solution.

With *dodo.py* and python you are more likely to write the processing in cross-platform way. Use *pathlib.Path* and *shutils* magic to create directories, move files around, copy them, etc.

Quick Start

Installation

Latest version of *doit* requires python 3.4 and newer.

Install via pip (best into activated virtualenv):

```
$ pip install doit
```

and test version:

```
$ doit --version
```

Note: For Python 2.7 use:

```
$ pip install doit==0.29.0
```

Advanced parts of this doc may break in Python 2.7.

Deploy power of your shell commands

To make our example easy to run on Windows as well as on Linux, we will use command *echo* redirecting some text into a file:

```
$ echo welcome > file.txt
$ echo dear user >> file.txt
$ echo enjoy powers of doit >> file.txt
```

If you used the > (overwrite) and >> (append) properly, the *file.txt* shall now have lines:

```
welcome
dear user
enjoy powers of doit
```

Let's do something similar by *doit*. Create file *dodo.py*:

```
def task_echo_to_cave():
    """Call holla to cave.txt"""
    return {
        "actions": [
             "echo pssst > cave.txt",
             "echo holla >> cave.txt",
             "echo olla >> cave.txt",
             "echo lla >> cave.txt",
             "echo la >> cave.txt",
             "echo a >> cave.txt",
        ]
    }
```

The function *task_echo_to_cave* declares the task "echo_to_cave" by means of returning dictionary with task metadata. The keyword *"actions"* here simply lists shell commands to call.

First, list all tasks defined in *dodo.py* file by:

```
$ doit list
echo_to_cave   Call holla to cave.txt
```

File *cave.txt* was not created yet as the task was only listed.

Let's execute it by:

```
$ doit run echo_to_cave
. echo_to_cave
```

The command reported the task run and executed it.

Check content of the file *cave.txt*:

```
pssst
holla
olla
lla
la
a
```

Delete the file *cave.txt* manually and execute the command again but do not add any *run* or task name:

```
$ doit
. echo_to_cave
```

By default, *doit* runs all default tasks.

The file *cave.txt* shall be again present and shall have the same content as before.

Parametrize name of target file

Hard coded file names are not always practical. Let's add *"targets"* keyword into task declaration and provide it with list of target files names, in this case just one.

Also modify the string to be echoed (adding "parametrized") to see our fingerprint in the new result.

```
def task_echo_to_cave():
    """Call holla to PARAMETRIZED target file"""
    return {
        "actions": [
             "echo parametrized pssst > %(targets)s",
             "echo parametrized holla >> %(targets)s",
             "echo parametrized olla >> %(targets)s",
             "echo parametrized lla >> %(targets)s",
             "echo parametrized la >> %(targets)s",
             "echo parametrized a >> %(targets)s",
        ],
        "targets": ["cave.txt"]
    }
```

Call:

```
$ doit
. echo_to_cave
```

Checking content of the *cave.txt* we shall see:

```
parametrized pssst
parametrized holla
parametrized olla
parametrized lla
parametrized la
parametrized a
```

The string of *cmd-action* is (always) interpolated with keyword *targets* (there are more such keywords). The value for *%(targets)s* is taken from task declaration value (list) converted to space delimited list of file names (so the result is a string).

Who is going to clean that?

Manually deleting files we have created is no fun (imagine you created tens of such files).

doit is ready to clean the mess for you. Simply add *"clean"* keyword to task declaration and provide it with value *True*:

```
def task_echo_to_cave():
    """Call holla to PARAMETRIZED target file"""
    return {
        "actions": [
             "echo parametrized pssst > %(targets)s",
             "echo parametrized holla >> %(targets)s",
             "echo parametrized olla >> %(targets)s",
             "echo parametrized lla >> %(targets)s",
             "echo parametrized la >> %(targets)s",
             "echo parametrized a >> %(targets)s",
        ],
        "targets": ["cave.txt"],
        "clean": True
    }
```

This time ask *doit* to do the cleaning work:

```
$ doit clean
echo_to_cave - removing file 'cave.txt'
```

The file *cave.txt* shall be removed now.

Repeating the last (cleaning) command will not do anything.

Try running the main actions of task *echo_to_cave* again and check, all still works as expected.

Speed up by skipping already done

Its time to do something real. We will need command *7z* installed in our system. Feel free to modify this example with your favourite archiving command.

The plan is to automate compressing huge file in such a way, it recreates the archive only when the source file or target archive has changed.

We will need a file to compress. Best is to use a file, which takes at least few seconds to compress. Put it into current directory and call it *huge.dat*.

Use *7z* command *a* (add to archive):

```
$ 7z a huge.compressed.7z huge.dat
7-Zip [64] 9.20  Copyright (c) 1999-2010 Igor Pavlov  2010-11-18
p7zip Version 9.20 (locale=cs_CZ.UTF-8,Utf16=on,HugeFiles=on,4 CPUs)
Scanning

Creating archive huge.compressed.7z

Compressing  huge.dat

Everything is Ok
```

File *huge.compressed.7z* shall be now present in our directory.

