pkglts Documentation

Release 2.1.0

revesansparole

Dec 07, 2017

Contents

Contents:

CHAPTER 1

pkglts

Building packages with long term support

The rationale behind the creation of this 'package builder' is to keep the life of a python programmer as easy as possible by providing three core functions:

- A way to add more functionality to an existing package.
- A way to keep the package structure up to date with currently known best practices.
- Remove repetitive tasks that can be automated from the list of things to do.

1.1 Quick start

Create a virtual environment for development:

```
$ virtualenv dvlpt
```

Activate it:

```
$ (on windows)dvlpt\Scripts\activate
$ (on linux)dvlpt/bin/activate
```

Install pkglts:

```
(dvlpt)$ pip install pkglts
```

Create a directory for your package:

```
(dvlpt)$ mkdir toto
```

Run 'manage' inside this directory:

```
(dvlpt)$ cd toto
(dvlpt)toto$ pmg init
(dvlpt)toto$ pmg add base
(dvlpt)toto$ pmg regenerate
```

This will create the bare basic minimum for a python package. Add more options (see the add_option for more options) afterward. Especially, since in the example above we just added the 'base' option that will create a 'src' directory to put your code in it.

1.2 Documentation

More documentation can be found on readthedocs_pkglts. If you just intend to use this package you can start with some tutorials. However, if the core functionality are not sufficient and you want to be part of the development you might be interested with the developer section of the doc.

1.3 Upgrade Package Structure

Packages generated with pkglts contains three different types of files:

- Configuration files for pkglts, which are all located inside a '.pkglts' directory at the root of your package. This directory especially contains a 'pkg_cfg.json' resource file that contains all the information necessary to regenerate your package in order to take into account new developments in the way of packaging or new options you added to your package.
- Files that contain at least one 'pkglts' tag that have been automatically generated by pkglts. Everything inside a pkglts section is regenerated every time the user call 'regenerate' on his package and are not meant to be modified. Between pkglts sections the user is free to add his own code that won't be touched in the regenerate process.
- developer data and modules edited by hand which contains the actual python code of the package independently of the structure of the package. pkglts will never touch them. If they conflict with some files used by a newly added option, the user will be prompted and will have to solve the conflict in order to install the option.

Every time you make changes to the structure of your package by adding a new option for example, a call to 'pmg regenerate' is mandatory to rebuilt the package structural files:

```
(dvlpt)toto$ pmg regenerate
```

This phase will never overwrite any files you modified or created. You'll be prompted in case of conflicts but it is your responsibility to solve them and relaunch the command.

It is also good practice to regenerate your package from time to time to take into account the latest version of pkglts which will reflect the latest development in the way to package things.

1.4 Add Package Structure Functionality

Pkglts provides a set of options to introduce new functionality to an already existing package:

- *base*: base option, basic package management
- *license*: will help the developer to choose a license and add the relevant files
- *doc*: Add some documentation to your package

- *test*: basic unitests using Nose
- *coverage*: add code coverage to the basic test configuration
- *pysetup*: make your package distributable with setuptools (i.e. setup.py)
- *data*: will guide through all the steps to add non python files to a package
- *github*: will guide through all the step to safely store the package on Github
- *gitlab*: will guide through all the step to safely store the package on a Gitlab repo
- *readthedocs*: step by step guide to have your documentation on ReadTheDocs
- *travis*: will guide through all the steps to use Travis-CI for continuous integration of your package
- *tox*: defines config files to use multi environment testing, Tox
- *flake8*: install and config Flake8 tools to check for code compliance to PEP8 guidelines.
- *pypi*: step by step guide and configuration to upload package on PyPi.

1.4.1 Install a new option

To install a new option call the 'add' action:

```
(dvlpt)toto$ pmg add license
```

The script will perform different tasks sequentially:

- Check if this option requires other options in order to be installed: e.g. the 'pysetup' option requires all 'base, 'doc', 'test', 'license' and 'version' in order to run properly.
- Check if this option requires some extra packages in order to setup: e.g. the 'test' option depends on the Nose package to function properly.

Note: Nothing will be installed without your consent

Multiple call to add options can be serialized but you explicitly needs to call regenerate to see the action of the new options on your package:

```
(dvlpt)toto$ pmg regenerate
```

Before calling 'regenerate' however, take the time to browse through 'pkg_cfg.json' in the '.pkglts' directory to edit the parameters relevant for your option.

1.4.2 Install example files

Some options come with example files that can be installed with the special directive:

```
(dvlpt)toto$ pmg example test
```

The files will be directly installed without the need to a regenerate call. You can reinstall them at any time (you'll be prompted for action if conflicts occur).

1.5 Extra services

Warning: Work In Progress

Package Builder also provides a few useful services to check that the python modules follow code best practices:

- 'add_object': will create a new python module with the proper headers and a skeleton of a python class.
- 'add_plugin': will wrap a given python class into a usable plugin.
- 'add_script': will wrap a given python functionality into a command line script.
- 'reset_file_header': will loop through all python modules and try to rewrite file header to match current best practices.
- fmt_doc: check code documentation and format it according to given standard if possible. Requires some already good documentation, just a quick fix to pass from one style to another (e.g. google to numpy).

1.6 Contributing

You can contribute to this package by:

- improving the documentation
- correcting some bugs
- closing a few issues
- implementing a new option to add a new functionality to package structures

CHAPTER 2

Installation

Download source then, at the command line:

```
$ python setup.py
```

Alternatively the command line:

```
$ easy_install pkglts
```

Or:

```
$ pip install pkglts
```

Preferred method use virtual environments:

```
$ virtualenv 'myenv'
$ myenv/scripts/activate
(myenv)$ pip install pkglts
```

or conda environments:

```
$ conda create -n toto python
$ activate toto
(toto) $ pip install pkglts
```

CHAPTER 3

Usage

pkglts is intended to be used every time you want to create a package and maintain it throughout time despite changes in the different services you use to test and distribute your package.

It has been though so you just have to regenerate your package everytime one of the tools you use changes its interface (e.g. change some option in the yaml config file associated with the tool).

The sequence is always the same:

- initialize your package in an empty directory
- add options that represent the tools you intend to use
- change settings for these options in the pkg_cfg file
- regenerate your package

Then you can happily write some code to actually create functionalities in your package knowing you won't have to care about continuous integration, testing or deployment :)

If you find yourself always using the same options with the same parameters everytime you create a new package you can consider writing your very own plugin to smoothen your life. Then creating a new package will be as simple as typing:

```
$ pmg init my_plugin_name
$ pmg rg
```

CHAPTER 4

Tutorials

These tutorials will guide you through all the steps of setting up a development environment and creating a package.

4.1 Setting up your development environment

This tutorial will guide you through all the steps of setting up a development environment on your machine. The use of virtualenv is not mandatory but we strongly encourage you to use it or something similar in order to simply wipe out any mistake you can make in the process of creating a package.

Install python (multiple versions?)

Create virtual env

Install pkglts that will install all its dependencies

4.1.1 To go further

See *Creating a distributable package*.

4.2 Creating a distributable package

This tutorial will guide you through all the steps of creating a package from scratch all the way to uploading it on PyPi.

4.2.1 Setting up your development environment

I will assume you already have a development environment running with pkglts installed. If not check :doc:dvlpt_env.

In the following, I will assume you have created a 'dvlpt' virtual environment and that you activated it. Just forget about it if you are not using virtual environments.

4.2.2 Create a new basic package

Create package

add options base, test, doc, version, license

4.2.3 Make a distribution of your package

Use of pysetup option

4.2.4 Check your code thoroughly

add options coverage, flake8, tox

4.2.5 Make your package available to the community

add pypi option

4.2.6 To go further

See *Making use of web tools*.

4.3 Migrate an already existing package to use pkglts

Ok, you already have a well defined package running but you are interested in using pkglts to add new features in the future.

Note: You don't have to break anything

The following steps will guide you through the process of adding pkglts management to your existing package with minimal troubles.

4.4 Making use of web tools

This tutorial will help you figure out how to add some interesting tools to help you make best use of commonly available tools on internet.

This tutorial assume you already have a Pypi ready package. If not, check the previous tutorials (:doc:pypi).

In the following, I will assume you have created a 'dvlpt' virtual environment and that you activated it. Just forget about it if you are not using virtual environments.

4.4.1 Travis-CT

4.4.2 ReadTheDocs

4.4.3 Landscape.io

4.5 Single option tutorials

These tutorials are specific to a single option.

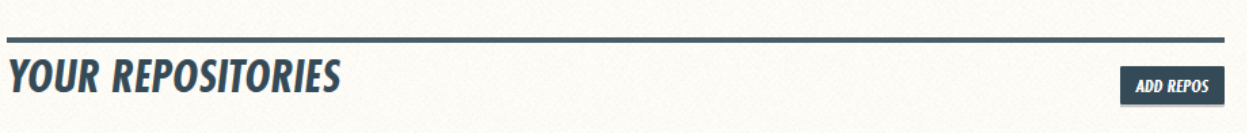
4.5.1 Register on Coveralls.io

As always, you need to register on Coveralls_io. This process is made easy since you can used you GitHub id to sign up.

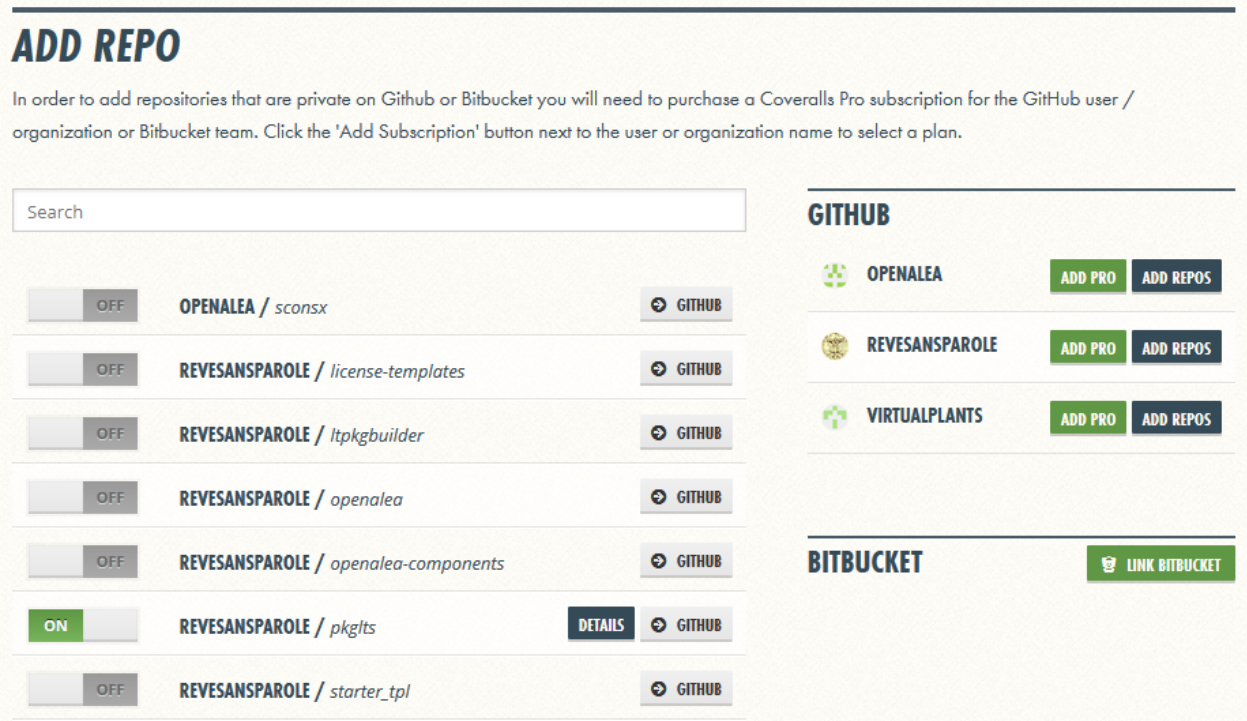
You must arrive on the main page of Coveralls_io with a small set of instructions to add a repository.

Add your project

Registering a project is easy and you just need to click on the 'add repos' button.



You must see a list of repositories you collaborate with. If yours is not visible maybe hit the 're-sync repos' button (you may also have to refresh your browser at some point to see the changes).

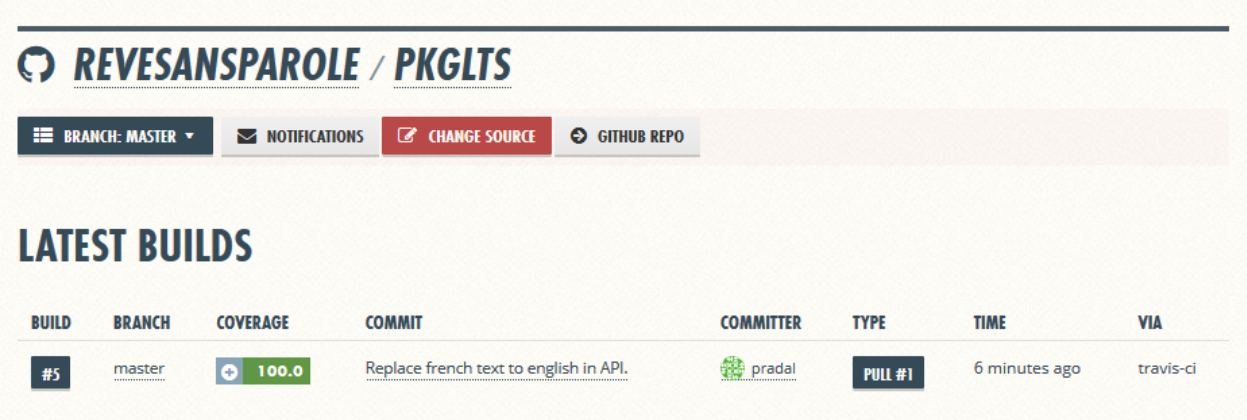


Flip the switch for your project and click on the details button.

