DepHell Documentation

Release 0.8.3

@orsinium

Jan 11, 2021

Main Info

DepHell – project management for Python.

- Manage dependencies: *convert between formats*, *install*, lock, *add new*, resolve conflicts.
- Manage virtual environments: *create*, *remove*, *inspect*, *run shell*, *run commands inside*.
- *Install CLI tools* into isolated environments.
- Manage packages: *install*, *list*, *search* on PyPI.
- *Build* packages (to upload on PyPI), *test*, *bump project version*.
- *Discover licenses* of all project dependencies, show *outdated* packages, *find security issues*.
- Generate *.editorconfig*, *LICENSE*, *AUTHORS*, *.travis.yml*.

CHAPTER 1

Quick start

1. *Install* DepHell.
2. *Convert* dependencies from one format to another.
3. *Make config* for DepHell to simplify commands.

And that's it! Now you can use any tools with any Python project. Every other commands build around ability to read project dependencies and metadata and resolve conflicts. When you ready to boost your productivity, read how to manage your environment with DepHell:

1. *Create virtual environment*.
2. *Install* or *synchronize* dependencies.
3. *Run* commands inside or *activate virtual environment*.

1.1 Installation

Stable version

```
curl -L dephell.org/install | python3
```

It will install DepHell into DepHell's jail (WE NEED TO GO DEEPER).

If installation script doesn't work:

```
python3 -m pip install --user dephell[full]
```

This command installs DepHell globally, without virtual environment. Extra `full` is optional and installs some dependencies that isn't required, but makes DepHell a little bit better. So, if you can't install by the recommend way and have some conflicts with your global modules, install without it:

```
python3 -m pip install --user dephell
```

1.1.1 Get development version

Install dev version inside of site-packages:

```
curl -L dephell.org/install > install.py
python3 install.py --branch=master
```

Or get whole project and teach Python to use it:

```
$ git clone https://github.com/dephell/dephell.git
$ cd dephell
$ sudo python3 -m pip install -e .
```

1.2 Configuration and parameters

Dephell makes config from 4 layers:

1. Default parameters.
2. Section from config file.
3. Environment variables.
4. CLI arguments.

1.2.1 Config file

Config should be TOML file with `tool.dephell.ENV_NAME` sections (see PEP-518).

1. By default, dephell tries to read `pyproject.toml` or `dephell.toml`. You can change it by `--config` argument.
2. Default environment: `main`. Environment is the name of the section inside of `tool.dephell` section in config file. You can change environment by `--env` argument.

Config example:

```
[tool.dephell.main]
# read from poetry format
from = {format = "poetry", path = "pyproject.toml"}
# drop dev-dependencies
envs = ["main"]
# and convert into setup.py
to = {format = "setuppy", path = "setup.py"}

[tool.dephell.pytest]
# read dependencies from setup.py
from = {format = "setuppy", path = "setup.py"}
# install main dependencies and `tests` extra dependencies
envs = ["main", "tests"]
# run command `pytest`
command = "pytest"
```

1.2.2 CLI arguments

You can (re)define any config options with CLI arguments. For example, there is how you can define the same parameters as in the `pytest` section above:

```
$ dephell deps install \
    --from-format setuppy \
    --from-path setup.py \
    --envs main tests --

$ dephell deps run --command="pytest"

# Also for `venv run` you can specify command as positional argument:
$ dephell venv run pytest
```

It's OK for one-time actions, but for everyday usage we recommend to define config section for every kind of tasks you perform. For example, for dephell we have defined envs `main` to convert from poetry to setup.py, `flake8` for linting, `pytest` for tests, and `docs` for generating documentation. So, you can install and run test environment for dephell much simpler:

```
$ dephell venv create --env=pytest
$ dephell deps install --env=pytest
$ dephell venv run --env=pytest
```

Also, by default, DepHell uses `--env` to generate path to the virtual environment, so different `--env` values have different virtual environments.

1.2.3 Environment variables

Sometimes, specifying config parameters in environment variables can be more suitable for you. Most common case is to set up `env` or path to config file. For example:

```
export DEPHELL_ENV=flake8
export DEPHELL_CONFIG="./project/dephell.toml"

# commands below will be executed with specified above env and path to config
dephell venv create
dephell deps install
dephell venv run

# do not forget to remove variables after all
unset DEPHELL_ENV
unset DEPHELL_CONFIG
```

DepHell do type casting in the same way as dynaconf. Just use TOML syntax for values:

```
# Numbers
DEPHELL_CACHE_TTL=42
DEPHELL_SDIST_RATIO=0.5

# Text
DEPHELL_FROM_FORMAT=pip
DEPHELL_FROM_FORMAT="pip"

# Booleans
DEPHELL_SILENT=true
```

(continues on next page)

(continued from previous page)

```
DEPHELL_SILENT=false

# Use extra quotes to force a string from other type
DEPHELL_PYTHON="'3.6'"
DEPHELL_PROJECT="'true'"

# Arrays
DEPHELL_ENVS="['main', 'dev']"

# Dictionaries
DEPHELL_FROM='{format="pip", path="req.txt"}'
```

1.2.4 See also

1. *inspect config command* to discover how dephell makes config for your project.
2. dephell's own config to see real and full example.

1.3 Parameters list

Parameters represented as CLI arguments. To make config file parameter name from CLI name just strip `--` from the begining and split by `-`.

For example, `--cache-path=.cache --` and `--project=./dephell` can be written in the next way:

```
[tool.dephell.main]
cache = {path = ".cache"}
project = "./dephell"
```

To make sure which of these options accepted by some command use `dephell COMMAND --help`. For example, `dephell deps convert --help`.

1.3.1 Select config file and environment

- `-c`, `--config` – path to config file.
- `-e`, `--env` – environment in config.

Of course, you can use this options only in CLI. You can't specify path to config in the config :)

1.3.2 Paths to dependencies

- `--from` – path or format for reading requirements. If it is format then dephell will scan current directory to find out file that can be parsed by this converter. If it is filename then dephell will automatically determine file format.
- `--from-format` – format for reading requirements. See *deps convert* command documentation for full list of accepted formats.
- `--from-path` – path to input file.
- `--to` – path or format for writing requirements.

- `--to-format` – output requirements file format.
- `--to-path` – path to output file.
- `--sdist-ratio` – ratio of tests and project size after which tests will be excluded from sdist. By default is 2 that means tests will be included while their size less than doubled package size.

Commands that accept these parameters:

- Only `deps convert` accepts `from` and `to` at the same time.
- `deps install` prefers `to` option if available. This is because when you specified in config file source dependencies in `from` and locked dependencies in `to` then, of course, you want to install dependencies from lock file. However, if `to` (and `to-format` and `to-file`) isn't specified in the config and CLI arguments then `from` will be used.
- `deps licenses` uses dependencies from `from`, lock them and shows licenses specified on PyPI.
- `jail install`, `venv create`, `venv run`, and `venv shell` commands use `from` to determine preferred python version for project.

1.3.3 Resolver and API

- `--strategy` – algorithm to select best release. Available values: `min` and `max`. By default is `max`, because almost all resolvers uses this strategy. Read blog post Minimal Version Selection for details about `min` strategy.
- `--prereleases` – allow prereleases.
- `--mutations` – maximum mutations when trying to resolve conflicts. 200 by default.
- `--warehouse` – warehouse URLs or local paths to archives with releases.
- `--bitbucket` – bitbucket API URL. Dephell isn't use Bitbucket API yet, but option already available.
- `--repo` – force repository for first-level dependencies. Useful when you want to use `conda` instead of `pypi` (for example, in *dephell package search* command).

1.3.4 Virtual environment

- `--venv` – path to venv directory for project. Replacements:
  - `{project}` will be replaced by the project name (name of path from `project` option, this is name of the current directory by default).
  - `{digest}` will be replaced by the short 4-letters digest of the project path to avoid conflicts for the projects with the same name in different locations.
  - `{env}` will be replaced by current environment (`main` by default).
- `--python` – python version for venv. This can be reloaded in the dependencies file.
- `--dotenv` – path to a .env file. Will be sourced on a venv activation.
- `vars` (config only) – dict of environment variables to pass in virtual environment.

1.3.5 Docker

- `--docker-repo` – image name without tag to use for *Docker-based commands*.
- `--docker-tag` – image tag.

- `--docker-container` – container name. By default, automatically generated from the project name.

1.3.6 Project upload

- `--upload-url` – URL of API endpoint to use to upload dist.
- `--sign` – a flag indicates that dists must be signed before uploading.
- `--identity` – GPG identity to use to sign dists.

1.3.7 Output

- `--format` – output format.
- `--level` – minimal level for log messages. Available levels: `DEBUG`, `INFO`, `WARNING`, `ERROR` and `EXCEPTION`. `INFO` by default. `DEBUG` and `INFO` writes in the stdout, other levels in the stderr.
- `--nocolors` – do not color output.
- `--table` – format output as a nice ASCII table.
- `--silent` – suppress any output except errors. Disables progress bar for resolver.
- `--filter` – *filter for JSON output*.
- `--traceback` – show traceback for exceptions.
  - `--pdb` – run pdb when critical exception occurred.

Other:

- `--owner` – name of the owner.
- `--cache-path` – path to dephell cache.
- `--cache-ttl` – Time to live for releases list cache (in seconds). 1 hour by default.
- `--project` – path to the current project. Current directory by default.
- `--bin` – path to the dir for installing scripts.
- `--ca` – path to a custom CA bundle file. If provided, will be used for both `requests` and `aiohttp`.
- `--envs` – environments (`main`, `dev`) or extras to install or convert.
- `--tests` – path to test files for *dephell project test* command.
- `--versioning` – versioning scheme for project. See *dephell project bump* for details.
- `--tag` – template for git tag to use in *dephell project bump*.
- `command` (config-only) – default command to run in *dephell venv run* and *dephell docker run*.
- `--vendor-exclude` – dependencies that shouldn't be *vendorized*.
- `--vendor-path` – path to store vendorized dependencies..

1.3.8 Default values

Default values a little bit varies for different systems. Please, use *inspect config* to view your actual config for current system, project and environment.

1.4 Filter JSON output

JSON output of any command can be filtered with `--filter` argument.

1.4.1 Commands with JSON output

- *dephell deps licenses*
- *dephell inspect config*
- *dephell inspect venv*
- *dephell jail list*
- *dephell package list*
- *dephell package search*
- *dephell package show*

1.4.2 Filters

Filters separated by `.` or `-` and can be one of the following type:

- Field name to get some field from dict output.
- Sum of fields. Will return dictionary with given fields. For example, `name+license` will return `{"license": "BSD-2-Clause", "name": "click"}`.
- Index to get some element from list output.
- Slice to get set of elements from list output. For example:
  - `:10` to get first 10 elements,
  - `10:` to drop out first 10 elements,
  - `2:5` to get elements with indices 2, 3 and 4.
- Function to process output.

Functions:

- `each()` or `#` – convert list of dicts to dict of lists and vice versa. For example, `[{a: 1, b: 2}, {a: 3, b: 4}]` will be converted into `{a: [1, 3], b: [2, 4]}`.
- `first()` or `0` – get first element from list.
- `flatten()` or `flat()` – squash list of lists into one-level (flat) list.
- `last()` or `latest()` – get last element from list.
- `len()`, `length()`, `count()` or `size()` – get count of elements in a list.
- `max()` – get maximum value from a list.
- `min()` – get minimum value from a list.
- `reverse()` or `reversed()` – reverse values in a list.
- `sort()` or `sorted()` – sort values in a list.
- `sum()` – sum of values in a list.
- `type()` – get value type.

- `zip()` – transpose output. `[[a, b], [c, d], [e, f]]` will be converted into `[[a, c, e], [b, d, f]]`.

First filter gets command output. Next filters get output from previous filter.

1.4.3 Example

Let's filter output of *dephell package show*:

```
$ dephell package show textdistance
{
  "authors": [
    "orsinium <master_fess@mail.ru>"
  ],
  "description": "Compute distance between the two texts.",
  "installed": [],
  "latest": "4.1.2",
  "license": "MIT",
  "links": {
    "download": "https://github.com/orsinium/textdistance/tarball/master",
    "homepage": "https://github.com/orsinium/textdistance",
    "package": "https://pypi.org/project/textdistance/"
  },
  "name": "textdistance",
  "updated": "2019-03-18"
}
```

Get some fields:

```
# one field value:
$ dephell package show --filter=latest textdistance
4.1.2

# a few fields:
$ dephell package show --filter="latest+installed" textdistance
{
  "installed": [],
  "latest": "4.1.2"
}
```

Filter list items:

```
$ dephell package show --filter=authors click
[
  "Armin Ronacher <armin.ronacher@active-4.com>",
  "Pallets Team <contact@palletsprojects.com>"
]

# first element
$ dephell package show --filter="authors.first()" click
Armin Ronacher <armin.ronacher@active-4.com>

# last element
$ dephell package show --filter="authors.last()" click
Pallets Team <contact@palletsprojects.com>

# get element by index
```

(continues on next page)

(continued from previous page)

```
$ dephell package show --filter="authors.0" click
Armin Ronacher <armin.ronacher@active-4.com>

# reverse list
$ dephell package show --filter="authors.reverse()" click
[
  "Pallets Team <contact@palletsprojects.com>",
  "Armin Ronacher <armin.ronacher@active-4.com>"
]

# get records count
$ dephell package show --filter="authors.len()" click
2
```

Work with items in a list:

```
dephell package search author:orsinium
[
  {
    "description": "Work with python versions",
    "name": "dephell-pythons",
    "url": "https://pypi.org/project/dephell-pythons/",
    "version": "0.1.0"
  },
  ...
]

# get field from each record
$ dephell package search --filter="#.name" author:orsinium
[
  "dephell-discover",
  "pros",
  "homoglyphs",
  ...
]

# sort
$ dephell package search --filter="#.name.sort()" author:orsinium
[
  "advice",
  "aop",
  "deal",
  ...
]

# get a few fields
$ dephell package search --filter="#.name+description.each()" author:orsinium
[
  {
    "description": "Find project modules and data files (packages and package_data␣
↪for setup.py).",
    "name": "dephell-discover"
  },
  {
    "description": "UNIX pipeline on python and steroids",
    "name": "pros"
  },
```

(continues on next page)

(continued from previous page)

```
  ...
]

# get only first 10 elements for previous filter:
$ dephell package search --filter="#.name+description.each().:10" author:orsinium
```

1.4.4 Alternatives

In some rare cases you could want to specify some complex filter that not covered by DepHell. So, you can process DepHell output into some other command that can process JSON. Some of them:

- jq
- jj
- jd

Also, it's recommend for better processing to disable INFO-messages and progress bars. For example:

```
$ dephell deps licenses --level=WARNING --silent | jq --compact-output '."Apache-2.0"'
["aiofiles","aiohttp",...]
```

1.5 Python and venv lookup

Some commands try to find out the best venv to get information about packages or the best python executable to install packages or create venv. There is described lookup order for these commands.