It's time for automation. Create *dodo.py*:

```
def task_compress():
    """Compress input file(s)"""
    return {
        "actions": [
             "7z a %(targets)s %(dependencies)s",
        ],
        "file_dep": ["huge.dat"],
        "targets": ["huge.compressed.7z"],
        "clean": True
    }
```

Remove the *huge.compressed.7z*.

First list the commands available:

```
$ doit list
compress   Compress input file(s)
```

and then run it:

```
$ doit
.  compress
```

The archive file shall be created now.

Ask *doit* to do the work again:

```
$ doit
-- compress
```

Almost zero execution time and "–" in front of task name (instead of ".") tell us, *doit* saw no need to repeat the task, whose results are already up to date.

This will hold true as long as content of input and output files do not change (checked by MD5 checksum).

Assembling mammuth piece by piece

Real tasks are complex. With *doit*, things can be split to smaller tasks and executed step by step until required result is completed.

Following example takes one existing file *huge.dat*, creates another two (*python.help* and *doit.help*) and finally creates *result.7z* containing all three compressed.

```
def task_python_help():
    """Write python help string to a file"""
    return {
        "actions": [
            "python -h > %(targets)s",
        ],
        "targets": ["python.help"],
        "clean": True
    }


def task_doit_help():
    """Write doit help string to a file"""
    return {
        "actions": [
            "doit help > %(targets)s",
        ],
        "targets": ["doit.help"],
        "clean": True
    }


def task_compress():
    """Compress input file(s)"""
    return {
        "actions": [
             "7z a %(targets)s %(dependencies)s",
        ],
        "file_dep": ["huge.dat", "python.help", "doit.help"],
        "targets": ["result.7z"],
        "clean": True
    }
```

File dependencies are clearly defined, tasks are well documented:

```
$ doit list
compress      Compress input file(s)
doit_help     Write doit help string to a file
python_help   Write python help string to a file
```

tasks are run when needed:

```
$ doit
.  python_help
.  doit_help
.  compress
```

processing is optimized and run only when necessary:

```
$ doit
.  python_help
.  doit_help
-- compress
```

and to clean is easy:

```
$ doit clean
compress - removing file 'result.7z'
```

```
doit_help - removing file 'doit.help'
python_help - removing file 'python.help'
```

If you wonder why tasks *python_help* and *doit_help* are always rerun, the reason is, these tasks do not have *file_dep* so they are always considered outdated.

Python defined actions

Todo

show python actions in action

Tutorials

Todo

elaborate on it

Concepts

Reference Manual

Todo

list of topics to cover here is still not complete.

Terms and definitions

dodo file python file, usually named *dodo.py*, containing task definitions in form of functions *task_** which when called, return dictionaries with task declarations.

action simple call of shell command or python function. Action can be parametrized and can calculate values usable in other tasks. Actions can be invoked only via tasks.

task invokable named set of actions and other tasks metadata (required inputs, created outputs, clear behaviour...)

subtask task with parametrized name

task declaration the act of providing dictionary of the task metadata.

target a file, which can be created by a task.

(stateful) file any file mentioned in a task declaration as *file_dep* (task input) or *target* (task output). *doit* keeps track of file existence and state (by default using MD5 checksum) between invocations.

file dependency dependency of a file on another one(s). All files in task target are dependent on files in the task *file_dep*.

task dependency dependency of one task on another one(s). Explicit task dependency is set by *task_dep* in task declaration. Each task is implicitly dependent via file dependency, thus the task is dependent on all tasks creating files mentioned in it's *file_dep*.

task up to date evaluation the act of evaluating all (explicit and implicit) task dependencies and task up to date rules with aim to recognize that the task as a whole is up to date or not.

task execution the act of executing all actions of a task.

task collection the act of collecting all task declarations with aim to get list of invocable tasks and targets.

invokable target the fact, that name of any of files specified in a task's *targets* can be used to invoke the task. Each file from *targets* is invokable.

python-action action defined by means of python function and optionally positional and keyword arguments.

cmd-action action invoked via shell.

cmd-action string cmd-action defined as single string which is supposed to be invoked via shell.

cmd-action list cmd-action defined as list of strings and pathlib file names representing shell command arguments. The cmd-action list is invoked in one shell call.

(task) main actions actions mentioned in task declaration under key *actions*, defining the activity creating task results.

(task) clean actions actions mentioned in task declaration under key *clean*, defining the activity cleaning task results.

(task) up to date actions actions mentioned in task declaration under key *uptodate*, defining the activity determining, if the task is up to date.

result database database persisting status of last task execution incl. files from all *file_dep* and *targets* lists.

DB-Backend specific technology used to serialize data in result database.

command run doit subcommand used to invoke main task actions

command clean doit subcommand used to invoke clean task actions

command list doit subcommand used to list available tasks and subtasks

command ignore doit subcommand used to temporarily disable check of a task results.

command forget doit subcommand used to remove temporarily disabled check of a task results.

Task execution life-cycle

Following chart explains simplified task execution life-cycle.