Coveralls_io use Travis-CI to gather information on your project, so you need to trigger a new build if you want to see some result, either:

```
$ git push
```

or manually click the rebuild button on Travis-CI. After the build has finished and coveralls had time to gather information, if you refresh the 'coveralls' web page you must see the statistics on your code.



Final remark

If everything is successful, you must now have a coverage-100% green badge that show on top of your readme in the homepage of your project on github (hit refresh if you see nothing, you may also have to click on the badge urls button on landscape).

If you want more statistics you can always look at the details of the latest build on landscape but they are mostly the same information available with a local call to 'coverage' :)

4.5.2 Add data tutorial

Regenerating your package after adding the option 'data' will create a 'pkgname_data' in the 'src' directory:

```
(dvlpt)$ pmg add data
(dvlpt)$ pmg regenerate
```

Just copy all your data inside this directory and they will be packaged and installed along your package whatever method you choose to distribute your package:

```
(dvlpt)$ python setup.py sdist
(dvlpt)$ python setup.py install
(dvlpt)$ python setup.py develop
(dvlpt)$ python setup.py bdist_egg
```

Accessing data from inside the code

The recommended way to access data would be to use package_resources. However this method fail. Depending on the way you distribute your code, the path to your data will be changed. We choose a simple rough approach and, along the package_data directory, the 'data' option also create a 'data_access.py' file in your sources. To access 'data' you just need to call the 'get_data_dir' function and you'll be returned with a valid pth to the package_data directory.

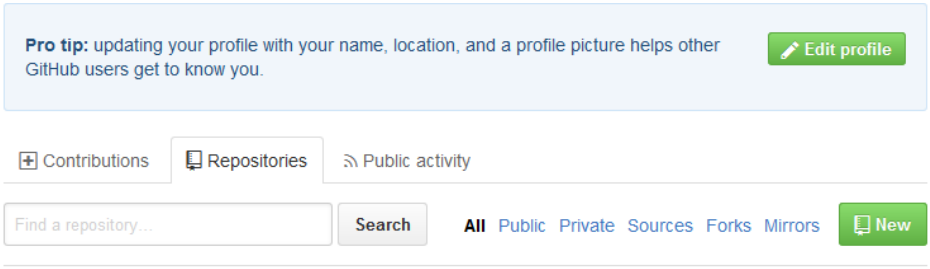
4.5.3 Register on Github

First step, you need to create yourself an account on GitHub.

In the following I will assume that you now have a login and that you are logged on.

Personal package

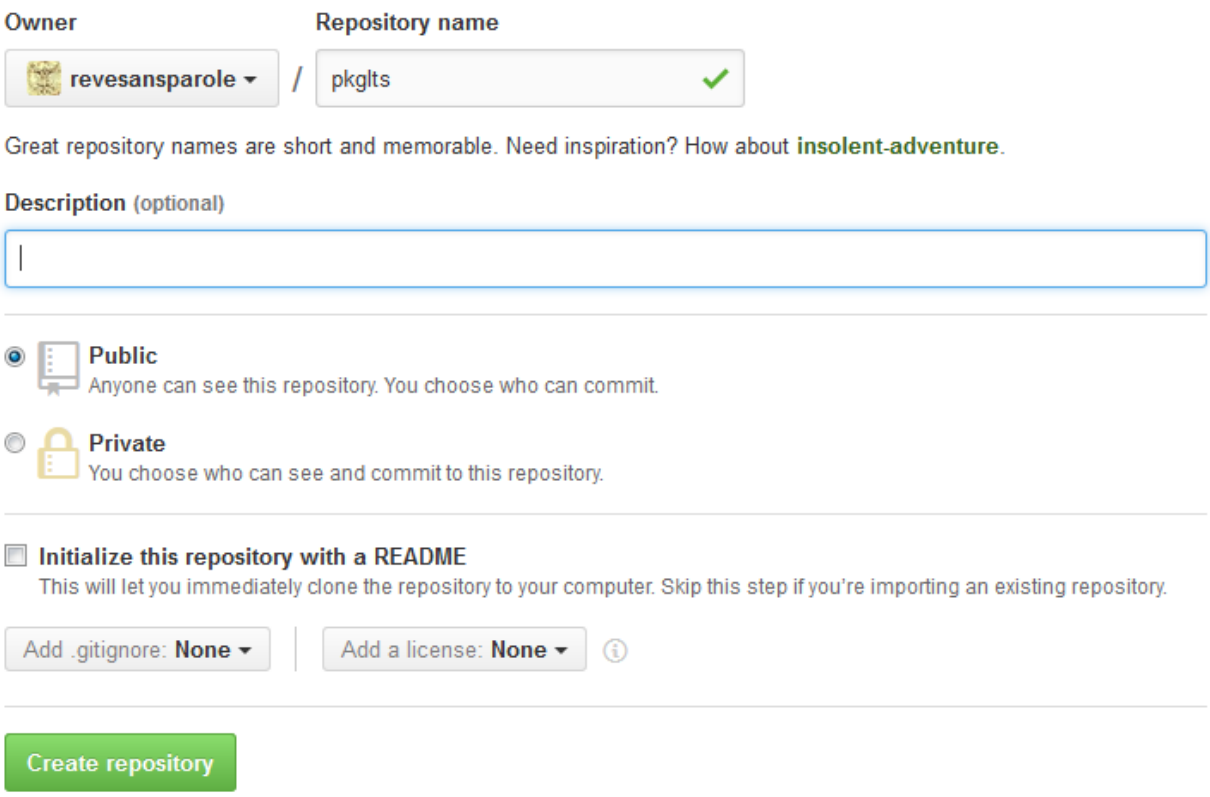
On your personal web page 'https://github.com/your_login_here', on the 'repository' tab there is a button 'New'.



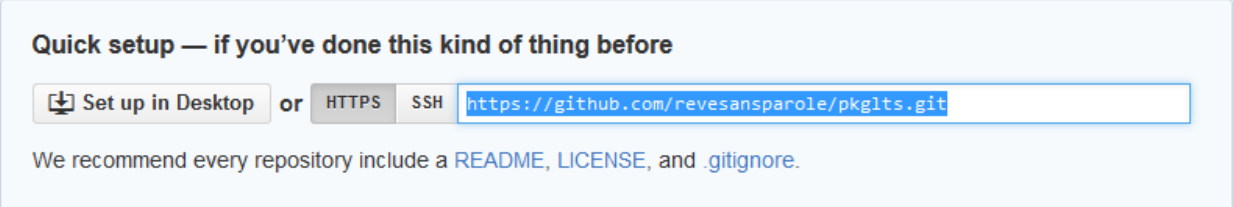
Click it and enter the name of the project (same name used when adding the 'github' option, you can always edit the option again to change it if you want).

You can leave the description blank and we suggest to keep it public.

Don't initialize this repository with a README since we already have one to commit. Leave the 'add .gitignore' and 'add license' to None. And just click 'create repository'.



Felicitations, you just created the package on GitHub. Take a mental note of the url they provide since it will be used later.



To populate it you'll need to use the 'git' tool as explained briefly in *Quick Git Tutorial*.

Organization package

May mess up with current information storage

Warning: TODO

Quick Git Tutorial

If you are not familiar with Git, now is the time to read some tutorials. The following will assume you have a basic understanding of Git and will just explained what needs to be done to first push your package on the newly created project.

In the root directory of your project:

```
$ git init
$ git status
```

If you have added the github option already and regenerated the package, a '.gitignore' has been created so the 'status' command should not show any temporary files like 'build', '.coverage', '.tox' for example. Therefore you can safely add all:

```
$ git add --all
```

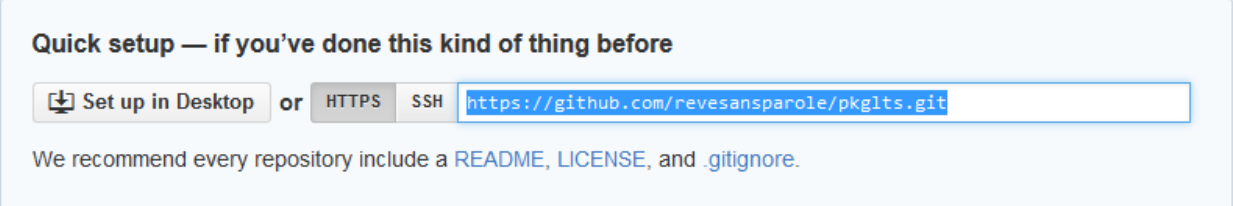
and make your first commit:

```
$ git commit -m"initial commit :)"
```

You then need to create a remote to push your code on the github repository:

```
$ git remote add origin url_provided_on_github
```

The url to provide is the one provided once you successfully created your project.



Now you can push your local copy on the master branch on github:

```
$ git push --set-upstream origin master
```

You'll be prompted for your github credential and upon success you can check on the github page of your project that there is some sources.

Git usage

The github option is responsible for maintaining a proper '.gitignore'. Therefore you can safely assume that no unwanted files will be covered when you 'add –all' which makes one life easier to add new modules in a package.

4.5.4 Register on Gitlab

First step, you need to create yourself an account on Gitlab.

In the following I will assume that you now have a login and that you are logged on.

Personal package

On your personal web page 'https://framagit.org/your_login_here', on the 'repository' tab there is a button 'New'.

Click it and enter the name of the project (same name used when adding the 'gitlab' option, you can always edit the option again to change it if you want).

You can leave the description blank and we suggest to keep it public.

Don't initialize this repository with a README since we already have one to commit. Leave the 'add .gitignore' and 'add license' to None. And just click 'create repository'.

Felicitations, you just created the package on Gitlab. Take a mental note of the url they provide since it will be used later.

To populate it you'll need to use the 'git' tool as explained briefly in *Quick Git Tutorial*.

Organization package

May mess up with current information storage

Quick Git Tutorial

If you are not familiar with Git, now is the time to read some tutorials. The following will assume you have a basic understanding of Git and will just explained what needs to be done to first push your package on the newly created project.

In the root directory of your project:

```
$ git init
$ git status
```

If you have added the gitlab option already and regenerated the package, a '.gitignore' has been created so the 'status' command should not show any temporary files like 'build', '.coverage', '.tox' for example. Therefore you can safely add all:

```
$ git add --all
```

and make your first commit:

```
$ git commit -m"initial commit :)"
```

You then need to create a remote to push your code on the gitlab repository:

```
$ git remote add origin url_provided_on_gitlab
```

The url to provide is the one provided once you successfully created your project.

Now you can push your local copy on the master branch on gitlab:

```
$ git push --set-upstream origin master
```

You'll be prompted for your gitlab credential and upon success you can check on the gitlab page of your project that there is some sources.

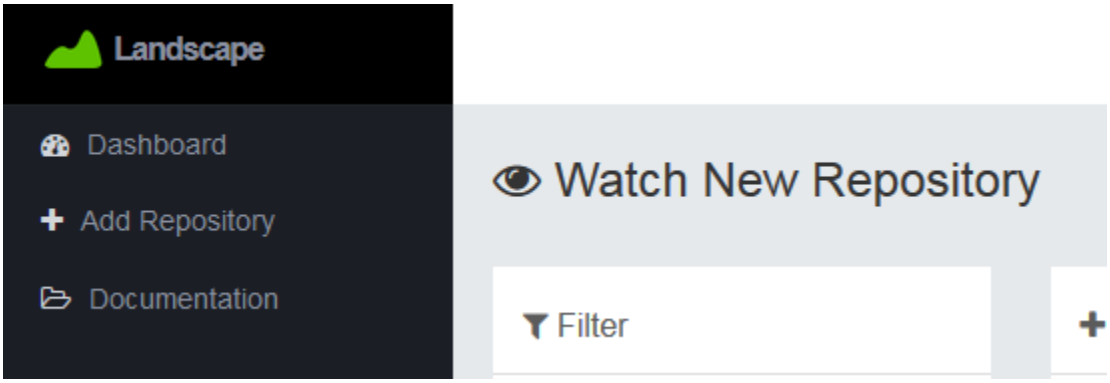
Git usage

The gitlab option is responsible for maintaining a proper '.gitignore'. Therefore you can safely assume that no unwanted files will be covered when you 'add –all' which makes one life easier to add new modules in a package.