1.5.1 Python interpreter lookup

1. `python` parameter in the *config* (or as CLI argument `--python`, of course). You can define python here in any way as you want:
    1. Version (`3.7`)
    2. Exact version (`3.7.2`)
    3. Constraint (`>=3.5`)
    4. Executable name (`python3`)
    5. Path to the executable (`/usr/bin/python3`)
2. Dependencies file defined in `from` parameter. For example, python_requires from setup.py.
3. If nothing was found current interpreter (that runs DepHell) will be used.

This lookup is used in commands that can create virtual environment:

- *dephell jail install*
- *dephell venv create*
- *dephell venv run*
- *dephell venv shell*

1.5.2 Virtual environment (venv) lookup

1. If virtual environment for current project (can be specified with `--config`) and environment (can be specified with `--env`) exists then this virtual environment will be used. This is the reason why you have to *create virtual environment* before dependencies installation. Can be overwritten by `--venv` parameter.
2. If some venv is active then it will be used.

This lookup is used in command *dephell inspect venv*.

1.5.3 Python environment

Python environment – any python interpreter: virtual environment or globally installed interpreter. This lookup used when DepHell looks for place to work with packages (analyze, install, remove).

1. First of all, DepHell tries to find virtual environment by virtual environment lookup.
2. If there is no virtual environment then DepHell looks for best global interpreter by Python interpreter lookup.

Commands that use this lookup:

- *dephell deps install*
- *dephell deps outdated*
- *dephell package install*
- *dephell package list*
- *dephell package show*

If you want to force DepHell ignore project venvs and use global interpreter you can pass into command non-existent venv:

```
$ dephell package install --venv=none homoglyphs
INFO build dependencies graph...
INFO installation... (executable=/usr/local/bin/python3.7, packages=1)
...
```

1.6 deps: project dependencies

Commands to manage project dependencies: *convert between formats*, *install*, lock, *add new*, resolve conflicts, do *security audit*, *review licenses*, *find outdated packages* in lockfile or environment, *build dependencies tree*.

1.6.1 dephell deps add

Add new dependencies into project.

Algorithm:

1. Get dependencies from `from` file.
2. Add new dependencies.
3. Check that these new dependencies has no conflicts with existing.
4. Write dependencies back into `from` file.

You can specify `--envs` to add dependencies into.

Basic usage

Simple usage:

```
dephell deps add --from=poetry flake8
```

Best practice is specify your dependencies file in `pyproject.toml` DepHell config:

```
[tool.dephell.main]
from = {format = "poetry", path = "pyproject.toml"}
```

And after that DepHell will automatically detect your dependencies file:

```
dephell deps add flake8
```

See *configuration documentation* for more details.

Specify dependencies environments

Environments for dependencies is the name of dependencies section (`main` and `dev` for `poetry` and `pipfile`) or name of extras. DepHell uses `main` by default, but you can specify another one:

```
dephell deps add --envs dev tests -- flake8==3.1.0 pytest
```

This will produce dependencies next lines in your poetry config:

```
[tool.poetry.dev-dependencies]
pytest = "*"
flake8 = "==3.1.0"

[tool.poetry.extras]
tests = ["flake8", "pytest"]
```

See also

1. *How to configure DepHell*.
2. *How to filter commands JSON output*.
3. *dephell deps convert* for details about locking dependencies and supported file formats.
4. *dephell deps install* to install new dependencies into virtual environment.
5. *dephell package install* to install single package without adding it into requirements.

1.6.2 dephell deps audit

Returns list of known vulnerabilities for your dependencies.

This command returns non-zero code if some vulnerabilities was found, so you can use it on CI.

Sources

pyup.io provides public repository safety-db with all vulnerabilities in their database. DepHell uses it. This repository automatically updates every month. So, if you want to get actual reports you have to use their official solutions. They provide Safety CI that free for Open Source and $30 for personal usage. If you have "Business" plan you also can get API key and use their official CLI.

Dependencies lookup

1. If some package and version explicitly specified then this package will be used. Example: `dephell deps audit jinja2==2.0`.
2. If `to` format is a lockfile (`piplock`, `pipfilelock` or `poetrylock`) dependencies from this file will be used.
3. If `to` isn't specified and `from` is a lockfile dependencies from this file will be used.
4. Otherwise it uses common *Python environment lookup*. TL;DR: project venv, current venv, python from config, python from dependencies file, current interpreter.

Examples

Audit dependencies:

```
$ dephell deps audit
[
  {
    "current": "2.10",
    "description": "Sandbox Escape in jinja2 (pip) with medium severity ",
    "latest": "2.10.1",
    "links": [
      "https://pypi.org/project/Jinja2/",
      "https://palletsprojects.com/blog/jinja-2-10-1-released",
      "https://snyk.io/vuln/SNYK-PYTHON-JINJA2-174126"
    ],
    "name": "jinja2",
    "updated": "2019-04-06",
    "vulnerable": "<2.10.1"
  }
]
```

Audit a package:

```
$ dephell deps audit jinja2==2.0
[
  {
    "current": "2.0",
    "description": "jinja2 2.7.2 fixes a security issue: Changed the default folder␣
↪for the filesystem cache to be user specific and read and write protected on UNIX␣
↪systems.  See  for more information.",
    "latest": "2.10.1",
    "links": [
      "http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=734747"
    ],
    "name": "jinja2",
    "updated": "2019-04-06",
```

(continues on next page)

(continued from previous page)

```
    "vulnerable": "<2.7.2"
  },
  ...
]
```

Show only descriptions:

```
$ dephell deps audit --filter="#.description" jinja2==2.0
[
  "jinja2 2.7.2 fixes a security issue: Changed the default folder for the filesystem␣
↪cache to be user specific and read and write protected on UNIX systems.  See  for␣
↪more information.",
  "The default configuration for bccache.FileSystemBytecodeCache in Jinja2 before 2.7.
↪2 does not properly create temporary files, which allows local users to gain␣
↪privileges via a crafted .cache file with a name starting with __jinja2_ in /tmp.",
  "Sandbox Escape in jinja2 (pip) with medium severity "
]
```

Show only links:

```
$ dephell deps audit --filter="#.links.flatten()" jinja2==2.0
[
  "http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=734747",
  "https://nvd.nist.gov/vuln/detail/CVE-2014-1402",
  "https://pypi.org/project/Jinja2/",
  "https://palletsprojects.com/blog/jinja-2-10-1-released",
  "https://snyk.io/vuln/SNYK-PYTHON-JINJA2-174126"
]
```

See *filtering documentation* for more information how to work with JSON output.

See also

1. *How DepHell choose Python environment*.
2. *How to filter commands JSON output*.
3. *dephell deps outdated* to find outdated dependencies.
4. *dephell package list* to show information about installed packages.
5. *dephell package show* to get information about package.

1.6.3 dephell deps check

Show difference between virtual environment and project dependencies.

```
dephell deps check
INFO get dependencies (format=setuppy, path=setup.py)
INFO build dependencies graph...
[
  {
    "action": "remove",
    "installed": "2.2.1",
    "locked": null,
```

(continues on next page)

(continued from previous page)

```
    "name": "deal"
  }
]
```

Fields:

- `action` – what would happened if you run *dephell deps sync*. Can be `install`, `update` or `remove`.
- `name` – package name (wow!).
- `installed` – installed version of a package. `null` if a package isn't installed.
- `locked` – version of a package in the dependencies graph or locked by resolver. `null` if a package isn't represented in dependencies graph.

See also

1. *How DepHell choose Python environment*.
2. *dephell deps install* to install project dependencies.
3. *dephell venv create* to create virtual environment for dependencies.
4. *dephell package install* to install single package
5. *dephell jail install* to install some Python CLI tool into isolated virtual environment.

1.6.4 dephell deps convert

Convert dependencies between formats. Dephell will automagically lock them if needed and resolve all conflicts.

Dephell uses four pieces of information for conversion:

1. `--from-format`: The format to convert from (e.g. `poetry`)
2. `--from-path`: The path to the file to read from (e.g. `pyproject.toml`)
3. `--to-format`: The format to convert to (e.g. `setuppy`)
4. `--to-path`: The path to the file where the result should be put (e.g. `setup.py`). You can provide the special case 'stdout' to this option to output to the screen instead of a file.

Dephell can try to guess the formats or paths you want to use given the other piece of information, giving you three different ways to specify what you want:

1. Explicitly specify path and format: `--from-format=poetry --from-path=pyproject.toml` and `--to-format=setuppy --to-path=setup.py`.
2. Specify only path: `--from=pyproject.toml` and `--to=setup.py`.
3. Specify only format: `--from=poetry` and `--to=setuppy`.

Supported formats

1. Archives:
    1. *.egg-info (`egginfo`)
    2. *.tar.gz (`sdist`)
    3. *.whl (`wheel`)

2. pip:
    1. requirements.txt (`pip`)
    2. requirements.lock (`piplock`)
3. pipenv:
    1. Pipfile (`pipfile`)
    2. Pipfile.lock (`pipfilelock`)
4. poetry:
    1. pyproject.toml (`poetry`)
    2. poetry.lock (`poetrylock`)
5. Environment:
    1. Imports in the package (`imports`). Pass path to package or module and dephell will automatically detect required packages.
    2. Installed packages (`installed`). It works like pip freeze. Dephell can only read from this format, of course. If you want to install packages, use *install command*.
6. Other:
    1. setup.py (`setuppy`)
    2. flit (`flit`)
    3. conda's environment.yml (`conda`)
    4. pyproject.toml build-system requires (`pyproject`)

Examples

Lock dependencies for Pipfile:

```
$ dephell deps convert --from=Pipfile --to=Pipfile.lock
```

Or the same, but more explicit:

```
$ dephell deps convert \
    --from-format=pipfile --from-path=Pipfile \
    --to-format=pipfilelock --to-path=Pipfile.lock
```

Best practice is specify your dependencies file in `pyproject.toml` DepHell config:

```
[tool.dephell.main]
from = {format = "pipfile", path = "Pipfile"}
to = {format = "pipfilelock", path = "Pipfile.lock"}
```

And after that DepHell will automatically detect your dependencies file:

```
$ dephell deps convert
```

You can still override this config for one-off actions:

```
$ dephell deps convert --to requirements.txt
```

See *configuration documentation* for more details.

More examples

You can convert anything to anything:

1. Lock requirements.txt: `dephell deps convert --from=requirements.in --to=requirements.txt`
2. Lock Pipfile: `dephell deps convert --from=Pipfile --to=Pipfile.lock`
3. Lock poetry: `dephell deps convert --from=pyproject.toml --to=poetry.lock`
4. Migrate from setup.py to poetry: `dephell deps convert --from=setup.py --to=pyproject.toml`
5. Migrate from pipenv to poetry: `dephell deps convert --from=Pipfile --to=pyproject.toml`
6. Generate setup.py for poetry (to make project backward compatible with setuptools): `dephell deps convert --from=pyproject.toml --to=setup.py`
7. Generate requirements.txt from Pipfile to work on a pipenv-based project without pipenv: `dephell deps convert --from=Pipfile --to=requirements.txt`
8. Generate requirements.txt from poetry to work on a poetry-based project without poetry: `dephell deps convert --from=pyproject.toml --to=requirements.txt`
9. Pipe poetry requirements into pip for installation in a custom environment: `pip install -r <(dephell deps convert --to-path stdout --to-format pip)`

Filter dependencies

You can filter dependencies by envs with `--envs` flag. All dependencies included in `main` or `dev` env. Also, some dependencies can be included in extras. There is an example of poetry config with envs in comments:

```
[tool.poetry.dependencies]
python = ">=3.5"
aiohttp = "*"       # main, asyncio
textdistance = "*"  # main

[tool.poetry.dev-dependencies]
pytest = "*"    # dev, tests
sphinx = "*"    # dev

[tool.poetry.extras]
asyncio = ["aiohttp"]
tests = ["pytest"]
```

Examples, how to filter these deps:

```
$ dephell deps convert --envs main
# aiohttp, textdistance

$ dephell deps convert --envs asyncio
# aiohttp

$ dephell deps convert --envs main tests
# aiohttp, textdistance, pytest
```

See also

1. *dephell project build* to fast convert dependencies into setup.py, sdist and wheel.
2. *dephell deps install* to install project dependencies.
3. *dephell deps tree* to show dependencies tree for project.

1.6.5 dephell deps install

Install project dependencies.

Dependencies from `to` option will be used if available. This is because when you specified in config file source dependencies in `from` and locked dependencies in `to` then, of course, you want to install dependencies from lock file. However, if `to` (and `to-format` and `to-file`) isn't specified in the config and CLI arguments then `from` will be used.

Place to install lookup:

1. If some virtual environment already active in the current shell then this environment will be used.
2. If virtual environment for current project (can be specified with `--config`) and environment (can be specified with `--env`) exists then this virtual environment will be used. This is the reason why you have to *create virtual environment* before dependencies installation.
3. If virtual environment isn't found then your current python will be used.

See also

1. *How DepHell choose Python environment*.
2. *dephell deps sync* to install project dependencies and drop obsolete packages.
3. *dephell venv create* to create virtual environment for dependencies.
4. *dephell package install* to install single package
5. *dephell jail install* to install some Python CLI tool into isolated virtual environment.
6. *dephell project register* to make the current project importable from another project.