Fig. 1.1: Task execution life-cycle

Configuration options

Describe: - what can be configured

- doit commands
- tasks (mostly described separately)
- configuration means
  - *DOIT_CONFIG* global variable in dodo file

  - configuration file *doit.cfg*
  - environmental variables
- what are priorities of configuration options

DOIT_CONFIG global variable

describe all keys usable in *DOIT_CONFIG*

- *default_tasks*
- *check_file_uptodate* (custom file up to date checker)
- *backend*
- *dep_file*
- *verbosity*
- *reporter*
- *minversion*

doit.cfg configuration file

Describe: - where is the file looked for (current directory?) - general concept, how are variables named and located - exhaustive list of all sections and options.

Configuration using environmental variables

What environmental variables can be used.

Configuration on task level

In general these options are described elsewhere.

dodo.py file

- it is the default file from which *doit* loads task definitions.
- can have another name, then *doit* must explicitly set it via *-f* option.

action definitions

- cmd-action as string
  - simple string (no interpolation)
  - string with interpolated keyword arguments
  - *CmdAction(cmd_string_builder_fun)*
    - without any argument
    - with keyword arguments (*dependencies*, *changed*, *targets* plus custom ones)

    - parsed cmd-action (cmd-action as string)
- python-action
    - just function name (no explicit keyword argument)
        - not using implicit keyword arguments
        - using implicit keyword arguments
    - function name with positional and keyword arguments
        - not using implicit keyword arguments
        - using implicit keyword arguments

actions calculating values

Actions can calculate values usable by other tasks.

- how to calculate and store value by python-action
- how to calculate and store value by cmd-action (*CmdAction* with *save_out*)

getargs: read values calculated by other tasks

action usage scenarios

When can be actions used:

- main actions
- clean actions
- up to date actions
- ??? some more?

doit invocation

- *doit* CLI
- *$ python -m doit*
- from iPython
- embed into another python code

Todo

Elaborate on it

---

doit command line interface

Todo

Elaborate on it

Task declaration parameters

- *name*
- *basename*
- *actions*
- *file_dep*
- *targets*
- *task_dep*
- *clean*
- *uptodate*
- *title*
- *doc*
- *verbosity*

Task names

- explicit task name
- sub-task name
- implicit task via target reference
- private/hidden tasks

Task selection

- default tasks
- DOIT_CONFIG[”default_tasks”]
- multiple task selection by list
- multiple task selection by wildcard
- indirect selection via task dependency

Task dependencies

- file dependency - default MD5 checksum - custom file status calculation
- explicit task dependency

Task up to date status

- when is task up to date (rules to determine it) - all following conditions must be true
  - each file from *file_dep* is up to date at the moment of task evaluation
  - each file from *targets* is up to date (last execution was successful)
  - calculated up to date is "OK"
- calculating file up to date status
- calculating task up to date status
  - all *file_dep* are up to date

Complex dependency options

Todo

explain, tasks can be declared also later on during execution.

reporter: reporting progress and result

- explain existing options
- refer to customization where custom reporter can be defined

DB-Backend

Todo

purpose, formats, how to extend.

doit customization

- embed into your own product
- implement custom DB-Backend
- develop plugins/libraries
- custom reporter

Tools embedded in *doit*

There are tools, which are not essential to core of *doit* processing engine, but come very often handy. For that reason they are included in *doit*.

Todo

mostly the http://pydoit.org/tools.html stuff

How-to

Topics:

- **installation**
  - system wide installation on Linux
  - system wide installation on Windows
  - virtualenv installation (incl. doit) in one step
- testing and debugging
- using pathlib, shutil and tempfile
- checking binaries are available
- simplify cumbersome command line calls
- handle AWS credentials (use profile set by AWS_DEFAULT_PROFILE)
- standardize release of your python package
  - spell, build doc
  - test, coverage
  - bump version, build distribution packages, upload...
- sync deep directory of files with
  - put/update files in directory of files intended to by synced to FTP/AWS S3/...
  - put *dodo.py* into root of the tree
  - provide tasks:
    - publish
    - fetch
    - sync (will delete on remote what is locally not present)
  - call from anywhere within the tree using *-k* (*–seek-file*)
- multiple files fetch, convert, merge, filter + publish pattern
- Complete lengthy processing under one second
- Cross platform ZIP operations
- Cross platform tar operations
- Share *dodo.py* via directory
- Share doit script via *pypi*

Planning *dodo.py*

- clarify requirements
  - what shall the script do
  - why using doit (speed, orchestration, sharing *dodo.py* with others...)
  - environment it will run in (local machine, the same OS, multiple OS)
- available tools
  - what shell commands can help use
  - can we fulfil the same with pure python?
- identify possible inputs, intermediate files, outputs, actions and tasks
- flowchart: group actions into tasks
- decide about each action implementation (python-action or cmd-action)
- draft *dodo.py*: populate stub *task_** functions incl. docstrings.
- write README.rst (to summarize your plan) and cover:
  - installation
    - method (best use tox based with *requirements.txt*)
    - draft *requirements.txt* to specify required python packages
    - draft list of other shell commands required for the script
  - tasks visible to the user

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index