4.5.5 Register on Landscape.io

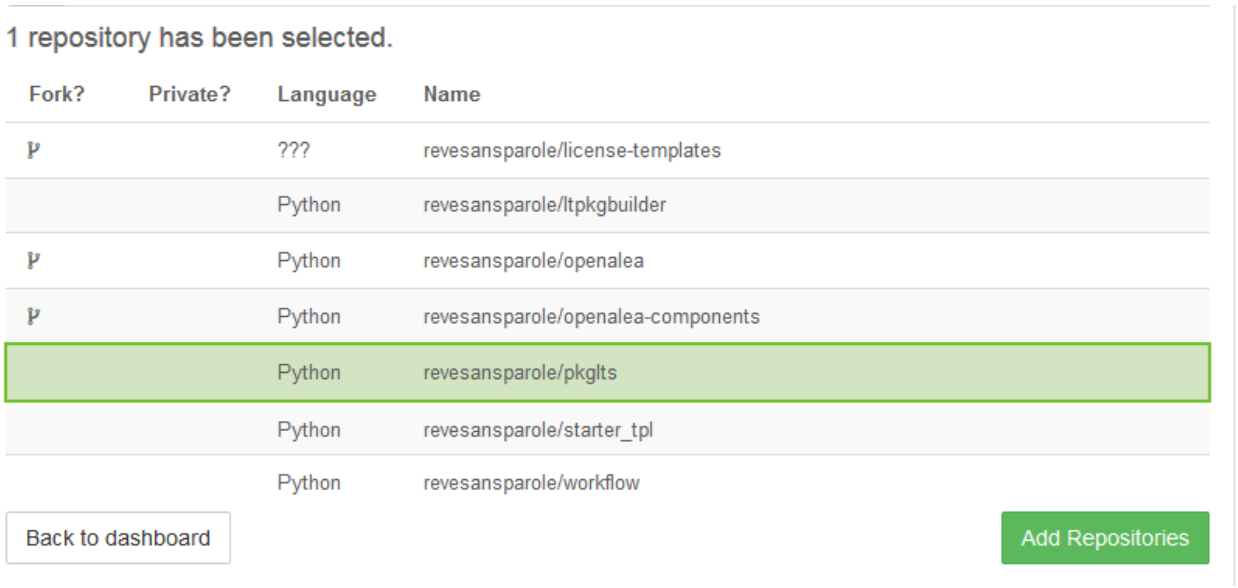
The process of registering on Landscape_io is made easy since you can use your github identity to sign up.

You must now be on the 'add project' page of landscape. If not click on the 'add repository' in the menu.

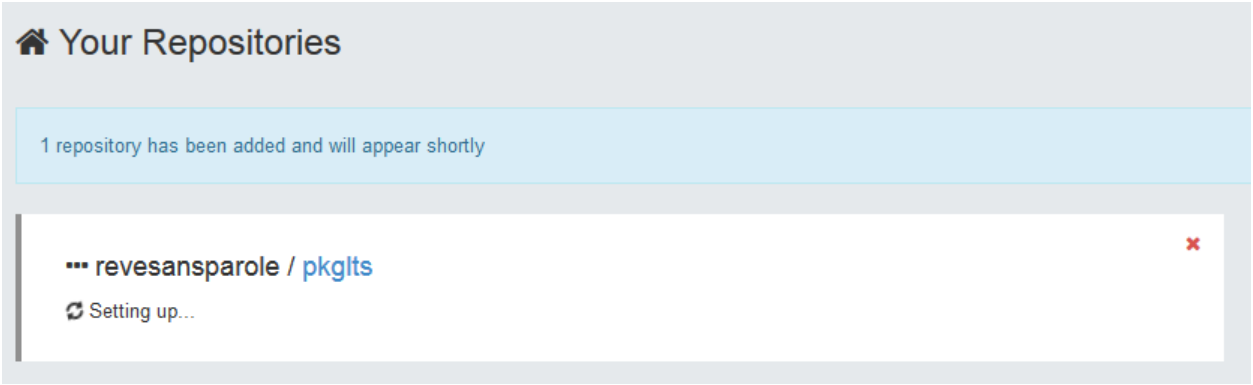


Adding your project

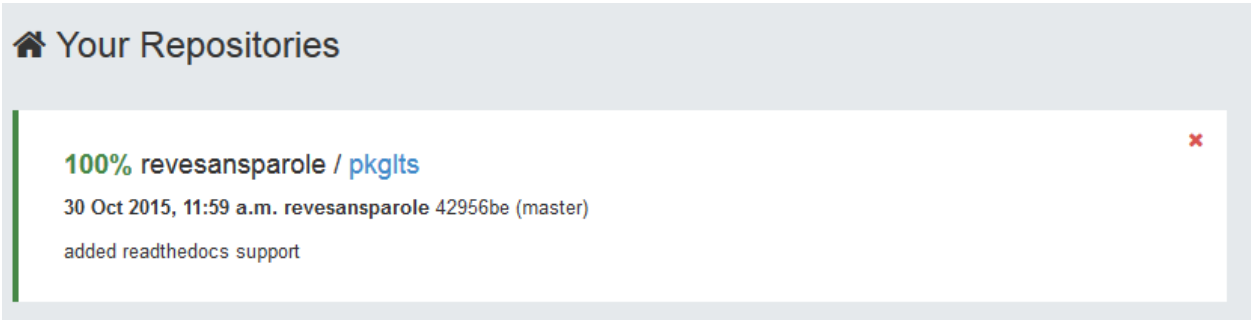
Scrolling down the page you will find a list of project you are contributing. Select yours and hit the 'add repositories' button.



Your project is added and you are redirected to the dashboard while your project is rebuilding.



However, you still need to hit the refresh button after some time if you want to see the result of the building process.



Final remark

If everything is successful, you must now have a health-100% green badge that show on top of your readme in the homepage of your project on github (hit refresh if you see nothing).

Unfortunately, Landscape_io keep track of your compliance over time. Starting at 100% will make it hard to improve :)

4.5.6 Simply create plugins

Warning: TODO

special group 'console_scripts' will expose your plugin as a command line tool.

4.5.7 Distribute on PyPi

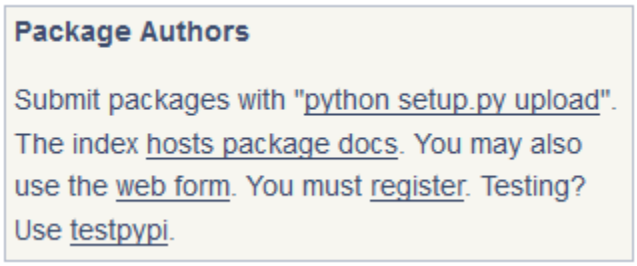
Uploading packages on PyPi is an expensive operation. So as not to pollute everybody environment just to perform some tests, in this tutorial we are going to use the testPyPi repository instead of the real one. Both behave the same so everything you do on one can be duplicated on the other once your package is ready for production stage.

Register

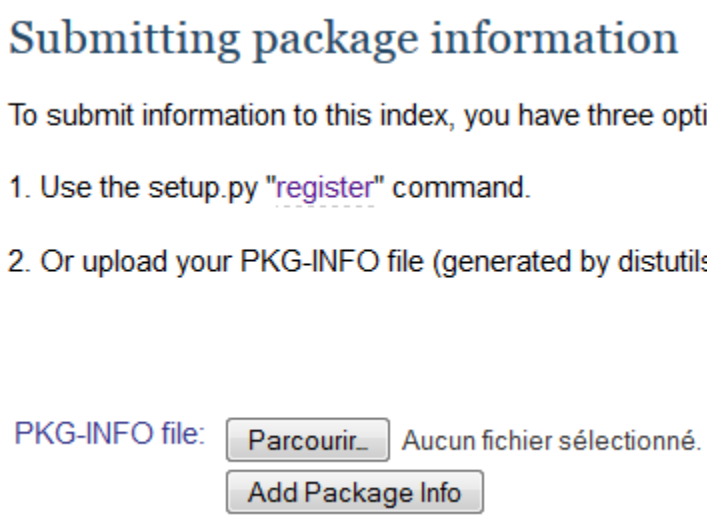
You need to setup an account on testPyPi. Sorry, no github identification possible and you'll have to redo this operation on the day you want to use the real PyPI.

Once you have your credentials, if you don't want to re-enter them every time you submit your package on pypi, you can store them in the '.pypirc' file that was added to your package. By default, even if you leave this file at the root of your package, it will be ignored by git. Hence no risk to store accidentally your credentials on github.

Once you are logged in you can register your package. Go on the 'Package Authors' section of the main page and click on the 'web form' link to register your package manually (we know there exists some method out there to do it automatically but let's just stick to things that work every time :)



You'll be redirected to a web page with three methods to register your package. Choose the one you prefer. We are going to follow the second one in this tutorial.

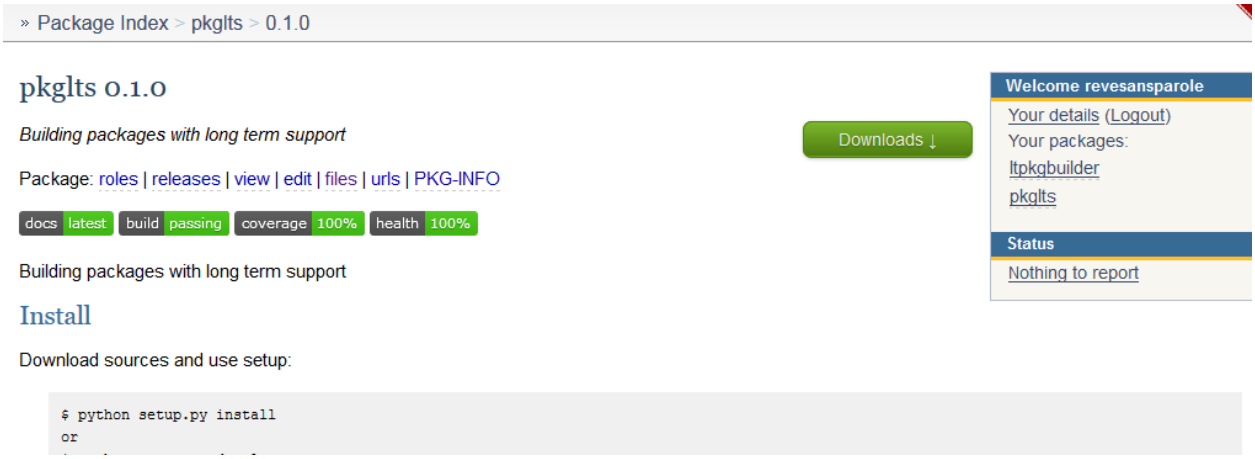


So first we need to create a 'PKG-INFO' file. This file is generated along your package whenever you run some packaging command so let's just create both a source distribution and a wheel of our package:

```
(dvlpt)$ python setup.py sdist bdist_wheel
```

Then on the web page we can click on the 'browse' or 'Parcourir' button on the 'PKG-INFO file:' line. In the explorer that pop up, we navigate to the directory of our package and then in "src/name_of_your_package.egg-info/" to find a 'PKG-INFO' file without extension.

We can now click on 'Add package info' button to register our package on testPyPi. If everything went well you must be redirected to the homepage of your project that display the README.rst file (must look similar to the github page then).



Upload your package

So far your package is registered but no distribution is available for download. Using twine tool, it is straightforward to upload both distributions packages we created above:

```
(dvlpt)$ twine upload dist/* -r test --config-file .pypirc
```

You can check at the bottom of your project page on Pypi, you must see a couple of line with the packages ready to be downloaded.

File	Type	Py Version	Uploaded on	Size
pkglts-0.1.0-py2.py3-none-any.whl (md5)	Python Wheel	py2.py3	2015-10-30	4KB
pkglts-0.1.0.zip (md5)	Source		2015-10-30	17KB

Downloads (All Versions):
0 downloads in the last day
0 downloads in the last week
0 downloads in the last month

Author: base.author_name
Keywords: packaging,package builder

Test your distribution

To test your distribution, simply create a new empty virtual environment and pip install your package in it. Then try to import it in python to check that everything went smoothly:

```
$ virtualenv testenv
$ testenv/scripts/activate
(testenv)$ pip install name_of_your_package --extra-index-url https://testpypi.python.
↪org/pypi
```

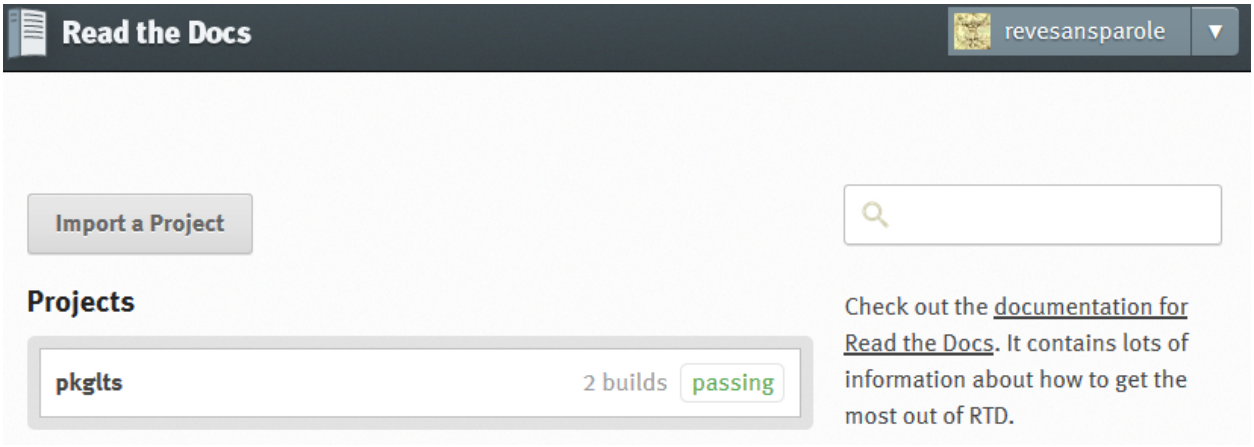
You can drop the 'extra-index-url' part if you used the regular pypi server. Now to test:

```
(testenv)$ python
>>> from name_of_your_package import version
>>> version.__version__
"0.1.0"
```

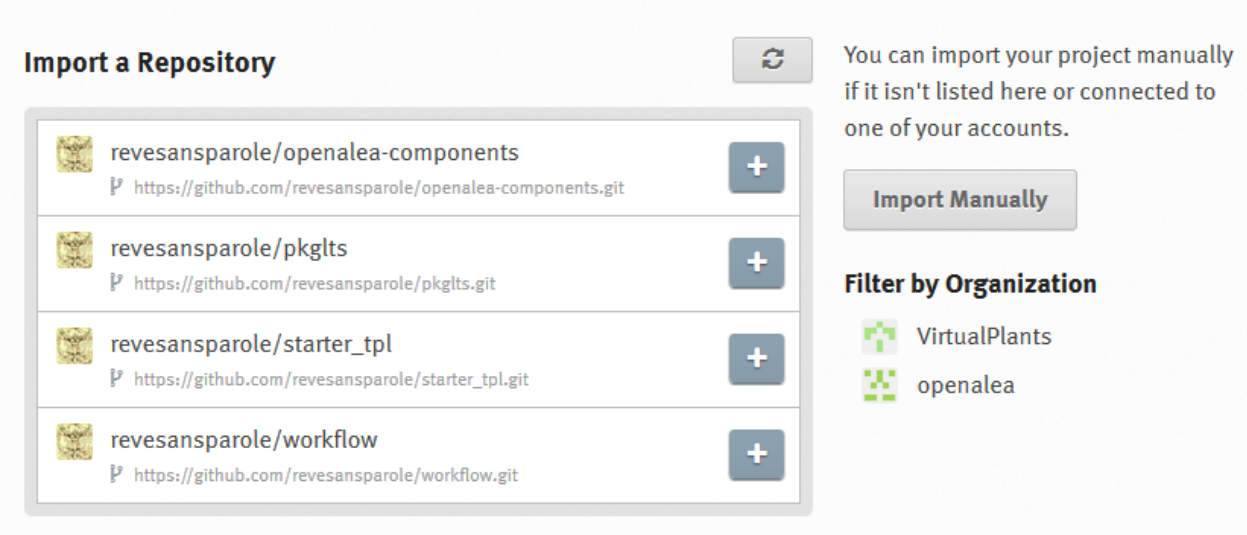
4.5.8 Register on ReadTheDocs

The first thing to do is to login on readthedocs. Unfortunately you can't use your github account this time and you have to create a couple login/password.