1.6.6 dephell deps licenses

This command shows license for all your project's dependencies (from `from` section of current environment) in JSON format. Dephell detects the same license described in the different ways, like "MIT" and "MIT License", and combine these dependencies together. Dephell shows licenses **for all project's dependencies** including dependencies of dependencies.

```
$ dephell deps licenses

INFO resolved
{
  "Apache-2.0": [
    "aiofiles",
    "aiohttp",
    ...
  ],
  ...
```

(continues on next page)

(continued from previous page)

```
  "Python Software Foundation License": [
    "backports-weakref",
    "editorconfig",
    "typing",
    "typing-extensions"
  ],
}
```

If you want to process this JSON to other tool disable dephell's helping output with `--level` and `--silent` arguments:

```
$ dephell deps licenses --level=WARNING --silent | jq --compact-output '."Apache-2.0"'
["aiofiles","aiohttp",...]
```

This example uses jq to filter only one license from output. However, for simple filtering by license name you can just pass this name as positional argument in the command:

```
$ dephell deps licenses --filter="Apache-2.0"

INFO resolved
[
  "aiofiles",
  "aiohttp",
  ...
]
```

See also

1. *Example of this command usage*
2. *dephell generate license* to make license file for your project.
3. *How dephell works with config and parameters*
4. *Full list of config parameters*

1.6.7 dephell deps outdated

Show outdated project dependencies. It compares latest package version on PyPI and version in the lockfile or project environment and shows packages that version is different.

Place to get dependencies from lookup:

1. If `to` format is a lockfile (`piplock`, `pipfilelock` or `poetrylock`) dependencies from this file will be used.
2. If `to` isn't specified and `from` is a lockfile dependencies from this file will be used.
3. Otherwise it uses common *Python environment lookup*. TL;DR: project venv, current venv, python from config, python from dependencies file, current interpreter.

Some packages can have different version because their latest version incompatible with some other project dependencies, and DepHell's dependency resolver has locked their older (compatible) version. These packages also will be listed in the `dephell deps outdated` command output because explicit better than implicit.

This command returns non-zero code if some vulnerabilities was found, so you can use it on CI.

Usage

Show all outdated packages:

```
$ dephell deps outdated

[
  {
    "description": "More routines for operating on iterables, beyond itertools",
    "locked": "6.0.0",
    "latest": "7.0.0",
    "name": "more-itertools",
    "updated": "2019-03-28"
  },
  ...
]
```

Filter only package name and latest release upload time:

```
$ dephell deps outdated --filter="#.name+updated.each()"
INFO get packages from project environment (path=/home/gram/.local/share/dephell/
↪venvs/dephell-nLn6/main)
[
  {
    "name": "more-itertools",
    "updated": "2019-03-28"
  },
  ...
]
```

See also

1. *How DepHell choose Python environment*.
2. *How to filter commands JSON output*.
3. *dephell deps audit* to check dependencies for known vulnerabilities.
4. *dephell package list* to show information about installed packages.
5. *dephell package show* to get information about package.
6. *dephell venv create* to create virtual environment for dependencies.
7. *dephell package install* to install a single package.

1.6.8 dephell deps sync

This command works in the same way as *dephell deps install*, but also removes from environment all packages that not presented in project dependencies (obsolete).

See also

1. *How DepHell choose Python environment*.
2. *dephell venv create* to create virtual environment for dependencies.

3. *dephell deps install* to install dependencies without dropping obsolete packages.
4. *dephell package install* to install single package.
5. *dephell jail install* to install some Python CLI tool into isolated virtual environment.

1.6.9 dephell deps tree

Show dependencies tree for your dependencies from `from` section or given package.

Show project dependencies:

```
$ dephell deps tree
- aiofiles [required: *, locked: 0.4.0, latest: 0.4.0]
- aiohttp [required: *, locked: 3.5.4, latest: 3.5.4]
  - async-timeout [required: <4.0,>=3.0, locked: 3.0.1, latest: 3.0.1]
  - attrs [required: >=17.3.0, locked: 19.1.0, latest: 19.1.0]
  - chardet [required: <4.0,>=2.0, locked: 3.0.4, latest: 3.0.4]
  - idna-ssl [required: >=1.0, locked: 1.1.0, latest: 1.1.0]
    - idna [required: >=2.0, locked: 2.8, latest: 2.8]
  ...
```

Field `locked` shows version that was resolved by this command, **not** the version that represented in any environment or lockfile.

Show dependencies for given package:

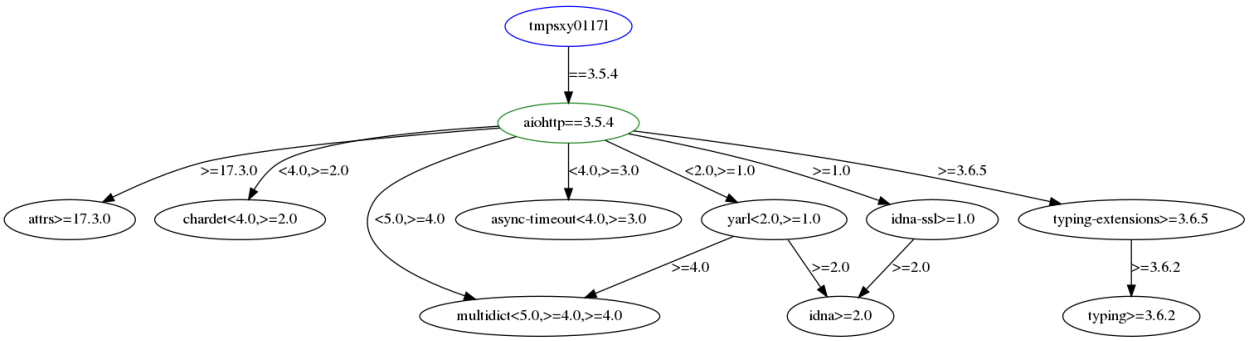
```
$ dephell deps tree aiohttp==3.5.4
- aiohttp [required: ==3.5.4, locked: 3.5.4, latest: 3.5.4]
  - async-timeout [required: <4.0,>=3.0, locked: 3.0.1, latest: 3.0.1]
  - attrs [required: >=17.3.0, locked: 19.1.0, latest: 19.1.0]
  - chardet [required: <4.0,>=2.0, locked: 3.0.4, latest: 3.0.4]
  - idna-ssl [required: >=1.0, locked: 1.1.0, latest: 1.1.0]
    - idna [required: >=2.0, locked: 2.8, latest: 2.8]
  ...
```

Graph output

You can specify `--type=graph` to build dependencies graph:

```
$ dephell deps tree --type=graph aiohttp==3.5.4
```

It will create next graph in `.dephell_report` directory:



graph example

JSON output

You can specify `--type=json` to generate JSON with information for every node in graph:

```
$ dephell deps tree --type=json aiohttp==3.5.4
[
  {
    "best": "3.5.4",
    "constraint": "==3.5.4",
    "dependencies": [
      "attrs",
      "chardet",
      "multidict",
      "async-timeout",
      "yarl",
      "idna-ssl",
      "typing-extensions"
    ],
    "latest": "3.5.4",
    "name": "aiohttp"
  },
  ...
]
```

As for any other command, you can *filter* JSON output:

```
dephell deps tree --type=json --filter="#.name+constraint.each()" aiohttp==3.5.4
[
  {
    "constraint": "==3.5.4",
    "name": "aiohttp"
  },
  {
    "constraint": "<4.0,>=3.0",
    "name": "async-timeout"
  },
  ...
]
```

See also

1. *Command usage example for git repo*.
2. *How to filter commands JSON output*.
3. *dephell package outdated* to show outdated packages in a lockfile or project virtual environment.
4. *dephell package list* to show information about installed packages.
5. *dephell package show* to get information about single package.

1.7 docker: venv on steroids

Commands to simply manage Docker containers for the current project: *create* and *destroy*, run *shell* and *commands* inside, *make it cool*. Motivation isn't wrap around whole Docker, but make it easier for newbies to work with simple

containers and integrate it with DepHell. It's achieved by providing all commands from *dephell venv* group and a little bit more.

1.7.1 dephell docker create

Create a new docker container for the project. Usually, you don't need to call this command directly because all other commands in *dephell docker* group (except *dephell docker destroy*) will create container if it doesn't exist.

```
$ sudo dephell docker create
INFO creating container for project... (container=dephell-dephell-nLn6-main)
INFO image not found, pulling... (repository=python, tag=latest)
INFO pulled
INFO container created
```

By default, the command creates container with authogenerated name (based on project path and current environment) from python:latest You can specify these parameters in *dephell config*:

```
[tool.dephell.main.docker]
container = "container-name"
repo = "python"
tag = "3.7.4-stretch"
```

Also, DepHell mounts your current directory into `/opt/project/` inside the container. However, it won't be mounted if you're running this command from root or home folder because it's too much to mount.

See also

1. *dephell docker prepare* to make a container nice.
2. *dephell venv create* to create a virtual environment (less isolation, better integration).
3. *dephell docker destroy* to remove a container.
4. *dephell docker shell* to run shell inside a container.

1.7.2 dephell docker destroy

Remove docker container for current project and environment. Like *dephell venv destroy*, but for docker.

See also

1. *dephell docker stop* to stop processes in a container without removing it.
2. *dephell docker create* to create a new container.

1.7.3 dephell docker prepare

Installs some nice things into container to make work with it more pleasant:

- zsh
- oh-my-zsh
- glances

- pure
- dephell

See also

1. *dephell docker create* to create a clean container.
2. *dephell docker shell* to run a shell inside a container.
3. *dephell docker run* to run a command inside a container.

1.7.4 dephell docker run

Run a command inside the Docker container.

```
$ sudo dephell docker run echo "Hello, world"

[sudo] password for gram:
INFO running... (container=dephell-dephell-nLn6-main, command=['echo', 'Hello, world
↪'])
Hello, world
INFO done
```

If a command isn't specified, dephell will try to get it from *config*:

```
[tool.dephell.main]
command = "echo 'Hello, world!'"
```

See also

1. *dephell docker create* to read how dephell creates a new container.
2. *dephell docker shell* to run a shell inside a container.
3. *dephell docker stop* to stop command execution inside a container.

1.7.5 dephell docker shell

Run a shell inside of the Docker container for a current project and environment. Dephell tries to get the best shell in the following order:

1. zsh
2. bash
3. sh

The command is also useful for quick experiments with project in the isolated environment:

```
$ sudo dephell docker shell --docker-container=tmp
WARNING creating container... (container=tmp)
INFO openning shell... (container=tmp)
sh: 1: zsh: not found
root@d6ceb924fea6:/opt/project#
```

See also

1. *dephell docker create* to read how dephell creates a new container.
2. *dephell docker run* to run a command inside a container.

1.7.6 dephell docker stop

The command stops the Docker container for a current project and environment. It works like docker stop. If something (even shell) executes inside the container, it will be stopped. However, you won't lost created files and installed programs inside of the container. If you want to get rid of everything inside, use *dephell docker destroy*.

See also

1. *dephell docker destroy* to remove a container.
2. *dephell docker shell* to run a shell inside a container.
3. *dephell docker run* to run a command inside a container.

1.7.7 dephell docker tags

Get available tags for a docker repository on Docker Hub. Use *filters* to get only last 10 tags:

```
$ dephell docker tags --docker-repo=elasticsearch --filter=:10
WARNING cannot find config file
[
  "7.2.0",
  "6.8.1",
  "7.1.1",
  "7.1.0",
  "6.8.0",
  "7.0.1",
  "6.7.2",
  "7.0.0",
  "6.7.1",
  "5-alpine"
]
```

See also

1. *How filters work*
2. *dephell docker create* to create a new container.

1.8 generate: files generation

Commands to generate useful files based on project metadata: *AUTHORS*, *pyproject.toml*, *.editorconfig*, *LICENSE*, *.travis.yml*, *CONTRIBUTING.md*.

1.8.1 dephell generate authors

This command looks into your git commits history, get list of all contributors, removes duplicates by e-mail and saves it into `AUTHORS` file.

```
$ dephell generate authors
```

Easy :)

See also

1. *dephell generate license* to make license file for project.

1.8.2 dephell generate config

This command scans project directory for known dependencies files, try to combine them into known most common combinations like `Pipfile + Pipfile.lock` and makes dephell config for them.

```
$ dephell generate config
```

Good for quick start.

See also

1. *dephell inspect config* to show current config parameters.
2. *How dephell works with config and parameters*
3. *Full list of config parameters*

1.8.3 dephell generate contributing

Adds `CONTRIBUTING.md` for your project. It reads environments from the dephell config and looks for some known environments, like `pytest`, `typing`, `flake8`. For every such environment a section with instructions will be created in the output file.

See also

1. *dephell generate license* to make LICENSE file for project.
2. *dephell generate travis* to generate config for TravisCI.

1.8.4 dephell generate editorconfig

This command scans project directory for known files formats and makes .editorconfig for them.

```
$ dephell generate editorconfig
```

See also

1. *dephell generate license* to make LICENSE file for project.
2. *dephell generate authors* to make AUTHORS file for project.

1.8.5 dephell generate license

Add LICENSE file in the project. This command gets the license name as input, downloads license template, substitutes current year and current system user name and saves result into LICENSE file.

```
$ dephell generate license MIT
$ dephell generate license --owner="Your Company Name" Apache-2.0
```

Which license should I choose

For open source software use MIT License. For proprietary software pay to a lawyer to help you make right choose in your case. You can discover other licenses on the choosealicense.com, but all of them has some limitations in real world that can harm your project:

1. Apache 2.0 cool license, but requires you to insert license notice at the beginning of every source code file in the project. Also, Apache 2.0 requires all contributors to track all changes in a special file (usually named CHANGELOG). It takes some time that you can spendmore effective. There are some special tools that allow you to generate CHANGELOG and insert copyright notice in source files, but so.
2. GNU GPLv3 forbid to use your project in proprietary projects. It useful when you want to make open source only for open source, but really limit your project popularity and usage. Almost all developers writes some proprietary software, and only a bit dvelopers get paid for open source. So, don't forbid your users to use your code.
3. The Unlicense and WTFPL have limitations on usage in some countries. Don't use them.
4. Most of your users don't know about other licenses like Mozilla Public License. Don't force them to read about new licenses specially for your project. Please, value their time.