If your registration and login are successful you must arrive on your home page.



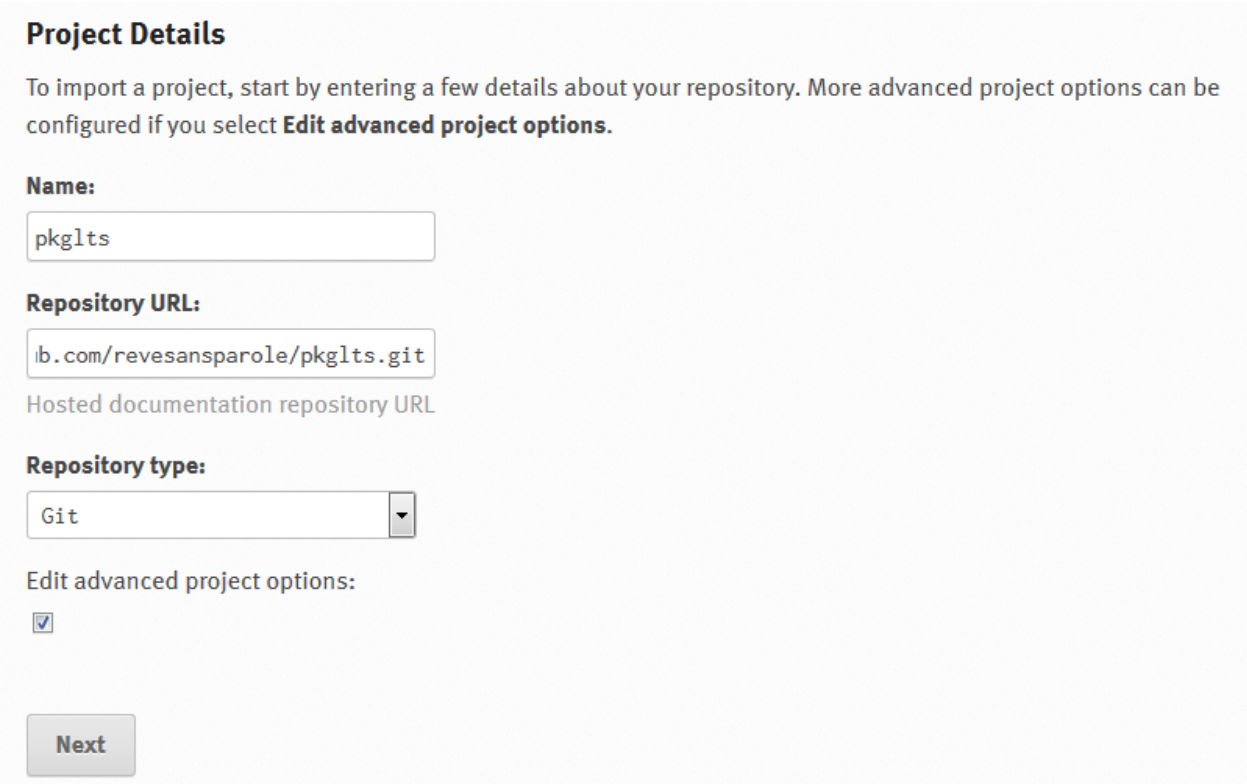
Hitting 'import a project' leads you to a list of projects you collaborate.



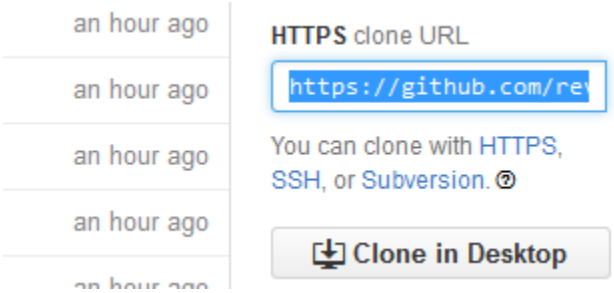
If nothing is showing try clicking on the refresh list button.

Importing your project

If you are lucky, clicking the '+' sign nearby your project name can be sufficient to import your project. Unfortunately you will certainly have to fall back on the 'import manually' method. Clicking the eponymous button will lead you to the registering page.



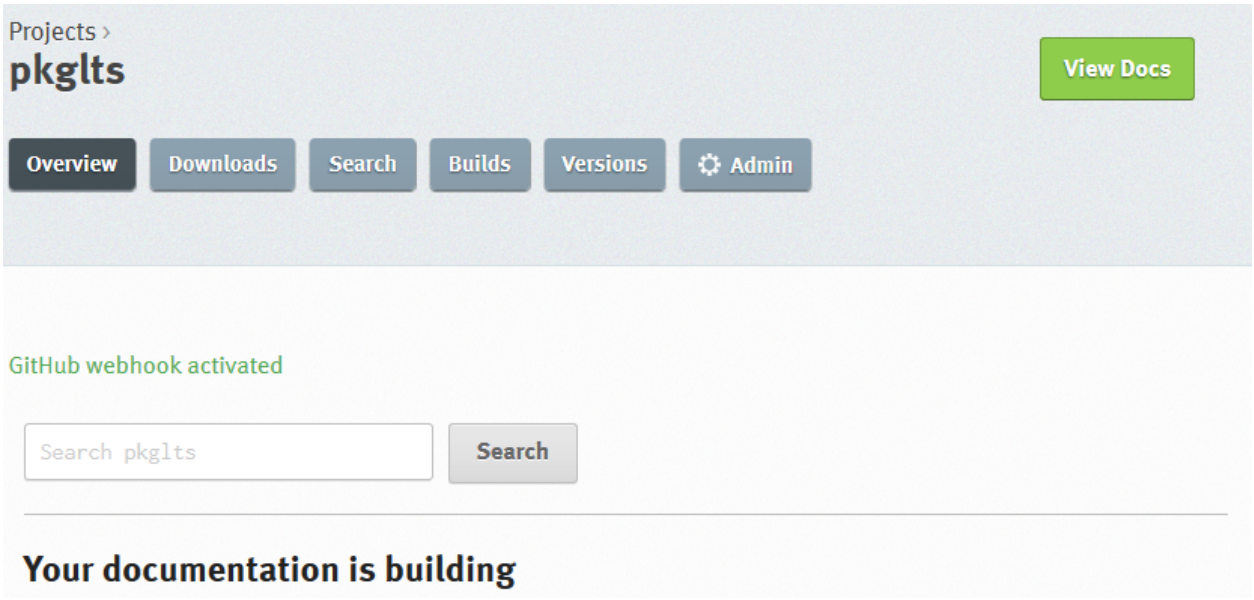
Fill the fields with the relevant information. 'name' can be the same name you use for your project on github. 'repository url' corresponds to the url of the github repository. You can copy/paste it from the 'HTTPS clone url' box on the home page of your project on github.



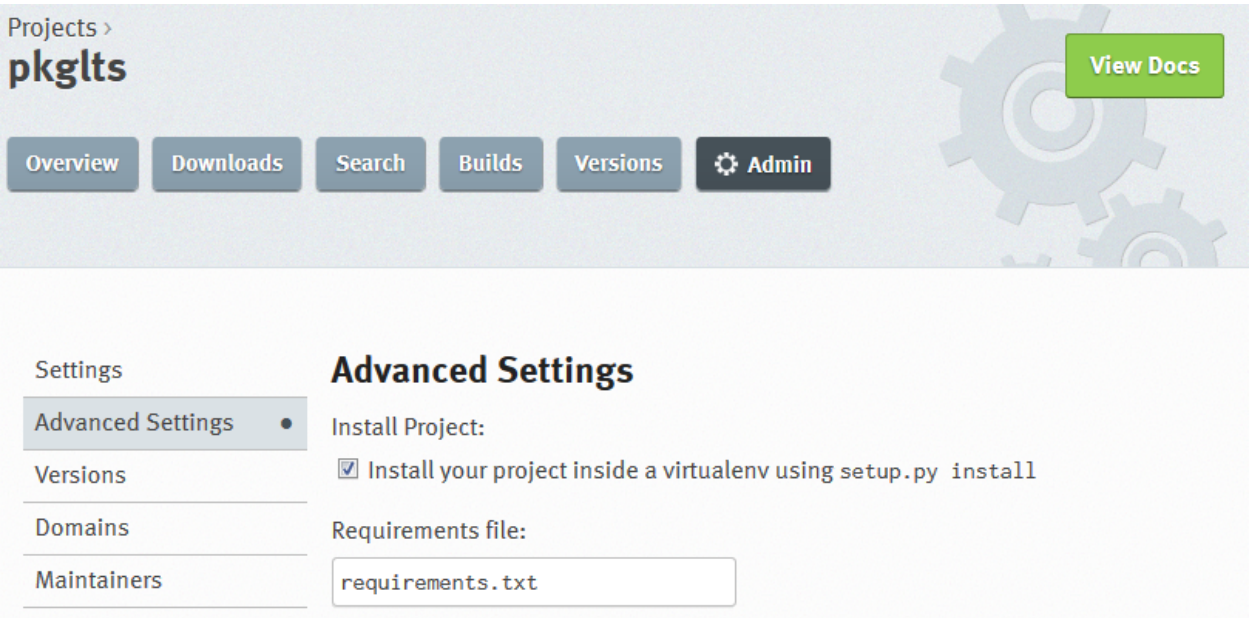
Leave the repository type on 'git' and don't forget to check the 'edit advanced' before you click on 'next' if you want to add more information.

Making the generation of your doc successful

The previous step must have led you to your project home page with a sentence reading that the build is in progress.



Unfortunately there is a good chance that this build will fail depending on the actual external package requirements of your project. To ensure subsequent build to be successful, you need to activate the virtualenv option. Check the corresponding box in the admin/advanced settings panel.



and enter 'requirements.txt' in the 'requirement file' field of the form. Don't forget to click on the 'submit' button at the bottom of the page. You'll be redirected to the project home page and a new build must be taking place. A successful one this time hopefully :)

Final remark

If everything is successful, you must now have a doc-latest green badge that show on top of your readme in the homepage of your project on github (hit refresh if you see nothing). This badge links to the public version of your doc.

No more excuses to postpone the writing of a comprehensive documentation for your beautiful code :)

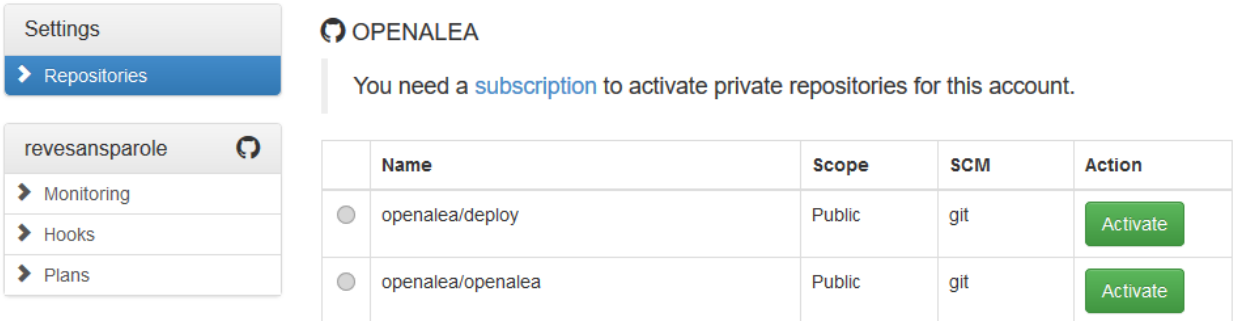
4.5.9 Register on requires.io

As always, you need to register on requires_io. This process is made easy since you can used you GitHub id to sign up.

You must arrive on the main page of requires_io with a small set of instructions to add a repository.

Add your project

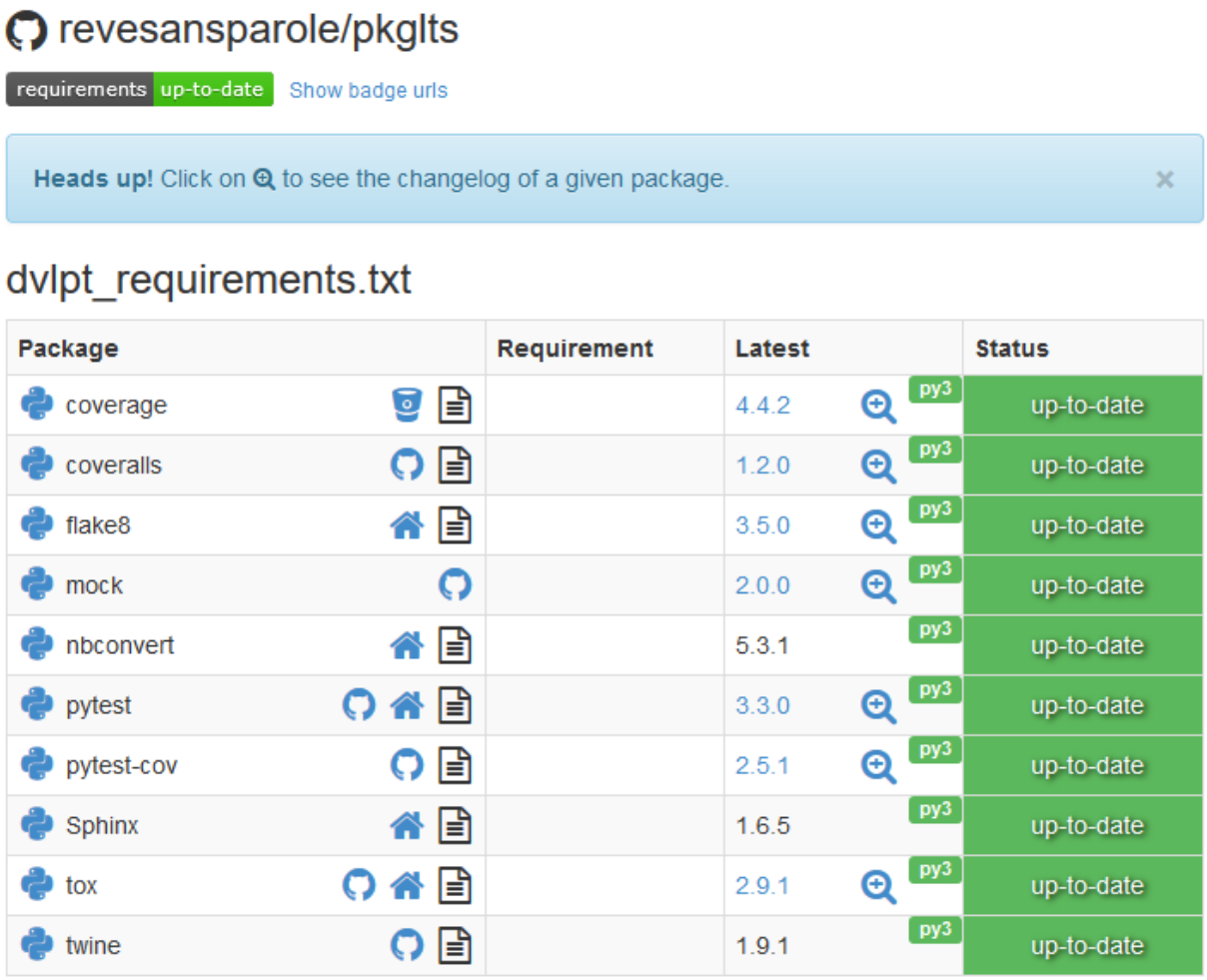
Registering a project is easy and you just need to click on the 'Activate' button.



requires_io use notifications from GitHub to gather information on your project, so you need to trigger a new action if you want to see some result, either:

```
$ git push
```

After requires had time to gather information, if you click the link provided for your porject on requires_io you must see the statistics on your code.



Final remark

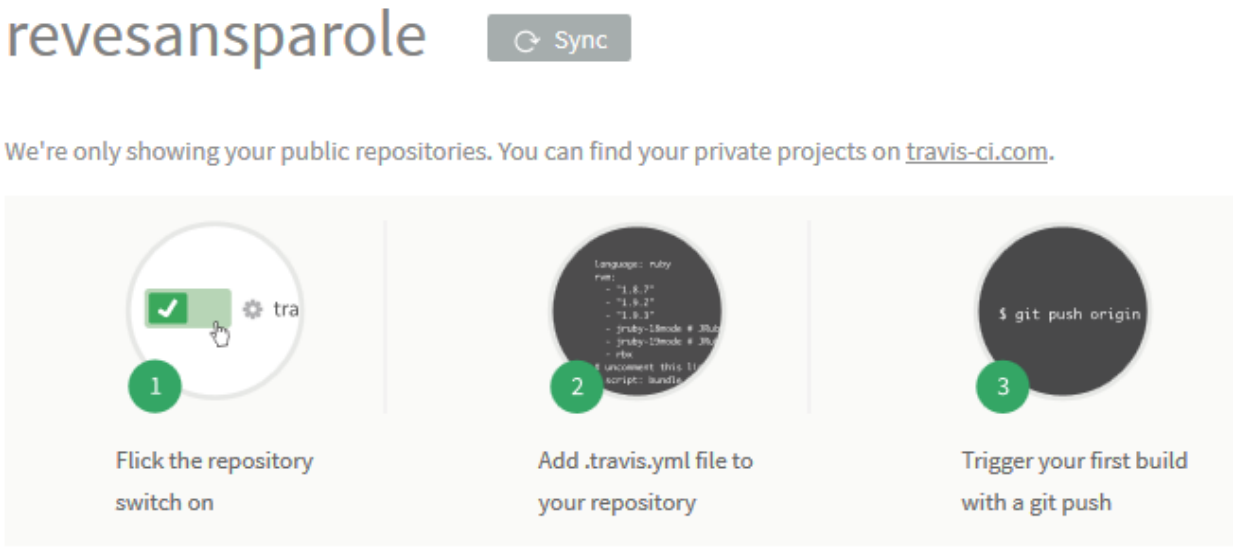
If everything is successful, you must now have a requirements up-to-date green badge that show on top of your readme in the homepage of your project on github (hit refresh if you see nothing).

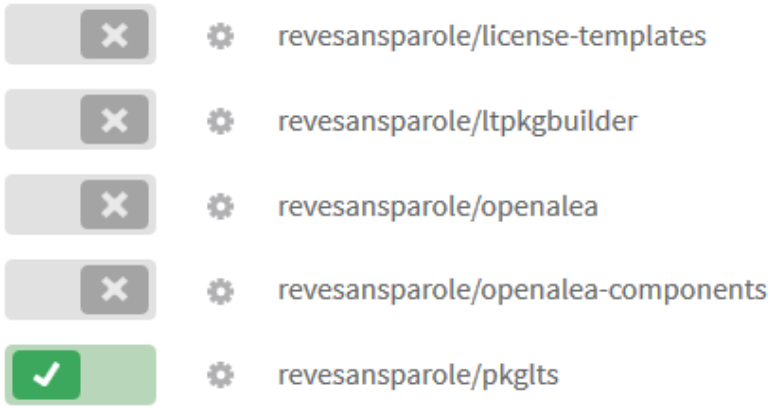
4.5.10 Register on Travis

The first thing to do is to sign up on Travis-CI. Luckily you can use your github credential to sign up so no need to create a new account.

Find your project

On your home page (‘https://travis-ci.org/profile/github_login_name‘) you will find all the projects you contribute. If yours is not showing try to hit the ‘Sync’ button.





The page offers a handy remainder on how to start, just follow it. Mostly just flip the switch in front of your project. Now is the right time to add the ‘travis’ option to your package if you have not done it already. It will take care of creating the needed ‘travis.yml’ file:

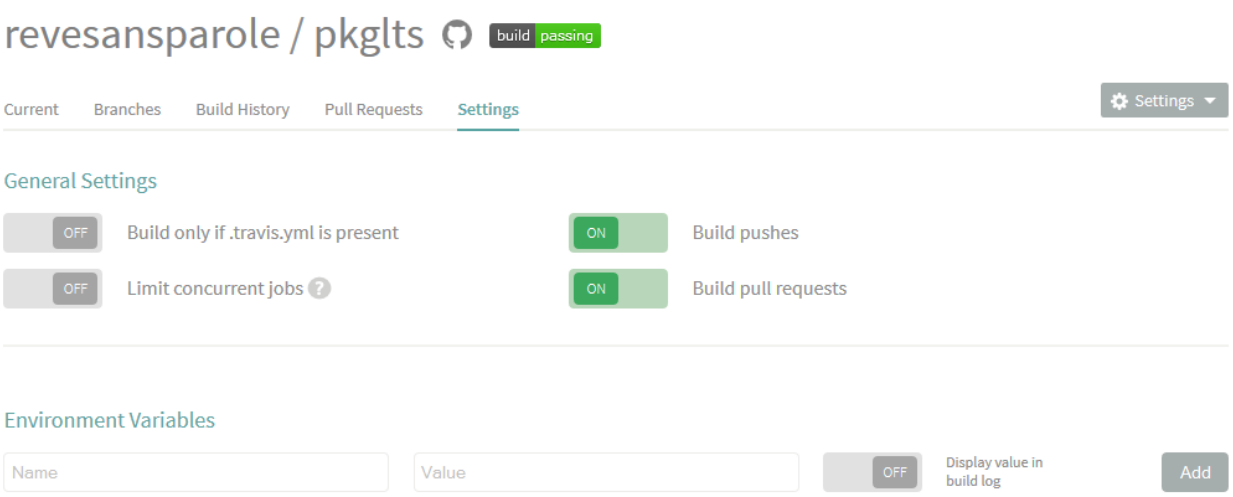
```
(dvlpt)$ pmg add travis
(dvlpt)$ pmg regenerate
```

A simple git push will trigger the first automatic rebuild:

```
$ git add --all
$ git commit -m"added travis-ct support"
$ git push
```

Check your build

If you click on the small gear near the name of your project you’ll be redirected to the setting page for your project.

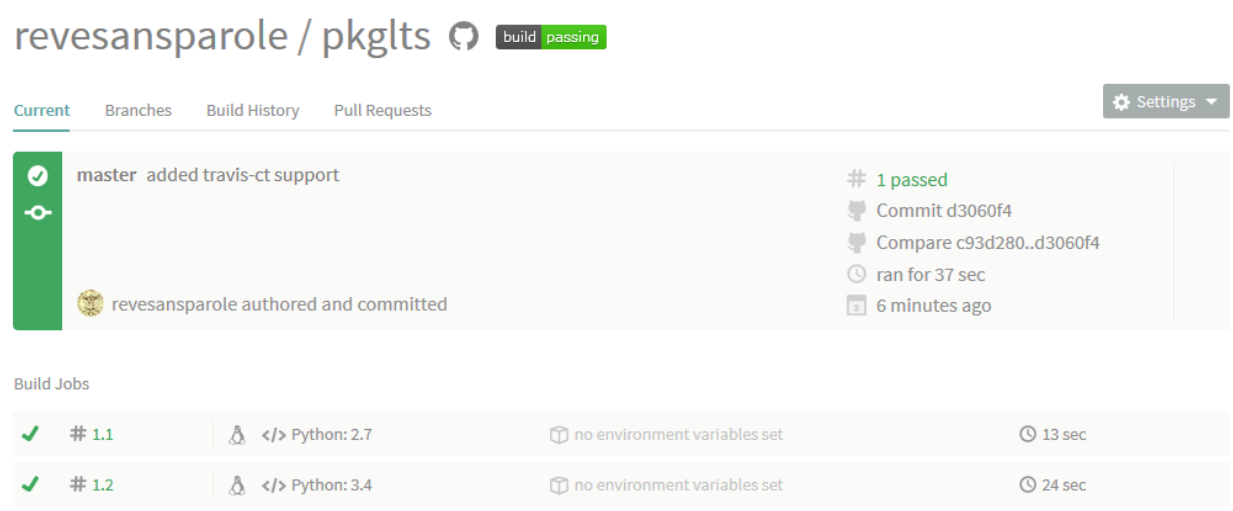


Hopefully, you'll see the badge stating that your build is successful. If you currently have no modules in your package, now maybe the right time to add some examples:

```
(dvlpt)$ manage example test

$ git add --all
$ git commit -m"added some example files"
$ git push
```

Wait for a few moments that the server rebuild your project and then you will see the build-passing badge turn green (maybe refresh your explorer). By clicking on the name of your package near the badge you will be redirected to the report page for your project.



Final remark

If everything is successful, you must now have a build-passing green badge that show on top of your readme in the homepage of your project on github (hit refresh if you see nothing).

Downside is that if you now make poor code, you get the infamous red build-fail badge showing :)

CHAPTER 5

List of currently available options

5.1 Basic

Basic options help you organize your code and provide core 'good practices' functionality in your package.