See also

1. *dephell deps licenses* to show licenses for all project dependencies.
2. *dephell generate authors* to make AUTHORS file for project.

1.8.6 dephell generate travis

Adds `.travis.yml` config for your project.

1. If your `main` env has lockfile as `to` format, DepHell adds *audit* and *outdated* checks. Also, DepHell marks they as `allow_failures` because these command can produce false-positive alerts. So, we don't want to fail whole your CI besause of it.
2. If some env has `pytest` command than this env will be ran on next envs:
    1. Linux: Python 3.5, 3.6, 3.7.
    2. Mac OS: Python 3.6.
3. If some envs has `command` specified (not `pytest`) then DepHell will make env for them too.

Of course, this file has to be manually validated and cleaned before running on CI. However, this is good bootstrap. If command doesn't work to you then use config example below to configure it on your own.

Output example:

```
# Config for Travis CI, tests powered by DepHell.
# https://travis-ci.org/
# https://github.com/dephell/dephell

language: python

before_install:
  # show a little bit more information about environment
  - sudo apt-get install -y tree
  - env
  - tree
  # install DepHell
  # https://github.com/travis-ci/travis-ci/issues/8589
  - curl https://raw.githubusercontent.com/dephell/dephell/master/install.py | /opt/
↪python/3.6/bin/python
  - dephell inspect self
install:
  - dephell venv create --env=$ENV --python="/opt/python/$TRAVIS_PYTHON_VERSION/bin/
↪python"
  - dephell deps install --env=$ENV
script:
  - dephell venv run --env=$ENV

matrix:
  allow_failures:
    - name: security
    - name: outdated

  include:
    - name: security
      install:
        - "true"
      script:
        - dephell deps audit
    - name: outdated
      install:
        - "true"
      script:
        - dephell deps outdated

    - python: "3.6"
      env: ENV=flake8

    - python: "3.6"
      env: ENV=typing

    - python: "3.5"
      env: ENV=pytest
    - python: "3.6"
      env: ENV=pytest
    - python: "3.7-dev"
      env: ENV=pytest
    - python: "pypy3.5"
```

(continues on next page)

(continued from previous page)

```
      env: ENV=pytest

    - os: osx
      language: generic
      env: ENV=pytest
      before_install:
        - curl https://raw.githubusercontent.com/dephell/dephell/master/install.py | /
↪usr/local/bin/python3
        - dephell inspect self
      install:
        - dephell venv create --env=$ENV --python=/usr/local/bin/python3
        - dephell deps install --env=$ENV
```

See also

1. *dephell generate config* to make DepHell config for project.

1.9 inspect: info about environment

Commands to get information about environment: *dephell config*, *dephell ecosystem versions*, *project metainfo*, *versioning scheme*, *virtual environment*, *stored credentials*.

1.9.1 dephell inspect auth

Shows all *added credentials*.

```
$ dephell self auth example.com gram "p@ssword"
INFO credentials added (hostname=example.com, username=gram)

$ dephell inspect auth
[
  {
    "hostname": "example.com",
    "password": "p@ssword",
    "username": "gram"
  }
]
```

Use *filters* to remove passwords from output:

```
$ dephell inspect auth --filter="#.hostname+username.each()"
[
  {
    "hostname": "example.com",
    "username": "gram"
  }
]
```

See also

1. *dephell self auth* to add new credentials.

2. *dephell inspect config* to show all other params in the config.
3. *Private PyPI repository* usage details and examples.

1.9.2 dephell inspect config

Shows current dephell config options. You can combine it with different arguments to inspect dephell behavior.

Show all config

```
$ dephell inspect config
{
  "bin": "/home/gram/.local/bin",
  "bitbucket": "https://api.bitbucket.org/2.0",
  "cache": {
    "path": "/home/gram/.local/share/dephell/cache",
    "ttl": 3600
  },
  "envs": [
    "main"
  ],
  ...
  "warehouse": "https://pypi.org/pypi/"
}
```

Filter output

Show one section:

```
$ dephell inspect config --filter=from
{
  "format": "poetry",
  "path": "pyproject.toml"
}
```

Show one value:

```
$ dephell inspect config --filter=from-format
poetry

$ dephell inspect config --filter=warehouse
https://pypi.org/pypi/
```

Combine it with arguments

```
$ dephell inspect config --from-path=lol --filter=from
{
  "format": "poetry",
  "path": "lol"
}

$ dephell inspect config --from=setup.py --filter=from
```

(continues on next page)

(continued from previous page)

```
{
  "format": "setuppy",
  "path": "setup.py"
}
```

See also

1. *dephell generate config* to initialize DepHell config for project.
2. *How dephell works with config and parameters*.
3. *Full list of config parameters*.
4. *How to filter commands JSON output*.

1.9.3 dephell inspect project

Shows metainfo from the `from` dependency file, like project name, version, required python, etc.

```
$ dephell inspect project
{
  "description": "Dependency resolution for Python",
  "links": {
    "documentation": "https://dephell.org/docs/",
    "homepage": "https://dephell.org/",
    "repository": "https://github.com/dephell/dephell"
  },
  "name": "dephell",
  "python": ">=3.5",
  "version": "0.7.8"
}
```

See also

1. *dephell project validate* to check project metadata for compatibility issues and missed information.
2. *dephell inspect config* to get information about the project configuration.

1.9.4 dephell inspect self

Shows information about DepHell installation.

```
$ dephell inspect self
{
  "path": "/home/gram/Documents/dephell/dephell",
  "python": "/usr/local/bin/python3.7",
  "version": "0.3.1"
}
```

See also

1. *dephell self uncache* to remove dephell cache.
2. *dephell self upgrade* to upgrade dephell and dependencies to the latest version.
3. *dephell self autocomplete* to enable params autocomplete for commands in your shell.
4. *dephell inspect config* to get information about config parameters.

1.9.5 dephell inspect venv

Shows information about virtual environment for current project and environment.

```
$ dephell inspect venv
{
  "activate": "/home/gram/.local/share/dephell/venvs/dephell-nLn6/main/bin/activate",
  "bin": "/home/gram/.local/share/dephell/venvs/dephell-nLn6/main/bin",
  "exists": true,
  "lib": "/home/gram/.local/share/dephell/venvs/dephell-nLn6/main/lib/python3.7/site-
↪packages",
  "lib_size": "32.93Mb",
  "project": "/home/gram/Documents/dephell",
  "python": "/home/gram/.local/share/dephell/venvs/dephell-nLn6/main/bin/python3.7",
  "venv": "/home/gram/.local/share/dephell/venvs/dephell-nLn6/main"
}
```

Specify `--env` to get information about other environment:

```
$ dephell inspect venv --env=docs
```

Specify `--filter` to get one field from command output:

```
$ dephell inspect venv --filter=project
/home/gram/Documents/dephell
```

See also

1. *dephell venv create* for information about virtual environments management in DepHell.
2. *dephell inspect config* to get information about config parameters like venv path template.
3. *How to filter commands JSON output*.

1.9.6 dephell inspect versioning

Shows info about project versioning scheme and current version.

```
$ dephell inspect versioning
{
  "rules": {
    "supported": [
      "init",
      "local",
      "major",
```

(continues on next page)

(continued from previous page)

```
      "minor",
      "patch",
      "pre",
      "premajor",
      "preminor",
      "prepatch",
      "release"
    ],
    "unsupported": [
      "dev",
      "post"
    ]
  },
  "scheme": "semver",
  "version": "0.7.9"
}
```

Read about schemes and rules in *dephell project bump* docs.

See also

1. *dephell project bump* to bump project version.
2. *dephell inspect project* to get information about the project metainfo.
3. *dephell inspect config* to get information about the project configuration.

1.10 jail: CLI tools management

Commands to manage CLI tools (like httpie) to keep them into isolated virtual environment: *install*, *show installed*, *remove*, *try some lib or tool without installation*.

1.10.1 dephell jail install

Install package into isolated virtual environment. It works similar to pipsi, but with DepHell magic:

1. Creates virtual environment named after package to install.
2. Resolves package dependencies.
3. Installs package and dependencies.
4. Creates symlinks for package entrypoints.

Like *dephell package install*, it can parse any pip-compatible input. For example:

```
$ dephell jail install isort[requirements,pipfile]
```

It will install isort with requirements.txt and Pipfile support in the isolated virtual environment named `isort`.

See also

1. *How DepHell choose Python interpreter*.

2. *dephell jail list* to show all created jails.
3. *dephell jail remove* to remove jail.
4. *dephell venv create* for information about virtual environments management in DepHell.
5. *dephell package install* to install package into project virtual environment.
6. *dephell deps install* to install all project dependencies.

1.10.2 dephell jail list

Shows a list of all packages installed by *dephell jail install* and their endpoints.

```
$ dephell jail list
{
  "flake8": [
    "flake8"
  ],
  "httpie": [
    "http"
  ]
}
```

Output can be filtered by jail name:

```
$ dephell jail list --filter=httpie
[
  "http"
]
```

See also

1. *How to filter commands JSON output*.
2. *dephell jail show* to show more info about a particular jail.
3. *dephell jail install* to create a new jail.
4. *dephell jail remove* to remove jail.

1.10.3 dephell jail remove

Removes isolated virtual environment and symlinks on it created by *dephell jail install*.

```
$ dephell jail remove isort
```

See also

1. *dephell jail install* to create a new jail.
2. *dephell jail list* to show all created jails.

1.10.4 dephell jail show

Shows information about a jail installed by *dephell jail install*.

```
$ dephell jail list
{
  "flake8": [
    "flake8"
  ],
  "httpie": [
    "http"
  ]
}
```

See also

1. *dephell jail list* to show a little bit less information but about all installed jails.
2. *dephell jail install* to create a new jail.
3. *dephell jail remove* to remove jail.

1.10.5 dephell jail try

Try python packages in an isolated environment.

Try textdistance:

```
$ dephell jail try textdistance
INFO creating venv... (python=/usr/local/bin/python3.7, venv=/tmp/tmpgixqt4_q)
INFO build dependencies graph...
INFO installation... (executable=/tmp/tmpgixqt4_q/bin/python3.7, packages=1)
Collecting textdistance==4.1.3 (from -r /tmp/tmpduyecsir/requirements.txt (line 2))
Installing collected packages: textdistance
Successfully installed textdistance-4.1.3
INFO installed
INFO running...
Python 3.7.0 (default, Dec 24 2018, 12:47:36)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

In this example DepHell installs latest `texdistance` release in a temporary virtual environment and runs python interpreter with already imported `textdistance` inside.

Use ipython instead of standard python interpreter:

```
$ dephell jail try --command=ipython textdistance
```

Set python version:

```
$ dephell jail try --python=3.5 textdistance
...
Python 3.5.3 (928a4f70d3de, Feb 08 2019, 10:42:58)
[PyPy 7.0.0 with GCC 6.2.0 20160901] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>>
```

Install flake8 and plugin for it and run checks on given path:

```
$ dephell jail try --command="flake8 ./dephell" flake8 flake8-commas
```

See also

1. *How DepHell choose Python interpreter*.
2. *dephell jail install* to install CLI tool in permanent jail.
3. *dephell venv create* for information about virtual environments management in DepHell.
4. *dephell package install* to install package into project virtual environment.
5. *dephell deps install* to install all project dependencies.

1.11 package: single package actions

Commands to work with single packages.

Get information: *download statistics*, *installed packages*, *available releases*, *package metainfo*, *search packages*.

Manage: *install*, *remove*, *remove with dependencies*, *report bug*, *verify GPG signature*.

1.11.1 dephell package bug

Report bug in a package. The command finds a bug tracker, associated with the package, and opens the tracker URL in a new browser tab.

```
$ dephell package bug flask
```

Packages in Conda also supported:

```
$ dephell package bug --repo conda textdistance
```

See also

1. *dephell package changelog* to find changelog for a package or release.
2. *dephell package show* to get information about package.
3. *dephell package search* to search packages on PyPI.

1.11.2 dephell package changelog

Find changelog for a package or release.

For package:

```
dephell package changelog pytest
```

For releases:

```
dephell package changelog pytest 3.0.7 3.0.6
```

The changelog will be printed as-is, with original grammar. So, you can redirect it into a file to render it then with an external tool:

```
dephell package changelog pytest > pytest_changelog.rst
```

The current implementation works for 80% cases. The changelog must be in one file and uploaded in the GitHub repository of the project.

See also

1. *dephell package bug* to find bugtracker for a package.
2. *dephell package show* to get information about package.
3. *dephell package search* to search packages on PyPI.

1.11.3 dephell package downloads

Get downloads statistic for package from PyPI.org. This command works with amazing PyPI Stats API, God bless this service.

```
$ dephell package downloads textdistance
{
  "pythons": [
    {
      "category": "35",
      "chart": "   ",
      "day": 120,
      "month": 3726,
      "week": 786
    },
    ...
  ],
  "systems": [
    {
      "category": "Linux",
      "chart": "   ",
      "day": 259,
      "month": 6947,
      "week": 1421
    },
    ...
  ],
  "total": {
    "day": 284,
    "month": 8751,
    "week": 1731
  }
}
```

Fields

- `pythons` – statistic by python versions.

- `systems` – statistic by operating systems.
- `total.day` – total downloads yesterday.
- `total.week` – total downloads for previous 7 days.
- `total.month` – total downloads from yesterday to the same day in the previous month.
- `pythons.#.chart` and `systems.#.chart` – downloads bar chart for last 28 days grouped by 7 days.
- `pythons.#.day` and `systems.#.day` – total downloads yesterday.
- `pythons.#.week` and `systems.#.week` – total downloads for previous 7 days.
- `pythons.#.month` and `systems.#.month` – total downloads for previous 30 days.

Filtering

This command, as all commands with JSON output, supports *filtering*. For example, get only month stat for pythons:

```
dephell package downloads textdistance --filter="pythons.#.category+month.each()"
[
  {
    "category": "27",
    "month": 332
  },
  ...
]
```

See also

1. *How to filter commands JSON output*.
2. *dephell package show* to get information about package.
3. *dephell package search* to search packages on PyPI.

1.11.4 dephell package install

Install package. See *how DepHell looks for Python environment*.

```
$ dephell package install pytest
```

Package specification the same as for pip requirements file:

```
$ dephell package install requests[security]>=2.17.0
```

See also

1. *How DepHell choose Python environment*.
2. *dephell venv create* for information about virtual environments management in DepHell.
3. *dephell deps install* to install all project dependencies.
4. *dephell jail install* to install Python CLI tools into isolated virtual environment.

1.11.5 dephell package list

Show installed packages. See *how DepHell looks for Python environment*.

```
$ dephell package list
[
  {
    "authors": [
      "Hynek Schlawack",
      "Hynek Schlawack"
    ],
    "description": "Classes Without Boilerplate",
    "installed": [
      "19.1.0"
    ],
    "latest": "19.1.0"
    "license": "MIT",
    "links": {
      "home": "https://www.attrs.org/",
      "project": "Documentation, https://www.attrs.org/"
    },
    "name": "attrs",
  },
  ...
]
```

Output of this command is really long. So, in most cases you want to *filter it*.

Show only names:

```
dephell package list --filter="#.name.sorted()"
[
  "aiofiles",
  "aiohttp",
  "appdirs",
  ...
]
```

Show only name and installed versions:

```
$ dephell package list --filter="#.name+installed.each()"
[
  {
    "installed": [
      "1.2"
    ],
    "name": "cerberus"
  },
  ...
]
```

Show name and description for first 10 packages (it can be useful for pagination by output):

```
$ dephell package list --filter="#.name+description.each().:10"
[
  {
    "description": "Lightweight, extensible schema and data validation tool for␣
↪Python dictionaries.",
```

(continues on next page)

(continued from previous page)

```
    "name": "cerberus"
  },
  ...
]
```

See also

1. *How DepHell choose Python environment*.
2. *How to filter commands JSON output*.
3. *dephell deps outdated* to show outdated packages in the virtual environment or lockfile.
4. *dephell package search* to search packages on PyPI.
5. *dephell package show* to get information about single package.
6. *dephell package install* to install package.

1.11.6 dephell package purge

Remove package with package dependencies:

```
$ dephell package purge tomlkit
```

This command removes package and package dependencies that aren't required for other packages in the environment. For example, you want to remove `pathlib2`. This package has `scandir` and `six` in the requirements. However, `six` also used in `requests`, that also installed on your system. `scandir` isn't used in another package. So, this command will remove only `pathlib2` and `scandir`. Of course, `scandir` can be used in some of your projects that isn't explicitly installed. So, if you want to avoid it and drop only package without dependencies use *dephell package remove*.

See also

1. *How DepHell choose Python environment*.
2. *dephell package remove* to remove package without dependencies.
3. *dephell package install* to install package into environment.
4. *dephell deps install* to install all project dependencies.
5. *dephell jail install* to install Python CLI tools into isolated virtual environment.

1.11.7 dephell package releases

Show available releases of package.

```
dephell package releases textdistance
[
  {
    "date": "2019-03-18",
    "python": "*",
    "version": "4.1.2"
```

(continues on next page)

(continued from previous page)

```
  },
  ...
]
```

Filter only versions:

```
dephell package releases --filter="#.version" textdistance
[
  "4.1.2",
  "4.1.1",
  "4.1.0",
  "4.0.0",
  "3.1.0",
  ...
]
```

Show 10 latest releases from git repository:

```
$ dephell package releases --filter=:10 git+https://github.com/orsinium/deal.git
↪#egg=deal
[
  {
    "date": "2018-02-04",
    "python": "*",
    "version": "1.1.0"
  },
  ...
]
```

Conda

Show releases on Anaconda Cloud:

```
$ dephell package releases --repo=conda textdistance
[
  {
    "date": "2019-03-13",
    "python": "*",
    "version": "4.1.0"
  },
  ...
]
```

See also

1. *How DepHell choose Python environment*.
2. *How to filter commands JSON output*.
3. *dephell package search* to search packages on PyPI.
4. *dephell package show* to show information about package.
5. *dephell package install* to install package.

1.11.8 dephell package remove

Remove package without package dependencies:

```
$ dephell package remove homoglyphs
```

This command doesn't remove package dependencies because we can't be sure that these dependencies aren't used anywhere in your system. For example, you want to remove `requests` that has `urllib3` in the dependencies list. But also you have somewhere on your system your personal project that depends on `urllib3` too. So, we can't sure that we can remove it. If it is OK to you use *dephell package purge*.

See also

1. *How DepHell choose Python environment*.
2. *dephell package purge* to remove package with dependencies.
3. *dephell package install* to install package into environment.
4. *dephell deps install* to install all project dependencies.
5. *dephell jail install* to install Python CLI tools into isolated virtual environment.

1.11.9 dephell package search

Search packages on PyPI or Anaconda Cloud.

Simple search by name

```
dephell package search dephell
[
  {
    "description": "Dependency resolution for Python",
    "name": "dephell",
    "url": "https://pypi.org/project/dephell/",
    "version": "0.3.1"
  },
  {
    "description": "Work with python versions",
    "name": "dephell-pythons",
    "url": "https://pypi.org/project/dephell-pythons/",
    "version": "0.1.0"
  },
  ...
]
```

Query filters

Supported query filters:

- author_email
- author
- description

- download_url
- home_page
- keywords
- license
- maintainer_email
- maintainer
- name
- platform
- summary
- version

Get all projects of author:

```
$ dephell package search author:orsinium
[
  {
    "description": "Find project modules and data files (packages and package_data␣
↪for setup.py).",
    "name": "dephell-discover",
    "url": "https://pypi.org/project/dephell-discover/",
    "version": "0.1.0"
  },
  ...
]
```

Or get first 10 packages with "environment markers" in the summary:

```
$ dephell package search --filter=":10" summary:"environment markers"
[
  {
    "description": "A compiler for PEP 345 environment markers.",
    "name": "markerlib",
    "url": "https://pypi.org/project/markerlib/",
    "version": "0.6.0"
  },
  {
    "description": "Work with environment markers (PEP-496)",
    "name": "dephell-markers",
    "url": "https://pypi.org/project/dephell-markers/",
    "version": "0.2.3"
  },
  ...
]
```

You can combine any query filters together:

```
$ dephell package search author:orsinium name:dephell
```

Anaconda Cloud

A few differences from search on PyPI:

1. Specify `--repo=conda` to search on Anaconda Cloud.
2. Search text (text without query filters) is required.
3. Available query filters:
    1. `type` (`conda`, `pypi`, `env`, `ipynb`)
    2. `platform` (`osx-32`, `osx-64`, `win-32`, `win-64`, `linux-32`, `linux-64`, `linux-armv6l`, `linux-armv7l`, `linux-ppc64le`, `noarch`)
4. Results also contain fields `links`, `license`, and `channel`.

Examples:

```
$ dephell package search --repo=conda textdistance
[
  {
    "channel": "conda-forge",
    "description": "TextDistance - python library for comparing distance between two␣
↪or more sequences by many algorithms.",
    "license": "LGPL-3.0",
    "links": {
      "anaconda": "http://anaconda.org/conda-forge/textdistance",
      "documentation": "https://pypi.org/project/textdistance/#description",
      "homepage": "https://github.com/orsinium/textdistance",
      "repository": "https://github.com/orsinium/textdistance"
    },
    "name": "textdistance",
    "version": "4.1.0"
  }
]
```

```
dephell package search --repo=conda --filter=":5" keras type:ipynb
[
  {
    "channel": "zenlambda",
    "description": "IPython notebook",
    "license": {},
    "links": {
      "anaconda": "http://anaconda.org/zenlambda/keras"
    },
    "name": "keras",
    "version": "2017.02.26.2159"
  },
  ...
]
```

See also

1. *How to filter commands JSON output*.
2. *dephell package show* to show information about single package.
3. *dephell package list* to show information about installed packages.
4. *dephell package install* to install package.

1.11.10 dephell package show

Show information about package by name.

```
$ dephell package show jsonschema
{
  "authors": [
    "Julian Berman <Julian@GrayVines.com>"
  ],
  "description": "An implementation of JSON Schema validation for Python",
  "installed": [
    "2.6.0",
    "3.0.1"
  ],
  "latest": "3.0.1",
  "license": "MIT",
  "links": {
    "homepage": "https://github.com/Julian/jsonschema",
    "package": "https://pypi.org/project/jsonschema/"
  },
  "locations": [
    "/home/gram/.local/lib/python3.7/site-packages/jsonschema",
    "/usr/local/lib/python3.7/site-packages/jsonschema"
  ],
  "name": "jsonschema",
  "size": "620.11Kb",
  "updated": "2019-03-01"
}
```

If virtual environment for current project and environment exists this command will get package version for this virtual environment. Otherwise, this command will get package versions from all paths from `sys.path` for current Python. This is the reason why `version.installed` is a list.

Conda (Anaconda Cloud and conda-forge recipes)

By default, this command uses information from PyPI. However, you can explicitly specify `--repo` to search package among conda recipes.

Search recipes in the conda-froge Github repository:

```
$ dephell package show --repo=conda_git make
{
  "authors": [],
  "description": "GNU Make is a tool which controls the generation of executables and␣
↪other non-source files of a program from the program's source files.",
  "latest": "4.2.1",
  "license": "GPLv3",
  "links": {
    "documentation": "https://www.gnu.org/software/make/manual/",
    "homepage": "https://www.gnu.org/software/make/"
  },
  "name": "make",
  "updated": "2019-01-12"
}
```

Search builded packages in Anaconda Cloud:

```
$ dephell package show --repo=conda_cloud make
{
  "authors": [],
  "description": "GNU Make is a tool which controls the generation of executables and␣
↪other non-source files of a program from the program's source files.",
  "latest": "4.2.1",
  "license": "GPLv3",
  "links": {
    "anaconda": "https://anaconda.org/conda-forge/make",
    "documentation": "https://www.gnu.org/software/make/manual/",
    "homepage": "https://www.gnu.org/software/make/",
    "source": "https://ftp.gnu.org/gnu/make/make-4.2.1.tar.bz2"
  },
  "name": "make",
  "updated": "1970-01-01"
}
```

```
$ dephell package show --repo=conda_cloud textdistance
{
  "authors": [],
  "description": "TextDistance - python library for comparing distance between two or␣
↪more sequences by many algorithms.",
  "latest": "4.1.0",
  "license": "LGPL-3.0",
  "links": {
    "anaconda": "https://anaconda.org/conda-forge/textdistance",
    "documentation": "https://pypi.org/project/textdistance/#description",
    "homepage": "https://github.com/orsinium/textdistance",
    "repository": "https://github.com/orsinium/textdistance",
    "source": "https://pypi.io/packages/source/t/textdistance/textdistance-4.1.0.tar.
↪gz"
  },
  "name": "textdistance",
  "updated": "2019-03-13"
}
```

See also

1. *How DepHell choose Python environment*.
2. *How to filter commands JSON output*.
3. *dephell package search* to search packages.
4. *dephell package list* to show information about installed packages.
5. *dephell package install* to install package.

1.11.11 dephell package verify

Verify GPG signature for a release from PyPI.

Verify files for the latest release:

```
$ dephell package verify flask
INFO getting release file... (url=https://files.pythonhosted.org/packages/.../Flask-1.
↪1.1-py2.py3-none-any.whl)
```

(continues on next page)

(continued from previous page)

```
{
  "created": "2019-07-08",
  "fingerprint": "AD253D8661D175D001F462D77A1C87E3F5BC42A8",
  "key_id": "7A1C87E3F5BC42A8",
  "name": "Flask-1.1.1-py2.py3-none-any.whl",
  "release": "1.1.1",
  "status": "signature valid",
  "username": "David Lord <davidism@gmail.com>"
}

INFO getting release file... (url=https://files.pythonhosted.org/packages/.../Flask-1.
↪1.1.tar.gz)
{
  "created": "2019-07-08",
  "fingerprint": "AD253D8661D175D001F462D77A1C87E3F5BC42A8",
  "key_id": "7A1C87E3F5BC42A8",
  "name": "Flask-1.1.1.tar.gz",
  "release": "1.1.1",
  "status": "signature valid",
  "username": "David Lord <davidism@gmail.com>"
}
```

Verify files for the given release:

```
dephell package verify django==2.0.1
```

Note that packages signing isn't popular in Python world. Most of packages have no signature:

```
$ dephell package verify pip
ERROR no signed files found
```

See also

1. *How to filter commands JSON output*.
2. *dephell package show* to show information about single package.
3. *dephell deps audit* to find known vulnerabilities in the project dependencies.

1.12 project: make releases

Commands to manage project: *run tests* into separated environment, *bump version*, *build packages*, *validate metadata*, *make the project importable*, and *upload distribution*.

1.12.1 dephell project build

Build dist packages for the project:

1. Get dependencies from `from` parameter.
2. Make `setup.py` and `README.rst`.
3. Make egg-info.

4. Make sdist (archived project source code and egg-info).
5. Make wheel.

After all you can use twine to upload it on PyPI.

Example

```
$ dephell project build --from pyproject.toml
$ twine upload dist/*
```

See also

1. *dephell deps convert* for details how DepHell converts dependencies from one format to another.
2. *dephell project bump* to bump project version.
3. *dephell project upload* to upload dist packages (on PyPI or somewhere else).
4. *dephell project test* to see how the project behaves on a clean installation.
5. *Full list of config parameters*

1.12.2 dephell project bump

Bump project version. Versioning scheme specified as `--versioning` and bumping rule or new version specified as positional argument. For example, bump minor version number by semver rules:

```
$ dephell project bump --versioning=semver minor
INFO generated new version (old=0.3.2, new=0.4.0)
INFO file bumped (path=/home/gram/Documents/dephell/dephell/__init__.py)
```

It's recommend to explicitly add `versioning` in config to let your users know which scheme you're using in your project:

```
[tool.dephell.main]
versioning = "semver"
```

If you don't have *dephell config*, you should explicitly specify `from` parameter:

```
dephell project bump --from-format=poetry --from-path=pyproject.toml minor
```

Or just add it into your config:

```
[tool.dephell.main]
from = {format = "poetry", path = "pyproject.toml"}
```

Command steps:

1. Try to detect version from `from` file.
2. Try to detect version from project source code.
3. Generate new version.
4. Write new version in source code. DepHell looks for `__version__` variable in project source and writes new version in it.

5. Write new version in `from` file.

Git tag

The command adds git tag if `--tag` option (or `tag = <your_template>` in the config) is specified as template. Template can be string with `{version}` placeholder (e.g. `v.{version}`) or just prefix string (e.g. `v.`)