5.1.1 base

Add some base functionality to a package, i.e.:

- a src directory to store python files
- some basic meta information like owner name and package name

Modifications

If namespace is None:

else:

Quick tutorial

Follow these steps for a quick setup:

```
(dvlpt)$ pmg init
(dvlpt)$ pmg add base
```

Edit your package config file ('pkg_cfg.json' in ".pkglts" directory at the root of your package) using your favorite json editor (a normal text editor will do).

```
{
    "_pkglts": {
        "auto_install": true,
```

```
        "install_front_end": "stdout",
        "use_prompts": false
        "version": 6
    },
    "base": {
        "authors": [
          [
            "moi",
            "moi@email.com"
          ]
        ],
        "namespace": null,
        "namespace_method": "pkg_util",
        "pkgname": "name",
        "url": null
    }
}
```

Do not change sections starting with "_" (e.g. '_pkglts'), they are private sections used by pkglts as configuration. Change owner for your own name and "pkgname" for the name of your package (by default this one must be the name of the directory in which you started to code).

```
"base": {
    "authors": [
      [
        "revesansparole",
        "revesansparole@gmail.com"
      ]
    ],
    "namespace": null,
    "namespace_method": "pkg_util",
    "pkgname": "mypkg"
    "url": null
}
```

Then:

```
(dvlpt)$ pmg regenerate
```

5.1.2 test

Simple testing facilities making extensive use of test frameworks to run tests. By default pytest is selected but this can be customized in the option of the 'test' section of the configuration file attached to the project.

For the moment only two frameworks are supported: pytest and nose.

Two different ways to run the tests, either through the setuptools call to setup.py:

```
(dvlpt)$ python setup.py test
```

or through the use of the 'pytest' or 'nosetests' command line argument:

```
(dvlpt)$ pytest
(dvlpt)$ nosetests
```

Have a look at nose_cmd for more information on the way to use nose.

Modifications

Quick tutorial

Follow these steps for a quick setup (do *base* quick tutorial first if you haven't done it yet):

```
(dvlpt)$ pmg add test
(dvlpt)$ pmg regenerate
(dvlpt)$ pmg example test

(dvlpt)$ pytest
```

```
(dvlpt) C:\Users\jerome\Desktop\tmp\toto>nosetests
.....
----------------------------------------------------------------------
Ran 5 tests in 0.015s

OK

(dvlpt) C:\Users\jerome\Desktop\tmp\toto>
```

5.1.3 doc

Add some documentation in your package:

- a brief description
- readme headers
- list of authors
- history
- contributing

The examples associated with this option provide a scaffold for a more complete documentation in the doc directory:

- a basic index file

Modifications

Configuration

You need to decide rapidly (aka before regenerating the package) the format you intend to use to write the documentation:

- Complete documentation of code, use the default 'rst' (only one that actually

work with sphinx). - webpages (aka user oriented documentation), then you can use 'md'

Quick tutorial

Follow these steps to install this option and the associated files:

```
(dvlpt)$ pmg add doc
(dvlpt)$ pmg regenerate
```

Meaning of pieces of information

Readme

Provided as an example file, a basic body for the readme file with different sections depending on the installed options. The README.rst file is regenerated automatically so out of reach. However, it simply add a few headers to a README_body file located in the 'doc' directory. We encourage you to modify this file to suit your needs.

Authors

The author list is automatically generated from the author_name, author_email information stored in the 'base' part of the package configuration.

If your package is stored on GitHub or Gitlab, the complete list of authors and contributors will be retrieved from the website.

Contributing

An example file describing how people can contribute to the package is available as an example:

```
(dvlpt)$ pmg example doc
```

This file makes more sense if you already installed the 'github' or 'gitlab' option.

History

History of your package is generated automatically from the github or gitlab commit tags and messages if possible.

5.1.4 license

Use the same approach than the dead project lice to add a license to your package. The name of the license is case insensitive (i.e. CeCILL-C and cecill-c refer to the same license in the end).

Quick setup:

```
(dvlpt)$ pmg add license
```

```
"license": {
    "name": "CeCILL-C",
    "organization": "organization name",
    "project": "{{key, base.pkgname}}",
    "year": "2015"
}
```

Then:

```
(dvlpt)$ pmg regenerate
```

Modifications

The list of templates for all available licenses in former project lice can be found here. They have been copied and extended into this project. Available licenses:

agpl3

header:

```
GNU AFFERO GENERAL PUBLIC LICENSE
Version 3, 19 November 2007
Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies
```

apache

header:

```
Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
```

bsd2

header:

```
Copyright (c) {{key, license.year}}, {{key, license.organization}}
All rights reserved.
Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:
```

bsd3

header:

```
Copyright (c) {{key, license.year}}, {{key, license.organization}}
All rights reserved.
Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:
```

cc0

header:

```
Creative Commons Legal Code
CC0 1.0 Universal
CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS DOCUMENT DOES NOT CREATE AN
```

cc_by

header:

```
Creative Commons Legal Code
Attribution 3.0 Unported
CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
```

cc_by_nc

header:

```
Creative Commons Legal Code
Attribution-NonCommercial 3.0 Unported
CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
```

cc_by_nc_nd

header:

```
Creative Commons Legal Code
Attribution-NonCommercial-NoDerivs 3.0 Unported
CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
```

cc_by_nc_sa

header:

```
Creative Commons Legal Code
Attribution-NonCommercial-ShareAlike 3.0 Unported
CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
```

cc_by_nd

header:

```
Creative Commons Legal Code
Attribution-NoDerivs 3.0 Unported
CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
```

cc_by_sa

header:

```
Creative Commons Legal Code
Attribution-ShareAlike 3.0 Unported
CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
```

cddl

header:

```
COMMON DEVELOPMENT AND DISTRIBUTION LICENSE Version 1.0 (CDDL-1.0)
1. Definitions.
1.1. Contributor means each individual or entity that creates or
contributes to the creation of Modifications.
```

cecill-c

header:

```
CeCILL-C FREE SOFTWARE LICENSE AGREEMENT
Notice
This Agreement is a Free Software license agreement that is the result
of discussions between its authors in order to ensure compliance with
```

epl

header:

```
Eclipse Public License, Version 1.0 (EPL-1.0)
THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC
LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM
CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.
```

gpl2

header:

```
The GNU General Public License (GPL-2.0)
Version 2, June 1991
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
```

gpl3

header:

```
GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007
Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies
```

inra_license_agreement

header:

```
{{key, base.pkgname}}
======
License agreement for final academic user
#########################################
```

isc

header:

```
Copyright (c) {{key, license.year}}, {{key, license.organization}}
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.
```

lgpl

header:

```
GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007
Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies of this license
```

mit

header:

```
Copyright (c) {{ license.year }} {{ license.organization }}
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
```

mpl

header:

```
Mozilla Public License, version 2.0
1. Definitions
1.1. "Contributor"
means each individual or legal entity that creates, contributes to the
```

unlicense

header:

```
This is free and unencumbered software released into the public domain.
Anyone is free to copy, modify, publish, use, compile, sell, or
distribute this software, either in source code form or as a compiled
binary, for any purpose, commercial or non-commercial, and by any
```

wtfpl-header-warranty

header:

```
This program is free software. It comes without any warranty, to
the extent permitted by applicable law. You can redistribute it
and/or modify it under the terms of the Do What The Fuck You Want
To Public License, Version 2, as published by Sam Hocevar. See
```

wtfpl

header:

```
DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE
Version 2, December 2004
Copyright (C) {{key, license.year}} {{key, license.organization}}
Everyone is permitted to copy and distribute verbatim or modified
```

x11

header:

```
Copyright (C) {{key, license.year}} {{key, license.organization}}
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
```

zlib

header:

```
Copyright (c) {{key, license.year}} {{key, license.organization}}
This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.
```

5.1.5 version

Simple versioning system for your package. Use of a generated 'version.py' file to store the '__version__' attribute of your package:

```
(dvlpt)$ pmg version
(dvlpt)$ pmg regenerate
```

By default, the system keep track of a version number major.minor.post. Each number needs to be entered manually by editing the config file then regenerating the package.

Note: Experimental: If your package is hosted on github you can set this option to go and fetch these numbers automatically each time you regenerate your package.

Modifications

5.1.6 git

Add the relevant files to make your package compliant with Git versioning system.

This option takes care of the local package side of git. If you wish to host your code on a distant server, consider using the *github* or *gitlab* option.

Modifications

5.2 Extended

Based on the previous options, a few tools make your life easier and extend basic functionalities.

5.2.1 sphinx

Extend basic documentation to use the sphinx set of tools. This option will use the 'default' theme by default instead of 'classic' to ensure that the right theme will be selected on readthedocs. If the 'autodoc_dvlpt' parameter is set to True, then docstrings will also be compiled.

Modifications

Quick tutorial

Follow these steps to build your first comprehensive documentation. I assume you also install the attached example files:

```
(dvlpt)$ pmg add sphinx
(dvlpt)$ pmg regenerate
(dvlpt)$ pmg example sphinx
```

If you already installed the *pysetup* option:

```
(dvlpt)$ python setup.py build_sphinx
```

will produce a set of html pages in 'build/sphinx'. Open the 'index.html' file in the 'html' sub directory to access the main index. Alternatively you'll find another 'index.html' file in the 'docexample' directory that provide examples of sphinx extensions usage.

If you don't want to install the *pysetup* option, you simply need to run the make command from the doc directory:

```
(dvlpt)$ cd doc
(dvplt) doc$ make
```

This will produce a build directory with the same architecture as explained above.

5.2.2 notebook

This option allows to convert all notebooks (.ipynb) specified in the src_directory parameter of the 'notebook' option (default : "example") to RestructuredText (.rst) format.

Each rst file produced is writen in "doc/_notebook" folder reproducing the hierarchy of files found in src_directory. An 'index.rst' file is also generated to list all the notebooks.

Command line

```
(dvlpt)$ pmg rg notebook
```

User Warnings

1. Each notebook need a title to be referenced in the index

Then you need to rebuild the documentation to integrate the notebooks:

```
(dvlpt)$ python setup.py build_sphinx
```

Example

Notebook

Notebook Example, Main Title

Some summary

subtitle

Easy to read documentation

Create variables

```
from math import pi, sin
```

```
x = range(100)
y = [sin(v * pi / 20) for v in x]
```

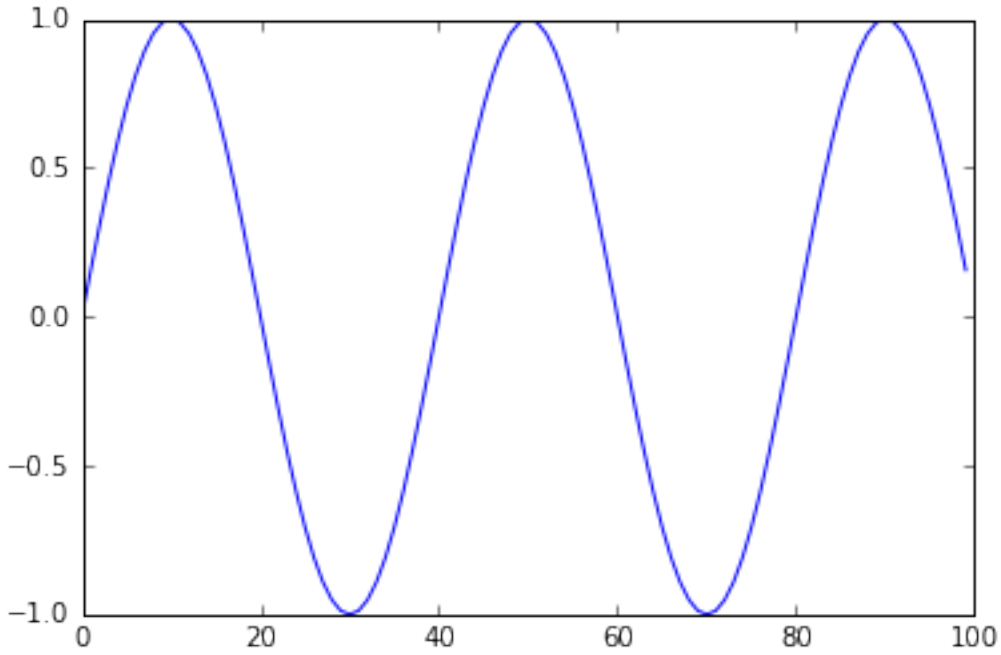
Display

```
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt

plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7f04d057ad90>]
```



5.2.3 data

Simply add data (i.e. non code files) to your package. This option provide a simple method to add data to a package that's robust enough to handle any way of packaging.

Look at a *Add data tutorial* for a way to use this option.

Modifications

5.3 Distributing your code

The base of distributing your code comes with:

5.3.1 pysetup

Add a 'setup.py' to your package to make it compliant with setuptools. This will allow an easy distribution of your package. Since this option requires most basic options, it's a good proxy to add to a newly created package to avoid multiple 'manage add opt' commands.

Quick setup:

```
(dvlpt)$ pmg add pysetup
> intended versions [27]:
...
(dvlpt)$ pmg regenerate
```

Modifications

In order to make your package easily accessible for others, these options help you host your code on a distant server:

5.3.2 github

Add the relevant files to make your package ready for hosting on GitHub. This option is the base for all web related options. It depends on the *git* option for the local side of version management.

This option takes care of the local package side of github (e.g. .gitignore file) but you still need to register your project on github. See *Register on Github* for example.

5.3.3 gitlab

Add the relevant files to make your package ready for hosting on a Gitlab. It depends on the *git* option for the local side of version management.

This option takes care of the local package side of gitlab (e.g. .gitignore file) but you still need to register your project on gitlab. See *Register on Gitlab* for example.

These options help you organize the structure to be PyPi ready:

5.3.4 coverage

Add the coverage option to your test suite. It tells you exactly which parts of your code need more thorough testing.

There is two ways to use it, either using the 'coverage' command line tool or through the coverage option of your test suite. Both are already configured by the system.

> **Warning:** Both approaches requires the *pysetup* option for nosetests or pytest to take the config into account. Use the *Nosetests command line* approach if you don't want to add a setup.py to your package.