```
$ dephell project bump --tag=v. minor
INFO generated new version (old=0.8.0, new=0.9.0)
INFO file bumped (path=/home/gram/Documents/dephell/dephell/__init__.py)
INFO commit and tag
INFO tag created, do not forget to push it: git push --tags
```

Rules

1. `init` – write initial version for current versioning scheme.
2. `major` or `breaking` – increment first number of version. In `pep`, `semver`, and `comver` it means breaking changes that can broke third-party code that depends on your project. Example: `1.2.3` → `2.0.0`. Zero major version has special meaning: before `1.0.0` release any increment of `minor` number can break anything.
3. `minor` or `feature` – increment second number of version. In `pep`, and `semver` it means non-breaking new features. Example: `1.2.3` → `1.2.0`.
4. `patch`, `fix` or `micro` – increment third number of version. In `pep`, and `semver` it means bugfixes that don't add new features or break anything. For `calver` it usually means hotfixes that must be delivered ASAP. Example: `1.2.3` → `1.2.4`.
5. `pre`, `rc` or `alpha` – increment pre-release number. Semantic depends on versioning scheme. A pre-release version indicates that the version is unstable and anything can be changed until release.
6. `premajor` – applies both `pre` and `major`. Example: `1.2.3` → `2.0.0-rc.1`
7. `preminor` – applies both `pre` and `minor`. Example: `1.2.3` → `1.3.0-rc.1`
8. `prepatch` – applies both `pre` and `patch`. Example: `1.2.3` → `1.2.4-rc.1`
9. `release` – removes any pre-release number. Example: `1.2.3-rc.1` → `1.2.3`
10. `post` – increment post-release number. This is supported only by `pep`. Post-release number increment means some changes that do not affect the distributed software at all. For example, correcting an error in the release notes, metainfo, including license in the package etc.
11. `dev` – increment `dev` number. Kind of pre-release that must not be used for any purposes except the project development. So, dev-releases should not be uploaded on public index servers. This version number also supported only by `pep`.
12. `local` – increment local version number. This number separated from main version by `+`. See more details in the next section.

Local number

1. Local number specified in `pep` and behave like post-release: `1.2+1` > `1.2`.
2. Be careful, local numbers compared as strings: `1.2+9` > `1.2+10`.
3. PEP recommends to use this number to indicate applying some patches to the release. For example, patch for compatibility with Ubuntu, with Django or with another project.

4. `semver` and `comver` allow to use "build metadata" with the same meaning as local number in `pep`. However, in `semver` and `comver` these metadata doesn't affect versions ordering. For example, `1.2+1 == 1.2`. For all Python projects all tools uses pep, so you shouldn't worry about it. Thence, DepHell allows you to use local number for `semver` and `comver` too.
5. Local number can contains any ASCII letters, digits and dot.
6. We recommend to use local number for nightly releases. It's like pre-release, but when you don't know version of future release and just add local number to the latest release version. See discussion in SemVer repository for more details: Nightly builds not supported
7. When you specifying rule as `local`, DepHell just increments previous local number: $1 \to 2 \to 3 \ldots$ When you want to specify exact value for local version you can pass instead of rule + sign and local version number. For example, if your current version `1.2.3+1` and you run `dephell project bump +lol` your new version will be `1.2.3+lol`.

Versioning schemes

1. pep – versioning scheme specified in PEP-420. Based on SemVer, but has much more features. All tools in Python (and DepHell too) parse projects versions by this PEP, so you can use it for your project and don't care about machines. However, this pep allows to make over-complicated versions that really difficult to understand for humans.
2. semver – most recommend versioning scheme. Allows your users (and machines) by version easily understand when you have broken something in your project, have added some new features or have fixed some bugs. If you don't know what to use, use it.
3. comver – this is semver without `patch` number. All changes that don't broke anything increments `minor` version number. You can use it if in your project it's difficult to separate bug fixes and features.
4. calver – it's when you use current date (year and month) instead of version. For example, `2018.12`. DepHell uses 4-numbers year as major number to explicitly indicate that your project uses CalVer. Also you can pass `micro` rule to add day in the version number. If previous release was today then `micro` rule will just increment this number. You can use this versioning if you don't want to care about versioning at all. However, this is strongly discouraged for any projects that can be used as dependency for third-party code.
5. romver – romantic versioning (not Sentimental Versioning, please) is when humans and marketing more important for you than machines. Bumping `major`, `minor` or `patch` number shows importance of changes and says nothing about type of this changes. Every update can break everything. As calver, never use this versioning in tools that can be used in any third-party code. But it's OK for products for users like Firefox. DepHell allows only `major`, `minor` and `patch` rules for RomVer because this versioning for humans, and humans don't understand complicated combinations of `pre`, `post` and `local`.
6. serial – this is just single number that you increment for every release (1, 2, 3...). Simplest versioning scheme, but doesn't provide any information about release changes type. Avoid this scheme if possible.
7. roman – roman numbers versioning. Never use it. It won't work after third release. However, you can try it for your internal project. Just for fun. Don't say anyone that I've recommended it to you.
8. zerover – kind of `semver`, but your `major` number always 0. Sounds like joke, but this is the best versioning for experimental projects that can broke anything in any release. So, if it's about your project then explicitly specify `zerover` versioning in your DepHell config. It says to your users that they can't upgrade without running quite strong integration tests.

Projects that use these versioning schemes

1. semver:

    - six
    - botocore
    - python-dateutil
    - requests
    - chardet
    - rsa
2. comver:
    - PyYAML
    - idna
    - terminator
3. calver:
    - pytz
    - certify
    - PyCharm
    - Ubuntu
4. romver:
    - pip (`1.5.6` → `6.0`)
    - pipenv (`0.2.8` → `3.0.0`)
5. roman:
    - Mac OS X
    - WordPerfect Office
    - 3.V album by Zebra band
    -
6. zerover:
    - html5lib 0.999999999
    - docutils 0.14 (16 years from first release)
    - pyasn1 0.4.5 (13 years from first release)
    - pandas 0.24.2 (10 years from first release)
    - colorama 0.4.1 (9 years from first release)
    - See more 0ver projects on 0ver.org.

Command examples

SemVer:

```
$ dephell project bump init
INFO generated new version (old=0.0.0, new=0.1.0)

$ dephell project bump fix
INFO generated new version (old=0.1.0, new=0.1.1)

$ dephell project bump minor
INFO generated new version (old=0.1.1, new=0.2.0)

$ dephell project bump major
INFO generated new version (old=0.2.0, new=1.0.0)

$ dephell project bump pre
INFO generated new version (old=1.0.0, new=1.0.0-rc.1)

$ dephell project bump post
ERROR ValueError: rule post is unsupported by scheme semver

$ dephell project bump local
INFO generated new version (old=1.0.0-rc.1, new=1.0.0-rc.1+1)

$ dephell project bump +ubuntu1
INFO generated new version (old=1.0.0-rc.1+1, new=1.0.0-rc.1+ubuntu1)
```

CalVer:

```
$ dephell project bump --versioning=calver init
INFO generated new version (old=1.0.0-rc.1+ubuntu1, new=2019.4)

# today
$ dephell project bump --versioning=calver micro
INFO generated new version (old=2019.4, new=2019.4.9)

# if execute `micro` again: today + 1
$ dephell project bump --versioning=calver micro
INFO generated new version (old=2019.4.9, new=2019.4.10)
```

PEP:

```
$ dephell project bump --versioning=pep init
INFO generated new version (old=2019.4.10, new=0.1.0)

$ dephell project bump --versioning=pep pre
INFO generated new version (old=0.1.0, new=0.1.0rc1)

$ dephell project bump --versioning=pep post
INFO generated new version (old=0.1.0rc1, new=0.1.0.post1)

# `dev` can be attached to `pre` or `post` too
$ dephell project bump --versioning=pep dev
INFO generated new version (old=0.1.0.post1, new=0.1.0.post1.dev1)
```

ZeroVer:

```
$ dephell project bump --versioning=zerover major
ERROR ValueError: rule major is unsupported by scheme zerover
```

(continues on next page)

(continued from previous page)

```
$ dephell project bump --versioning=zerover minor
INFO generated new version (old=0.3.2, new=0.4.0)
```

Roman:

```
$ dephell project bump --versioning=roman init
INFO generated new version (old=0.3.2, new=I)

$ dephell project bump --versioning=roman major
INFO generated new version (old=I, new=II)
```

Custom version:

```
$ dephell project bump 0.3.2
INFO generated new version (old=0.1.0.post1.dev1, new=0.3.2)
```

See also

1. *dephell inspect versioning* to get information about the project versioning scheme and available rules.
2. *dephell project build* to make release dist archives.

1.12.3 dephell project register

Register a project in the system or in a venv.

What the command does:

1. Creates egg-info in the root of the given project.
2. Creates egg-link that points on the created egg-info.
3. Creates a record in pth file (named `dephell.pth`) to make the project importable.

Note that it doesn't install any dependencies. Despite that, the command is quite similar to what `pip install -e .` and `python3 setup.py develop` do.

In a global interpreter

Register the current project in a global Python interpreter:

```
dephell project register
```

Explicitly specify python interpreter to register in:

```
dephell project register  --python=3.6
```

See also *how DepHell choice Python interpreter by default*.

In a venv

Register the current project in the `main` venv of the current project:

```
dephell project register .
```

Register another project in the `main` env of the current project:

```
dephell project register ./some/path/ /another/path/
```

Register another project in the `pytest` env of the current project:

```
dephell project register --env=pytest ./path/to/project/
```

Register the current project in a specific venv:

```
dephell project register --venv=./path/to/venv .
```

Dephell uses `from` of the current project by default. You can explicitly specify another one:

```
dephell project register --from-format=poetry --from-path=pyproject.toml ./path/to/a/
↪project/
```

See also

1. *dephell deps install* to install dependencies of a project.
2. *Python lookup* for details how you can specify Python version for commands.
3. *Full list of config parameters*

1.12.4 dephell project test

Test project package.

1. Get project dependencies from `from`.
2. Attach dependencies from `and`.
3. Build wheel package.
4. Detect pythons to run. By default, all installed pythons that supported by project. You can limit it by one python version with `--python`.
5. For every python:
    1. Make temporary virtual environment.
    2. Copy inside test files specified in `tests`.
    3. Install package from wheel.
    4. Install test command.
    5. Run command from `command` in config.

Use *dephell venv run* instead of this command if you want to run tests for current code without many pythons, temporary environments, and creating package.

Example

DepHell contains next environment in the config:

```
[tool.dephell.pytest]
# read dependencies from poetry format
from = {format = "poetry", path = "pyproject.toml"}
# copy files that requred for tests
tests = ["tests", "README.md"]
# run command `pytest`
command = "pytest -x tests/"
```

And next lines in the poetry config:

```
[tool.poetry]
name = "dephell"
version = "0.3.2"
# ...

[tool.poetry.dependencies]
python = ">=3.5"
# ...
```

You can run tests on this environment by the next command:

```
$ dephell project test --env=pytest
INFO creating wheel...
INFO get interpreters
INFO create venv (python=3.7.0)
INFO copy files (path=tests)
INFO copy files (path=README.md)
INFO install project (path=/home/gram/Documents/dephell/dist/dephell-0.3.2-py3-none-
↪any.whl)
INFO executable not found, installing (executable=pytest)
INFO run tests (command=['pytest', '-x', 'tests/'])
...
```

If you have installed python 2.7, 3.5, 3.6, and 3.7 then this command will test your code on 3.5, 3.6, and 3.7. Also, you can explicitly specify required python:

```
dephell project test --traceback --env=pytest --python=3.7
```

See also

1. *dephell venv run* to run tests for current codebase without complicated isolation.
2. *dephell deps convert* for details how DepHell converts dependencies from one format to another.
3. *Python lookup* for details how you can specify Python version for commands.
4. *Full list of config parameters*

1.12.5 dephell project upload

Upload project dist archives on PyPI or another pypi-compatible storage.

Upload on pypi.org

First of all, register on PyPI and add your credentials:

```
dephell self auth upload.pypi.org my_username my_password
```

Pro tip: add a space before the command and bash won't store it in the history (read more).

Don't forget *bump project version* and *build dist archives*. And after that you can upload created dist archives on PyPI:

```
dephell project upload
```

Upload on test.pypi.org

TestPyPI is an ephemeral instance that allows you to experiment a bit with uploading and installing of your package. Specify `upload.url` as `test` or `test.pypi.org` to use it:

```
dephell project upload --upload-url=test
```

Upload in a private repository

For example, upload on Artifactory):

```
$ dephell self auth artifactory.example.com "my-mail@example.com" "my-secret-api-key"
dephell project upload --upload-url="https://rtifactory.example.com/artifactory/api/
↪pypi/pypi-internal"
```

API Token

To keep your PyPI password secure you can generate API token in your account settings and use it instead:

```
dephell self auth upload.pypi.org "__token__" "pypi-my-secret-token"
```

This is required if you're using 2FA on PyPI.

Dist files lookup

DepHell gets project name and version from the `from` dependency file and makes on the base of it pattern to find dist files in `dist` directory in the project root. So, `from` config option is required for this command. If you have no *dephell config*, you have to explicitly specify it:

```
dephell project upload --from=setup.py
```

Or better is explicitly specify dist file:

```
$ dephell project upload --from-format=sdist --from-path=./dist/release-name.tar.gz
$ dephell project upload --from-format=wheel --from-path=./dist/release-name.whl
```

See also

1. *Configuration and parameters* to understand how DepHell configuration works.
2. *dephell project bump* to bump project version.
3. *dephell project build* to build release dist packages.
4. *dephell self auth* to provide credentials for the command.

1.12.6 dephell project validate

Validate project metadata that required to build good and compatible distribution package.

Errors

- "field is unspecified". Next fields should be specified and not empty:
  - name
  - version
  - license
  - keywords
  - classifiers
  - description
- "bad name". Project name should be normalized.
- "version should be str"
- "cannot find Python files for package"
- "short description is too long". Short description should be shorter than 140 chars.
- "short description is too short". Short description should be longer than 10 chars.
- "no authors specified"
- "no links specified"
- "no license specified in classifier"
- "no development status specified in classifier"
- "no python version specified in classifier"

Warnings

- "no dependencies found". Maybe, your project has no dependencies. Or maybe, you forgot to specify them.

See also

1. *dephell project build* to build package for the project.

1.13 self: manage dephell

Commands to manage dephell installation: *upgrade to the latest version*, *clear cache*, *enable autocomplete*, *add credentials*.

1.13.1 dephell self auth

Manage credentials: add, update, remove. These credentials are used for Basic HTTP authentication for custom PyPI repositories.

Add new credentials:

```
$ dephell self auth pypi.example.com my-useranme "my-p@ssword"
INFO credentials added (hostname=pypi.example.com, username=my-useranme)
```

Remove credentials for user:

```
$ dephell self auth pypi.example.com my-useranme
INFO credentials removed (hostname=pypi.example.com, username=my-useranme)
```

Remove credentials for all users for given hostname:

```
$ dephell self auth pypi.example.com
INFO credentials removed (hostname=pypi.example.com, count=1)
```

Credentials are stored in global config. If you add credentials for `example.com`, they will be used in all projects to connect to `example.com`.

See also

1. *Private PyPI repository* usage details and examples.
2. *dephell inspect auth* to list added credentials.
3. *dephell deps install* to install dependencies from private repository.

1.13.2 dephell self autocomplete

Enable dephell autocompletion for current shell. Supported shells:

1. bash
2. zsh

This command generates shell script to achieve better suggestions performance. So, please, execute `dephell self autocomplete` after DepHell update again.

1.13.3 dephell self uncache

Remove dephell cache.

```
$ dephell self uncache
INFO cache removed (size=64.98Mb)
```

See also

1. *dephell inspect self* to get information about dephell installation like current cache size.

1.13.4 dephell self upgrade

Upgrade dephell to the latest version. If dephell installed in venv or jail, it also updates all dependencies.

```
$ dephell self upgrade
```

See also

1. *dephell inspect self* to get information about dephell installation like current cache size.

1.14 vendor: vendorize dependencies

Vendorization is a placing external dependencies inside a folder in the project itself. It's a bad practice, but sometimes really useful and helpful. It allows you to pack dependencies inside your project to avoid conflicts with other packages, patch libraries, ship packages that aren't released on PyPI, make consistent environments etc. Use it only for internal projects and CLI tools. Never use it for libraries that used in other projects.

Some tools with vendorized dependencies: setuptools, pip, pipenv, pkg_resources.

DepHell can help you to *download and unpack* dependencies and *patch all imports*.

1.14.1 dephell vendor download

Download and extract project dependencies in a given directory.

```
dephell vendor download --from=requirements.txt --vendor-path=my_project/_vendor/
```

Some packages can be nightly and not ready for vendorization. So, you can exclude them:

```
[tool.dephell.vendorized]
from = {format = "pip", path = "requirements.txt"}

[tool.dephell.vendorized.vendor]
path = "my_project/_vendor"
exclude = ["jinja2", "setuptools"]
```

And then:

```
dephell vendor download --env=vendorized
```

How to find out packages that can't be vendorized? Do experiment:

1. `git checkout .`
2. Vendorize.
3. *Patch imports*
4. Try to run your project.
5. Have `ImportError` or `AttributeError`? Add this package into `exclude` list and try again.

See also

1. *vendor commands index* to read more about vendorization.
2. *dephell vendor import* to patch all imports in your project.

1.14.2 dephell vendor import

Patch all imports in your project and vendorized dependencies itself to use these vendorized dependencies. For example, if you're using `requests` third-party library and have `my_project/_vendor/requests` directory, run the next command:

```
dephell vendor import --vendor-path=my_project/_vendor/
```

After that all imports of `requests` inside `my_project` will be patched to import `my_project._vendor.requests` instead.

Python import system makes a big difference between packages and subpackages, and what worked in library itself can be broken when you place this library inside your project. So, be ready to exclude some libraries from vendorization. Read about it in *dephell vendor download* documentation.

See also

1. *vendor commands index* to read more about vendorization.
2. *dephell vendor download* to download your project dependencies in some directory.

1.15 venv: virtual environments

Commands to manage virtual environments for the project: *create*, and *destroy*, *activate*, *run commands inside*, *make an alias for a command*. Most important thing here is you can have as much separated environments for one project as you want. DepHell makes it really cheap.

1.15.1 dephell venv create

Create virtual environment for current project and environment. Always create virtual environment before executing *dephell deps install* or *dephell package install* if you want them to install packages into special virtual environment. Otherwise, these commands will use your current virtual environment (or global interpreter).

Path to virtual environment contains these substitutions:

- `{project}` will be replaced by the project name (name of path from `project` option, this is name of the current directory by default).
- `{digest}` will be replaced by the short 4-letters digest of the project path to avoid conflicts for the projects with the same name in different locations.
- `{env}` will be replaced by current environment (`main` by default).

So, virtual environment unique for every project and environment by default.

For example, create virtual environment for `docs` environment of current project:

```
$ dephell venv create --env=docs
```

Get venv path template with *dephell inspect config* command:

```
$ dephell inspect config --filter=venv
/home/gram/.local/share/dephell/venvs/{project}-{digest}/{env}
```

Get path to the current venv (if created) with *dephell inspect venv* command:

```
$ dephell inspect venv venv
/home/gram/.local/share/dephell/venvs/dephell-nLn6/main
```

See also

1. *How DepHell choose Python interpreter*.
2. *dephell deps install* to install project dependencies into created virtual environment.
3. *dephell package install* to install package into created virtual environment.
4. *dephell jail install* to install Python CLI tools into isolated virtual environment.
5. *dephell venv destroy* to remove virtual environment.
6. *dephell venv run* to run tool from virtual environment.
7. *dephell venv shell* to activate virtual environment for current shell.

1.15.2 dephell venv destroy

Removes virtual environment for current environment and project.

For example, destroy virtual environment for `docs` environment of current project:

```
$ dephell venv destroy --env=docs
```

1.15.3 dephell venv entrypoint

The command creates a script in user bin dir that does the next things:

1. Export env vars that specified in `vars` dictionary in the config.
2. Sources dotenv file specified in `dotenv` configuration.
3. Changes current dir for the command on the project path.
4. Runs `command` in the `venv`.

Before making the script, the command also does a few more things:

1. Creates the given `venv` if it doesn't exist yet.
2. Installs the given executable from the `command` if it's not installed in the `venv` yet.

Example

DepHell itself has the next section in `pyproject.toml`:

```
[tool.dephell.typing]
from = {format = "poetry", path = "pyproject.toml"}
command = "mypy --ignore-missing-imports --allow-redefinition dephell"
```

First of all, let's create a virtual environment and install dependencies:

```
$ dephell venv create --env=typing
$ dephell deps install --env=typing
```

And now we can expose entrypoint for the command:

```
$ dephell venv entrypoint --env=typing
WARNING executable is not found in venv, trying to install... (executable=mypy)
INFO build dependencies graph...
INFO installation... (executable=/home/gram/.local/share/dephell/venvs/dephell-nLn6/
↪typing/bin/python3.8, packages=5)
Collecting mypy-extensions==0.4.3
Collecting typed-ast==1.4.1
Collecting typing-extensions==3.7.4.1
Collecting mypy==0.770
Installing collected packages: mypy-extensions, typed-ast, typing-extensions, mypy
Successfully installed mypy-0.770 mypy-extensions-0.4.3 typed-ast-1.4.1 typing-
↪extensions-3.7.4.1
INFO installed
INFO script created (path=/home/gram/.local/bin/dephell-mypy)
```

MyPy isn't specified in `from` dependency file and wasn't installed by `dephell deps install`. So, `dephell venv entrypoint` does the installation by itself. Another thing to note is that the command has generated script name on the base of project name and the given executable (`dephell-mypy`). You can explicitly specify binary name to use:

```
dephell venv entrypoint --env=typing dephell-typing
```

Let's see what's inside the script (`cat /home/gram/.local/bin/dephell-mypy`):

```
#!/usr/bin/env bash
cd /home/gram/Documents/dephell
/home/gram/.local/share/dephell/venvs/dephell-nLn6/typing/bin/mypy --ignore-missing-
↪imports --allow-redefinition dephell $@
```

Now we can call the given script from another project without changing dirs and poking around with environments:

```
dephell-mypy --help
```

Use cases

1. Make a quick alias for a command you run really often. In the example above, we can type `dephell-mypy` instead of `dephell venv run --env=typing` and save a few precious seconds of life every day.
2. Create entrypoint to quickly get inside of project's ipython shell on your production server: `dephell venv entrypoint --command=ipython`
3. Expose your project's entrypoint from it's venv into the system. Almost the same as *dephell jail install* but for your local project.

See also

1. *dephell venv create* to create a venv.
2. *dephell deps install* to install project deps in a venv.
3. *dephell venv run* to run a command in a venv.
4. *dephell venv shell* to activate a venv for your current shell.
5. *dephell jail install* to install a third-party CLI tool into a separate venv.
6. *dephell project register* to make the project importable from anywhere in your system.

1.15.4 dephell venv run

Runs command in the virtual environment of current project and environment.

1. If the virtual environment doesn't exist DepHell will *create it*.
2. If script doesn't exist in the virtual environment DepHell tries to install it from PyPI.

For example, get help for `sphinx-build` from `docs` environment of current project:

```
$ dephell venv run --env=docs sphinx-build --help
```

Command can be specified in the config:

```
[tool.dephell.docs]
command = "sphinx-build --help"
```

In this case command can be omitted:

```
$ dephell venv run --env=docs
```

Environment variables

This command passes next environment variables into running command:

1. Your current environment variables.
2. Values from `vars` in config.
3. Values from .env file.

example of `.env` file:

```
export POSTGRES_USERNAME="dephell"
export POSTGRES_PASSWORD="PasswordExample"
export POSTGRES_URL="psql://$POSTGRES_USERNAME:$POSTGRES_PASSWORD@localhost"
```

DepHell supports any format of `.env` file: `export` word optional, quotes optional, = can be surrounded by spaces. However, we recommend to use above format, because it allows you to use source command to load these variables in your current shell.

Features for `.env` file:

1. Parameters expansion. In the example above `POSTGRES_URL` value will be expanded into `psql://dephell:PasswordExample@localhost`. If variable does not exist DepHell won't touch it. You can explicitly escape $ sign (`\$`) to avoid expansion.

2. Escape sequences. You can insert escape sequences like `\n` in values, and DepHell will process it.

Config example:

```
[tool.dephell.main]
vars = {PYTHONPATH = "."}
command = "python"

[tool.dephell.flake8]
vars = {TOXENV = "flake8"}
command = "tox"
```

Use `.env` for secret things like database credentials and `vars` in config for some environment-specific settings for running commands like environment for flake.

If you want to pass temporary variable that not intended to be stored in any file then just set this variable in your current shell:

```
$ CHECK=me dephell venv run python -c "print(__import__('os').environ['CHECK'])"
INFO running...
me
INFO command successfully completed
```

See also

1. *dephell venv shell* to activate virtual environment for your current shell.

1.15.5 dephell venv shell

Activates virtual environment of current project and environment for current shell. If virtual environment doesn't exist DepHell will create it.

Supported shells:

- cmd.exe
- PowerShell
- Bash
- Fish
- Zsh
- Xonsh
- Tcsh
- Csh

```
$ dephell venv shell --env=docs
```

This command build environment variables in the same way as *dephell venv run*.

See also

1. *dephell venv run* to run single command in a virtual environment.

1.16 Recipes and examples

There are some real-world examples, recipes and hacks how to use DepHell for different tasks.

1.16.1 Private PyPI repository

Add PyPI URL

By default, DepHell uses pypi.org as warehouse repository:

```
$ dephell inspect config --filter="warehouse"
[
  "https://pypi.org/pypi/"
]
```

You can reload it with `--warehouse` *parameter*:

```
$ dephell inspect config --warehouse example1.com example2.com --filter="warehouse"
[
  "example1.com",
  "example2.com"
]
```

You can specify it in the *DepHell config*:

```
[tool.dephell.main]
warehouse = ["example1.com", "example2.com"]
```

DepHell supports path to local directory with releases archives:

```
[tool.dephell.main]
warehouse = ["./path/to/releases"]
```

If you explicitly specify `warehouse`, DepHell drops default value and don't use pypi.org anymore. If you want to use it after your private repository, also add it in the list:

```
dephell inspect config --warehouse https://example1.com/simple https://pypi.org/ --
↪filter="warehouse"
[
  "https://example1.com/simple",
  "https://pypi.org/"
]
```

You can remove any repositories at all to use only specified in dependencies file:

```
$ dephell inspect config --warehouse --filter="warehouse"
[]
```

Authentication

Use *dephell self auth* to add credentials for host in global config:

```
$ dephell self auth example.com gram "p@ssword"
INFO credentials added (hostname=example.com, username=gram)
```

You can list stored credentials with *dephell inspect auth*:

```
$ dephell inspect auth
[
  {
    "hostname": "example.com",
    "password": "p@ssword",
    "username": "gram"
  }
]
```

Dependency file

Some dependency formats support explicit repository specification. These repositories always have higher priority than specified in config.

`requirements.txt`:

```
-i https://example.com/
-i https://pypi.org/simple/
...
```

`Pipfile`:

```
[[source]]
url = "https://example.com/"
verify_ssl = true
name = "example"

[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

# ...

[packages]
deal = {index="example"}
# ^ try to find in "example" repository before all
```

Poetry (`pyproject.toml`)

```
# ...