Modifications

Quick tutorial

Follow these steps to quickly install the option. I assume you already followed *test* quick tutorial:

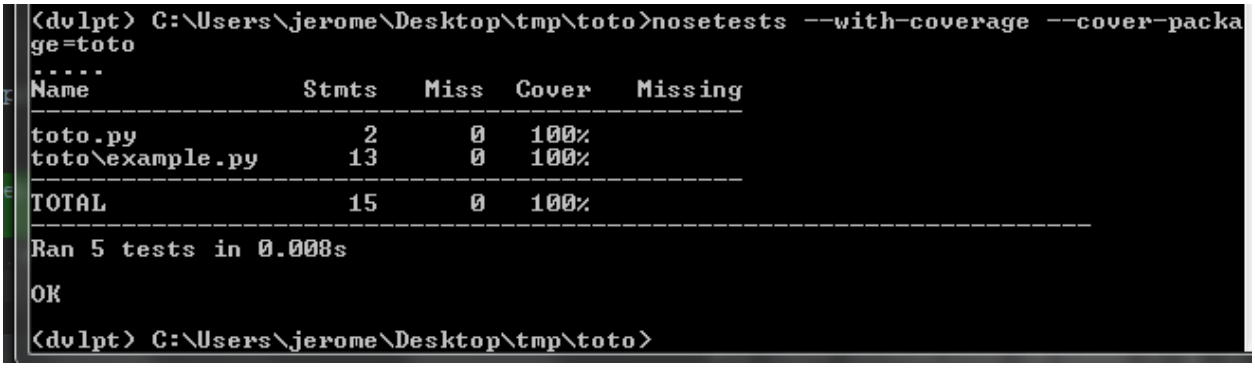
```
(dvlpt)$ pmg add coverage
(dvlpt)$ pmg regenerate
```

Nosetests

Relevant options have been written in the config file to include a coverage report with each call to 'nosetests' or 'pytest':

```
(dvlpt)$ nosetests
```

will run all the tests and print a simple report as explained on coverage_report.



Nosetests command line

You can force nosetests to use the coverage plugin if you prefer not to use a setup.cfg file:

```
(dvlpt)$ nosetests --with-coverage --cover-package=name_of_your_package
```

This will produce the exact same result than the above mentioned method.

Coverage command line tool

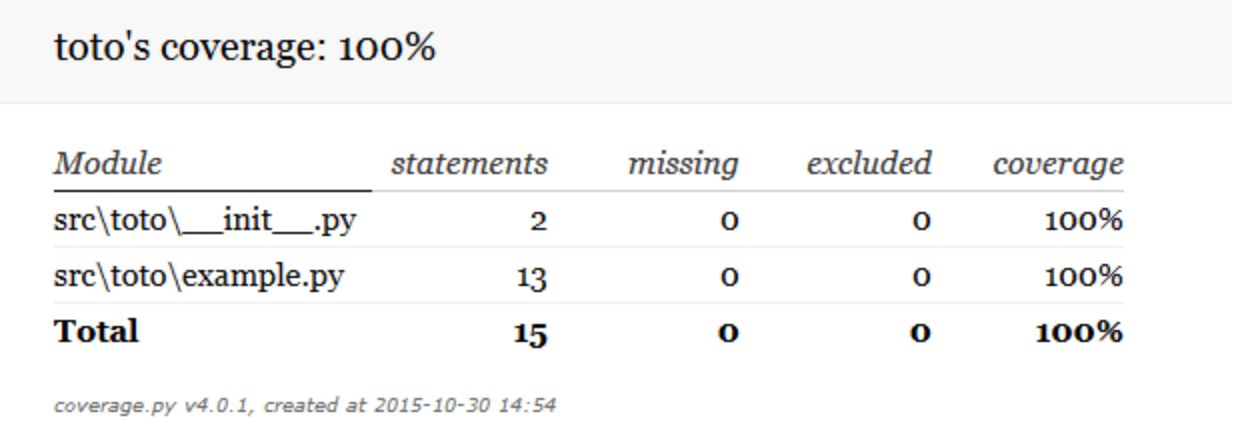
Alternatively you can use 'coverage' to both run the tests, collate the results and produce some reports:

```
(dvlpt)$ coverage run setup.py test
(dvlpt)$ coverage report
```

The coverage module is configured as to produce the same output than when calling nosetests. You can also produce an html report that directly point to the incriminated non covered statements:

```
(dvlpt)$ coverage html
```

The result will be written in 'build/htmlcov'. Open the 'index.html' file in it to show the report.



toto's coverage: 100%

Module	statements	missing	excluded	coverage
src\toto__init__.py	2	0	0	100%
src\toto\example.py	13	0	0	100%
Total	**15**	**0**	**0**	**100%**

coverage.py v4.0.1, created at 2015-10-30 14:54

5.3.5 tox

Extend the testing of your package to other python environments. It requires that you install other python interpreters on your computer. By default, each time you regenerate your package it will check among all the 'intended_versions' that you registered with the *pysetup* option the python environments that actually exists on your machine and create a tox.ini file with the relevant options:

```
(dvlpt)$ pmg add tox
(dvlpt)$ pmg regenerate
```

To invoke tox, just use the command line tool:

```
(dvlpt)$ tox
```

To produce a result like this one.

```
(dvlpt) C:\Users\jerome\Desktop\tmp\toto>tox
GLOB sdist-make: C:\Users\jerome\Desktop\tmp\toto\setup.py
py27 create: C:\Users\jerome\Desktop\tmp\toto\.tox\py27
py27 installdeps: -rC:\Users\jerome\Desktop\tmp\toto/requirements.txt
py27 inst: C:\Users\jerome\Desktop\tmp\toto\.tox\dist\toto-0.1.0.zip
py27 installed: coverage==4.0.1,nose==1.3.7,pluggy==0.3.1,py==1.4.30,toto==0.1.0
,tox==2.1.1,virtualenv==13.1.2,wheel==0.24.0
py27 runtests: PYTHONHASHSEED='949'
py27 runtests: commands[0] | nosetests
.....
Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
toto.py               2      0   100%
toto\example.py      13      0   100%
-----------------------------------------------
TOTAL                15      0   100%
----------------------------------------------------------------------
Ran 5 tests in 0.010s

OK
py34 create: C:\Users\jerome\Desktop\tmp\toto\.tox\py34
py34 installdeps: -rC:\Users\jerome\Desktop\tmp\toto/requirements.txt
py34 inst: C:\Users\jerome\Desktop\tmp\toto\.tox\dist\toto-0.1.0.zip
py34 installed: coverage==4.0.1,nose==1.3.7,pluggy==0.3.1,py==1.4.30,toto==0.1.0
,tox==2.1.1,virtualenv==13.1.2,wheel==0.24.0
py34 runtests: PYTHONHASHSEED='949'
py34 runtests: commands[0] | nosetests
.....
Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
toto.py               2      0   100%
toto\example.py      13      0   100%
-----------------------------------------------
TOTAL                15      0   100%
----------------------------------------------------------------------
Ran 5 tests in 0.025s

OK
_____________________________________ summary ______________________________________
  py27: commands succeeded
  py34: commands succeeded
  congratulations :)

(dvlpt) C:\Users\jerome\Desktop\tmp\toto>
```

Modifications

5.3.6 flake8

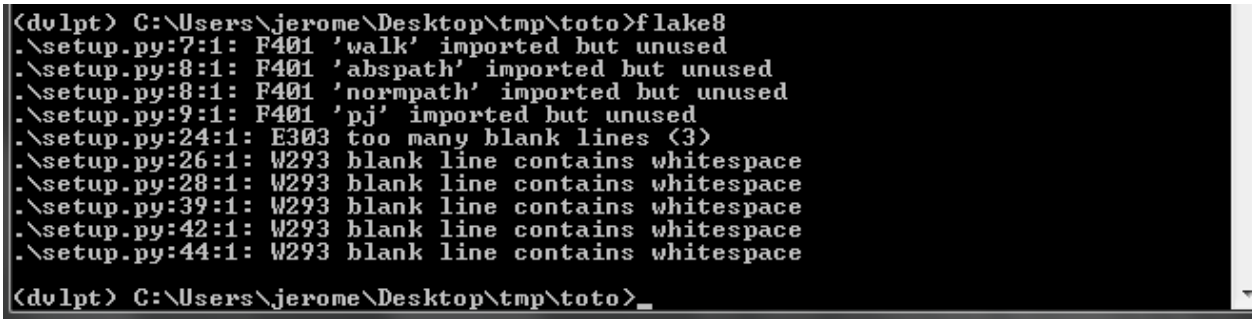
Test the compliance of your code with PEP8 definitions:

```
(dvlpt)$ pmg add flake8
(dvlpt)$ pmg regenerate
```

Basic usage:

```
(dvlpt)$ flake8
```

This will parse your entire package to produce a result like this one.



More advanced documentation can be found on the flake8 website.

Then you can use the 'pypi' option to help you put your package on the cheese shop:

5.3.7 pypi

Format your package to upload on PyPi:

```
(dvlpt)$ pmg add pypi
(dvlpt)$ pmg regenerate
```

Follow this simple tutorial: *Distribute on PyPi*, to learn an easy way to register your package on PyPi.

Modifications

Alternatively, if you chose to use 'conda' as your package manager:

5.3.8 conda

Install a Conda recipe in the package to generate Conda packages.

Modifications

5.4 Web services

Once your code is registered on GitHub, a set of options helps you benefit from already developed online tools:

5.4.1 travis

Your package test wonderfully using the *tox* tool. Now maybe the right time to automate the process using Travis-CI.

This option will add the relevant files to expose your package to Travis-CI. You still need to perform by hand the activation of your package on Travis-CI. Have a look at *Register on Travis* for a step by step guide on how to do that.

Modifications

5.4.2 readthedocs

You've built your documentation using *sphinx*, It looks perfect! Now maybe the right time to use the ReadTheDocs web service to perform this task automatically and expose your documentation to the rest of the world.

This option add the relevant files to expose your package to ReadTheDocs, a web service that will automatically compile and store your documentation. You still need to perform the registration by hand. Look at *Register on ReadTheDocs* for a step by step guide:

```
(dvlpt)$ pmg add readthedocs
(dvlpt)$ pmg regenerate

$ git add --all
$ git commit -m"added readthedocs support"
$ git push
```

5.4.3 landscape

You've run the 'flake8' command after installing the *flake8* option and you don't have that many complains. Maybe now is a good time to automate the process to keep track of how your code improve over time.

This option add the relevant files to expose your package to Landscape_io, a web service that will automatically check the compliance of your code with PEP8 recommendations. This option install local configuration files but you still need to register by hand. Have a look at the Landscape Documentation or *Register on Landscape.io* tutorial for more information:

```
(dvlpt)$ pmg add landscape
(dvlpt)$ pmg regenerate

$ git add --all
$ git commit -m"added landscape.io support"
$ git push
```

Modifications

5.4.4 coveralls

Travis-CI is automatically rebuilding and testing your code every time you push your modifications on GitHub. The *coverage* option let you test your code coverage locally. Now, if you want to get your code coverage with every Travis-CI build, Coveralls_io is the right tool for you.

This option add the relevant files to let your package use Coveralls_io, a web service that will automatically check the test coverage of your code. This option install local configuration files but you still need to register by hand. Have a look at the Coveralls Documentation or *Register on Coveralls.io* tutorial for more information:

```
(dvlpt)$ pmg add coveralls
(dvlpt)$ pmg regenerate

$ git add --all
$ git commit -m"added coveralls support"
$ git push
```

5.4.5 requires

requires_io is a web service that scan your code every time you push on GitHub, detect all your dependencies and check wether they are still up to date.

This option add the relevant files to let your package use requires_io. This option install local configuration files but you still need to register your project by hand. Have a look at the Requires.io Documentation or *Register on requires.io* tutorial for more information:

```
(dvlpt)$ pmg add requires
(dvlpt)$ pmg regenerate

$ git add --all
$ git commit -m"added requires support"
$ git push
```

5.5 External options

Options developed for specific purposes. Why not consider adding your option? You can easily customize *pkglts* for your very own purpose by creating a plugin. The *plugin_project* option allow you to easily create such package for extending *pkglts*.

5.5.1 plugin_project

Create a scaffold to write a plugin for *pkglts*

Modifications

If namespace is None:

Quick tutorial

Follow these steps for a quick setup that will create a plugin called **plugin_name**:

```
(dvlpt)$ mkdir plugin_name
(dvlpt)$ cd plugin_name
(dvlpt) \plugin_name\$ pmg init plugin_project
```

Edit your package config file ('pkg_cfg.json' in ".pkglts" directory at the root of your package) using your favorite json editor (a normal text editor will do).

Do not change sections starting with "_" (e.g. '_pkglts'), they are private sections used by pkglts as configuration. Change owner for your own name and "pkgname" for the name of your plugin (by default this one must be the name of the directory in which you started to code).

Then:

```
(dvlpt) \plugin_name\$ pmg rg
```

If you want your plugin to generate some files, you can install an example of them using:

```
(dvlpt) \plugin_name\$ pmg example plugin_project
```

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Implement your own option

You can easily extend *pkglts* by implementing your own option as a plugin. There are two types of plugins:

- options that implement a new feature useful for everybody. Don't hesitate to contact us or submit a

pull request if you want to insert it into the main pkglts repo. * plugins that implement a way to create packages specific to your organization. Don't hesitate to make your life easier if you always create projects with the same options.

6.1.3 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

6.1.4 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

6.1.5 Write Documentation

pkglts could always use more documentation, whether as part of the official pkglts docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.6 Submit Feedback

The best way to send feedback is to file an issue at issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *pkglts* for local development.