[[tool.poetry.source]]
name = "example"
url = "https://example.com/"

[[tool.poetry.source]]
name = "pypi"
url = "https://pypi.org/simple"
```

1.16.2 Pre-commit hook for git

Pre-commit allows you to do some action before every commit. In these examples, you can generate `requirements.txt` and `setup.py` from `poetry`.

Without DepHell config

Add next lines in .pre-commit-config.yaml:

```
- repo: https://github.com/mverteuil/precommit-dephell
  rev: master
  hooks:
    - id: pyproject-toml-to-setup-py
    - id: pyproject-toml-to-requirements-txt
```

With DepHell config

If you have *dephell config* you can make the same things quite easier.

Add next lines in .pre-commit-config.yaml:

```
- repo: https://github.com/mverteuil/precommit-dephell
  rev: master
  hooks:
    - id: dephell
```

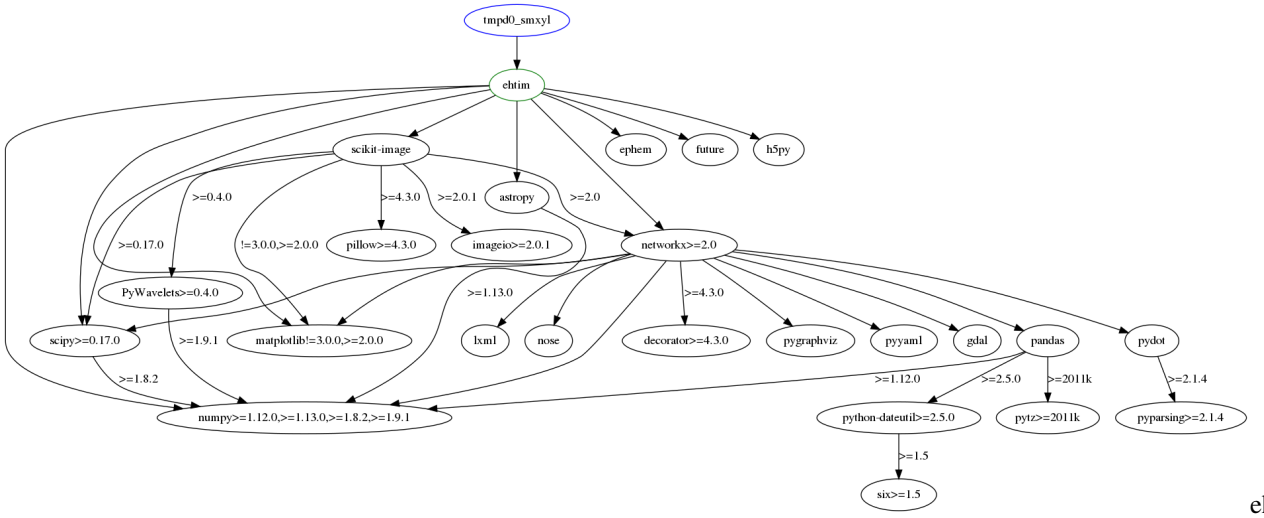
See also

1. Source repository.
2. *DepHell config*.
3. Hooks in Git.

1.16.3 Make dependencies graph for git repo

Let's make dependencies graph for ehtim. This is software that used to take a photo of black hole. We can make this graph with *dephell deps tree* command:

```
dephell deps tree --type=graph git+https://github.com/achael/eht-imaging.git#egg=ehtim
```



ehtim dependencies graph

1.16.4 Lock poetry dependencies

Let's lock poetry dependencies for dephell. First of all, get dephell source code:

```
git clone https://github.com/dephell/dephell.git
cd dephell
```

With CLI arguments

```
$ dephell deps convert --from=pyproject.toml --to=poetry.lock
```

With config

Add this in your `pyproject.toml`:

```
[tool.dephell.main]
from = {format = "poetry", path = "pyproject.toml"}
to = {format = "poetrylock", path = "poetry.lock"}
```

And then run:

```
$ dephell deps convert --config=pyproject.toml --env=main
```

Dephell by default uses `pyproject.toml` config and `main` section, so you can run it much simpler:

```
$ dephell deps convert
```

See also

1. *How dephell works with config and parameters*
2. *dephell deps convert command*

1.16.5 Show licenses for your dependencies

You can use *dephell deps licenses* command to reveal licenses for all dependencies (and dependencies of dependencies) of your project. For example, let's get licenses for flake8 plugins of dephell. First of all, get dephell source code:

```
git clone https://github.com/dephell/dephell.git
cd dephell
```

With CLI arguments

```
$ dephell deps licenses --from-format=pip --from-path=requirements-flake.txt

INFO resolved
{
  "BSD-2-Clause": [
    "enum34"
  ],
```

(continues on next page)

(continued from previous page)

```
  ...
  "Unknown": [
    "flake8-logging-format"
  ]
}
```

With config

Dephell contains this section in the `pyproject.toml`:

```
[tool.dephell.flake8]
from = {format = "pip", path = "requirements-flake.txt"}
```

So, you can write command above quite shorter:

```
$ dephell deps licenses --env=flake8
```

See also

1. *How dephell works with config and parameters*.
2. *dephell deps licenses* documentation.
3. *dephell generate license* to make license for project.

1.16.6 DepHell-powered projects

Need more inspiration? Check out real use cases.

50-100 stars

- deal – Design by contract for Python with many validators support.
- lightbus – RPC & event framework for Python 3.
- scdlbot – Telegram Bot for downloading MP3 rips of tracks/sets from SoundCloud, Bandcamp, YouTube with tags and artwork.

10-50 stars

- abilian-core – Abilian Core framework and services.
- abilian-sbe – Abilian Social Business Engine - an enterprise social networking / collaboration platform.
- aepp-sdk-python – Python SDK for the Æternity blockchain.
- fastapi-jsonrpc – JSON-RPC server based on fastapi.
- flakehell – Flake8 wrapper to make it nice, legacy-friendly, configurable.
- homoglyphs – get similar letters, convert to ASCII, detect possible languages and UTF-8 group.
- jazzband-roadies – Code for the Jazzband website.

- micropy-cli – Micropython Project Management Tool with VSCode support, Linting, Intellisense, Dependency Management, and more!
- plexdl – A plex direct downloader. The whole point is to get media that has not been modified.
- python-open-controls – Q-CTRL Open Controls.
- runrestic – A wrapper script for Restic backup software that inits, creates, prunes and checks backups.

1.17 Badge

If you want to show your users how to use your project more effective and what `dephell.tool.main` in your `pyproject.toml` means just add badge in your readme:

Powered by DepHell

```
[![Powered by DepHell](https://github.com/dephell/dephell/blob/master/assets/badge.
↪svg)](https://github.com/dephell/dephell)
```

1.18 CHANGELOG

Follow @PythonDepHell on Twitter to get updates about new features and releases.

1.18.1 v.0.8.3 (2020-04-28)

This release brings a lot of small fixes. See the milestone for the details. The only noticeable change is an ability to provide a custom CA bundle via `--ca` flag. A few next releases also will bring more stability and speed into DepHell. Stay tuned!

1.18.2 v.0.8.2 (2020-03-19)

New commands:

- dephell project upload (#401).
- dephell venv entrypoint (#404).
- dephell package verify (#400).
- dephell project register (#398).

Improvements:

- `dephell project bump` now can bump version in Sphinx config (#407).
- One less dependency. Bye-bye, flatdict (#394).
- A lot of `dephell jail list` improvements (#381, #379, #395).
- `--version` and `-h` flags (#397, #410).
- More info in `dephell inspect venv` (#396).
- `dephell jail` and some similar commands don't try to find project-level config (#408).
- Better `requirements.txt` lookup (#409).

- a little bit more.

1.18.3 v.0.8.1 (2020-01-27)

New commands:

- dephell package changelog (#361).

Improvements:

- Experimental Windows support. Test it and contribute! (#343). Special thanks to @espdev who has done almost all the migration.
- Lazy imports. About 10 dependencies were converted into optional and will be installed by-demand. Installation with `curl -L dephell.org/install | python3` isn't affected. (#349)
- Support new pip (362) and lock older pip because a new one is broken (#363)
- Rewritten installer (#365, #355)
- a little bit more.

1.18.4 v.0.8.0 (2019-12-19)

New commands:

- dephell package bug (#318).
- dephell jail show (#318).
- dephell inspect versioning (#318).

Improvements:

- Meet dephell_argparse (#317).
- Meet DepHell-powered projects list (#339)
- Rename `dephell autocomplete` into `dephell self autocomplete`, and `dephell auth` into `dephell self auth` (#321).
- Support `allow-prereleases` key from Poetry 1.0.0 (#323)
- From now DepHell will not be tested on Python 3.5 installation because nobody installs DepHell on Python 3.5 (#334).
- a little bit more.

1.18.5 v.0.7.9 (2019-11-19)

New commands:

- *dephell self uncache* (#312).
- *dephell self upgrade* (#311).
- *dephell generate contributing* (#255).
- *dephell inspect project* (#296).
- *dephell project validate* (#310).

Improvements:

- Smart `setup.py` parsing. Meet dephell_setuptools (#308).
- Stable `setup.py` generation (#292).
- Cleaner sdist (#297).
- a little bit more

1.18.6 v.0.7.8 (2019-10-22)

- Fuzzy command name search (#247, #122).
- *Configure* DepHell with environment variables (#248).
- Colored JSON output (#262, #260, #205).
- Table output with `--table` (#277, #267, #206).
- New attrs (#261).
- ruamel.yaml instead of pyyaml (#275)
- pip 19.3.1 support (#276).
- a little bit more

1.18.7 v.0.7.7 (2019-07-23)

- Meet dephell.org (#244).
- Lazy dependencies overwriting (#232, #229).
- Removed Snyk support (#245).
- Added custom User-Agent to all requests (#242, #243, #231)
- Updated documentation interface (#241).
- `path` support for `pip`, `pipenv`, `poetry` (#230, #227).

1.18.8 v.0.7.6 (2019-07-17)

- Docker support (#220, #49).
- Fixed dependencies for DepHell itself (#218, #216).
- Resolve paths to dependency files relatively to the project, and local dependencies relatively to the dependency file (#217, #88).
- Fixed repositories dumping for poetry (#215, #177).
- Simplified "usage" for commands' help (#212, #120).
- Install extras in *dephell project test* if needed (#204, #195).

1.18.9 v.0.7.5 (2019-07-07)

- Vendorization (*dephell vendor download* and *dephell vendor import*) (#194, #109)
- Now CLI for some commands accepts `--from` instead of `--to`, because it makes much more sense (#194, #138)
- Always PEP-compatible name for names of wheel and sdist (#203, #192)
- Now `--tag` option for *dephell project bump* allows to specify tag prefix or template (#199, #197)
- Meet dephell_versioning, our new friend to handle packages versioning (#191, #147)
- Shorter links in documentation (#183, #182)

1.18.10 v.0.7.4 (2019-06-17)

- Custom warehouse and simple index support (#53, #128).
- Fixed bug with packages names that made them incompatible with `pkg_resources` (#110, #117).
- Now `project bump` doesn't make git tag by default. Use `--tag` to add tag or add `tag = true` into config (#114, #106).
- Support for output into stdout for *dephell deps convert* (#140, #136).
- Allow to install prereleases into jail (#118, #113)
- Smarter detection of owner name for `generate license`. You can force the name with `--owner=Name` (or `owner = "Name"` in config) (#108, #107, #104, #87).
- Local filesystem path support for `--warehouse` parameter (#145).
- Improved documentation (#162).
- Improved pre-release support for *dephell project bump* (#144, #142).
- Improved poetry support (#159, #152, #154).
- Lazy load for bash autocomplete (#132).

1.18.11 v.0.7.3 (2019-05-19)

- Added `imports` converter to get dependencies from package imports (#97).
- `sdist` includes tests if they not too big (`--sdist-ratio` option) (#99, #95).
- You can specify path to `.env` file (#69, #100).
- `dephell package list` doesn't fail if some packages missed on PyPI (#85, #102).

1.18.12 v.0.7.2 (2019-05-19)

- flit support.
- Missed meta information (like project version when you read from `requirements.txt`) will be automatically parsed from magic variables (like `__version__`) in the project source code.
- Fix `plugins` parsing in poetry and `extras` parsing for `egg-info` and `sdist` (#86, #89).
- Fix `sdist` structure (#94, #93).

1.18.13 v.0.7.1 (2019-05-12)

- `dependency_links` support for `setup.py`, `sdist` and `wheel` (#79, #63).
- Python 3.8 support (#78).
- Fix autocomplete for Mac OS X (#65, #62).
- Preserve dots in packages names (#71, #80, pypa/pip#3666).
- Make autocomplete for zsh really cool: added support for paths and choices (#81).

1.18.14 v.0.7.0 (2019-05-05)

- Filter dependencies by envs (#56, #58).
- Change API: now all import must be from the second level. For example, `from dephell.models import Dependency` instead of `from dephell import Dependency` or `from dephell.models.dependency import Dependency`.
- Support for `allow-prereleases`, `python` and `platform` options in poetry (#59).
- Serial versioning support (#60).

1.18.15 v.0.6.0 (2019-04-30)

- Conda support (#48).
  - Anaconda Cloud.
  - Recipes from Github for conda-forge and bioconda.
  - Support in `project show`, `project search` and `project releases`.
  - Converter for environment.yml.
- Do not write hashes in `piplock` when some dependencies is local (#41, #47).
- Do not mess up setup.py on `project bump` (#46).

1.18.16 v.0.5.8 (2019-04-25)

- Fix some typos (#43, #40).
- Fix autocomplete when data directory wasn't created (#42).

1.18.17 Before

- The first public release: 2019-03-14.
- The first proof-of-concept: 2018-09-03.

1.19 Packaging issues

My favorite issues collection.

1.19.1 Pip

1. Pip doesn't support Pipfile (pipfile#80).
2. Pip doesn't have dependency resolution (pip#988)

1.19.2 Pipenv

1. Pipenv doesn't support more than 2 environments for project (pipfile#99).
2. Pipenv doesn't support "allow pre-releases" option for single dependency (pipenv#1760).
3. Pipenv doesn't support python version range (pipfile#87).
4. Pipenv doesn't support local packages installation like `--find-links` in `pip` or `dependency_links` in `setup.py` do (pipenv#2231)

1.19.3 Poetry

1. Poetry supports only `platform` and `python` specification. This is not documented (poetry#738). This doesn't allow you to specify other markers like python implementation (poetry#21).
2. Poetry doesn't allow you to specify editable dependencies in the config (poetry#34).

1.19.4 PyPI

1. PyPI API doesn't provide dependecies list for some packages (packaging-problems#54 and warehouse#789).