1. Fork the *pkglts* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pkglts.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenv installed, this is how you set up your fork for local development:

```
$ virtualenv dvlpt
$ dvlpt/script/activate
(dvlpt)$ python setup.py develop
```

4. Create a branch for local development (wip stands for work in progress):

```
(dvlpt)$ git checkout -b wip_name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
 (dvlpt)$ cd pkglts
 (dvlpt) pkglts$ flake8
 (dvlpt) pkglts$ nosetests
 (dvlpt) pkglts$ tox

To get flake8 and tox, just pip install them into your virtualenv.
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin wip_name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, and 3.5. Check https://travis-ci.org/revesansparole/pkglts/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ nosetests test/test_XXX
```

CHAPTER 7

Credits

7.1 Development Lead

- revesansparole, <revesansparole@gmail.com>

7.2 Contributors

- revesansparole <revesansparole@gmail.com>
- ThT12 <ThT12@ThT12.fr>
- Simon Artzet <simon.artzet@gmail.com>

CHAPTER 8

History

CHAPTER 9

creation (TODAY)

- First release on PyPI.

CHAPTER 10

Sources

10.1 src

10.1.1 pkglts package

Subpackages

pkglts.install_env package

Submodules

pkglts.install_env.conda_front_end module

Conda front end to install package in current environment.

pkglts.install_env.conda_front_end.**install**(*name*)

Install a package in the current python environment.

arg:

- name (str): name of the package

Returns whether installation was successful or not

Return type

- (bool)

pkglts.install_env.conda_front_end.**installed_packages**()

Iterate on all packages installed in the current python environment.

Returns (iter of str)

pkglts.install_env.load_front_end module

Factory to load a specific front end.

pkglts.install_env.load_front_end.**get_install_front_end**(*name*)
Load install front end.

Parameters **name** (–) – name of front end to load

Returns loaded python module

Return type

- (module)

pkglts.install_env.pip_front_end module

Pip front end to install package in current environment.

pkglts.install_env.pip_front_end.**install**(*name*)
Install a package in the current python environment.

arg:

- name (str): name of the package

Returns whether installation was successful or not

Return type

- (bool)

pkglts.install_env.pip_front_end.**installed_packages**()
Iterate on all packages installed in the current python environment.

Returns (iter of str)

pkglts.install_env.stdout_front_end module

Simple front end to emulate all environment modification by printing the required operations.

pkglts.install_env.stdout_front_end.**install**(*name*)
Install a package in the current python environment.

arg:

- name (str): name of the package

Returns whether installation was successful or not

Return type

- (bool)

pkglts.install_env.stdout_front_end.**installed_packages**()
Iterate on all packages installed in the current python environment.

Returns (list of str)

Module contents

pkglts.option package

Subpackages

pkglts.option.base package

Submodules

pkglts.option.base.config module

pkglts.option.base.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.base.config.**is_valid_identifier**(*name*)
Check that name is a valid python identifier sort of back port of “”.isidentifier()

pkglts.option.base.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either ‘option’, ‘setup’, ‘install’ or ‘dvlpt’
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.base.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

pkglts.option.base.handlers module

pkglts.option.base.handlers.**environment_extensions**(*cfg*)
Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

Module contents

pkglts.option.conda package

Submodules

pkglts.option.conda.config module

pkglts.option.conda.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

Module contents

pkglts.option.coverage package

Submodules

pkglts.option.coverage.config module

pkglts.option.coverage.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

Module contents

pkglts.option.coveralls package

Submodules

pkglts.option.coveralls.config module

pkglts.option.coveralls.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.coveralls.handlers module

pkglts.option.coveralls.handlers.**environment_extensions**(*cfg*)
Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

Module contents

pkglts.option.data package

Submodules

pkglts.option.data.config module

pkglts.option.data.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

Module contents

pkglts.option.doc package

Submodules

pkglts.option.doc.config module

pkglts.option.doc.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.doc.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'

- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.doc.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

Module contents

pkglts.option.doc.**fmt_badge**(*badge*, *url*, *txt*, *fmt*)

pkglts.option.flake8 package

Submodules

pkglts.option.flake8.config module

pkglts.option.flake8.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

Module contents

pkglts.option.git package

Submodules

pkglts.option.git.config module

pkglts.option.git.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.git.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.git.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

pkglts.option.git.handlers module

pkglts.option.git.handlers.**environment_extensions**(*cfg*)
Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

Module contents

pkglts.option.github package

Submodules

pkglts.option.github.config module

pkglts.option.github.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.github.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.github.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

Module contents

pkglts.option.gitlab package

Submodules

pkglts.option.gitlab.config module

pkglts.option.gitlab.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.gitlab.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.gitlab.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

Module contents

pkglts.option.landscape package

Submodules

pkglts.option.landscape.config module

pkglts.option.landscape.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either ‘option’, ‘setup’, ‘install’ or ‘dvlpt’
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.landscape.handlers module

pkglts.option.landscape.handlers.**environment_extensions**(*cfg*)
Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

Module contents

pkglts.option.license package

Submodules

pkglts.option.license.config module

pkglts.option.license.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.license.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either ‘option’, ‘setup’, ‘install’ or ‘dvlpt’
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.license.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

pkglts.option.license.handlers module

pkglts.option.license.handlers.**environment_extensions**(*cfg*)
Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

pkglts.option.license.handlers.**full_text**(*cfg*)
Generate a license

pkglts.option.license.handlers.**get_tpl_path**(*name*)
Return a path for license template file.

Warnings: Do not test if path exists

Parameters **name** (*str*) – name of license to fetch

Returns (str)

Module contents

pkglts.option.notebook package

Submodules

pkglts.option.notebook.config module

pkglts.option.notebook.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.notebook.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.notebook.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

pkglts.option.notebook.regenerate module

Module contents

pkglts.option.plugin_project package

Submodules

pkglts.option.plugin_project.config module

pkglts.option.plugin_project.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.plugin_project.config.**is_valid_identifier**(*name*)
Check that name is a valid python identifier sort of back port of "".isidentifier()

pkglts.option.plugin_project.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.plugin_project.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

Module contents

pkglts.option.pypi package

Submodules

pkglts.option.pypi.config module

pkglts.option.pypi.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.pypi.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.pypi.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

pkglts.option.pypi.handlers module

pkglts.option.pypi.handlers.**auto_classifiers**(*cfg*)

pkglts.option.pypi.handlers.**environment_extensions**(*cfg*)
Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

Module contents

pkglts.option.pysetup package

Submodules

pkglts.option.pysetup.config module

pkglts.option.pysetup.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.pysetup.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'

- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.pysetup.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

pkglts.option.pysetup.handlers module

pkglts.option.pysetup.handlers.**environment_extensions**(*cfg*)
Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

pkglts.option.pysetup.handlers.**pkg_url**(*cfg*)

pkglts.option.pysetup.handlers.**requirements**(*cfg*, *requirement_name*)
Check all requirements for installed options.

Parameters

- **cfg** (Config) – current package configuration
- **requirement_name** (*str*) – type of requirement 'install', 'dvlpt'

Returns list of required packages names

Return type (list of str)

Module contents

pkglts.option.readthedocs package

Submodules

pkglts.option.readthedocs.config module

pkglts.option.readthedocs.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.readthedocs.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.readthedocs.config.**update_parameters**(*cfg*)

Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

pkglts.option.readthedocs.handlers module

pkglts.option.readthedocs.handlers.**environment_extensions**(*cfg*)

Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

Module contents

pkglts.option.requires package

Submodules

pkglts.option.requires.config module

pkglts.option.requires.config.**require**(*purpose*, *cfg*)

List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.requires.handlers module

pkglts.option.requires.handlers.**environment_extensions**(*cfg*)

Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

Module contents

pkglts.option.sphinx package

Submodules

pkglts.option.sphinx.config module

pkglts.option.sphinx.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.sphinx.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either 'option', 'setup', 'install' or 'dvlpt'
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.sphinx.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

Module contents

pkglts.option.test package

Submodules

pkglts.option.test.config module

pkglts.option.test.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.test.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either ‘option’, ‘setup’, ‘install’ or ‘dvlpt’
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.test.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

Module contents

pkglts.option.tox package

Submodules

pkglts.option.tox.config module

pkglts.option.tox.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either ‘option’, ‘setup’, ‘install’ or ‘dvlpt’
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.tox.handlers module

Module contents

pkglts.option.travis package

Submodules

pkglts.option.travis.config module

pkglts.option.travis.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either ‘option’, ‘setup’, ‘install’ or ‘dvlpt’
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.travis.handlers module

pkglts.option.travis.handlers.**environment_extensions**(*cfg*)
Add more functionality to an environment.

Parameters **cfg** (Config) – current package configuration

Returns any

Return type dict of str

Module contents

pkglts.option.version package

Submodules

pkglts.option.version.config module

pkglts.option.version.config.**check**(*cfg*)
Check the validity of parameters in working environment.

Parameters **cfg** (Config) – current package configuration

Returns list of faulty parameters

Return type (list of str)

pkglts.option.version.config.**require**(*purpose*, *cfg*)
List of requirements for this option for a given purpose.

Parameters

- **purpose** (*str*) – either ‘option’, ‘setup’, ‘install’ or ‘dvlpt’
- **cfg** (Config) – current package configuration

Returns (list of Dependency)

pkglts.option.version.config.**update_parameters**(*cfg*)
Update config with parameters necessary for this option.

Notes: create a section with option name to store params.

Parameters **cfg** (*dict*) – dict of option parameters as seen in pkg_cfg.json

Returns update in place

Return type None

pkglts.option.version.github_versioning module

pkglts.option.version.handlers module

Module contents

Module contents

Submodules

pkglts.config module

pkglts.config_management module

class pkglts.config_management.**Config**(**args*, ***kwds*)
Bases: dict

Object used to store both a templated version of the config as a dict interface its resolution and a jinja2 environment that reflect the config.

installed_options()
List all installed options.

Returns (iter of str)

load_extra()
load option specific handlers.

Returns None

render(*txt*)
Use items in config to render text

Parameters **txt** (*str*) – templated text to render

Returns

same text where all templated parts have been replaced by their values.

Return type (str)

resolve()
Try to resolve all templated items.

Returns None

template()

class pkglts.config_management.**ConfigSection**
Bases: object

Small class to allow accessing parameters using the dot method instead of ['param_name'] method

pkglts.config_management.**get_pkg_config**(*rep='.'*)
Read pkg_cfg file associated to this package.

Parameters **rep** (*str*) – directory to search for info

Returns Config initialized with pkg_config

Return type (*Config*)

pkglts.config_management.**upgrade_pkg_cfg_version**(*pkg_cfg*, *version*)
Upgrade the version of pkg_cfg file from version to version +1

Parameters

- **pkg_cfg** (*dict of str, any*) – package configuration
- **version** (*int*) – current version of file

Returns any): a reference to an updated pkg_cfg

Return type (dict of str

pkglts.config_management.**write_pkg_config**(*cfg*, *rep='.'*)
Store config associated to this package on disk.

Parameters

- **cfg** (Config) – current working config
- **rep** (*str*) – directory to search for info

Returns None

pkglts.data_access module

Set of function to work with resources that are located inside this package data

pkglts.data_access.**get**(*file_name*, *mode='r'*)
Retrieve the content of a given filename located in the data part of this package.

Parameters

- **filename** (–) – name of the file to read
- **mode** (–) – mode to use to read the file either 'r' or 'rb'

Returns content of the file red in 'r' mode

Return type (str)

pkglts.data_access.**get_data_dir**()

pkglts.data_access.**ls**(*dir_name*)
List all files and directories in dir_name located in the data part of this package.

Parameters **dir_name** (–) – name of the directory to walk

Returns

list the content of dir_name without any specific order, items are (entity_name, is_directory)

Return type (list of [str, bool])

pkglts.dependency module

class pkglts.dependency.**Dependency**(*name*, *pkg_mng=None*, *channel=None*)
Bases: object

Simple container to keep track of all the required info to install a dependency.

fmt_requirement()
Format dependency for requirements files

Returns (str)

pkglts.file_management module

pkglts.file_management.**write_file**(*pth*, *content*)
Write the content of a file on a local path and register associated hash for further modification tests.

Parameters

- **pth** (*str*) – path to the new created file
- **content** (*str*) – content to write on disk

Returns:

pkglts.hash_management module

pkglts.hash_management.**compute_hash**(*txt*)
Compute hash summary of a text

Parameters **txt** (*str*) – content to hash

Returns hash key

Return type (str)

pkglts.hash_management.**get_pkg_hash**(*rep='.'*)
Read pkg_hash file associated to this package

Parameters **rep** (*str*) – directory to search for info

Returns

hash map of preserved sections in this package

Return type (dict of str, list)

pkglts.hash_management.**modified_file_hash**(*pth*, *pkg_hash*)
Check whether a file complies with previously stored hash

Parameters

- **pth** (*str*) – path to file to test
- **pkg_hash** (*dict of str, list*) – hash map of preserved sections in this package

Returns whether this file has been modified or not

Return type (bool)

pkglts.hash_management.**pth_as_key**(*pth*)
Normalize path to enable to use them as keys

Parameters **pth** (*str*) – path to normalize

Returns (str)

pkglts.hash_management.**write_pkg_hash**(*pkg_hash*, *rep='.'*)
Store hash associated to this package on disk.

Parameters

- **pkg_hash** (*dict of str, list*) – hash map of preserved sections in this package
- **rep** (*str*) – directory to search for info

Returns None

pkglts.local module

Regroup set of functions that make use of local environment inside a package. Just a way to normalize pre defined paths.

pkglts.local.**init_namespace_dir**(*pth*, *cfg*)
Populate a directory with specific __init__.py for namespace packages.

Parameters

- **pth** (*str*) – path in which to create the files
- **cfg** (Config) – current package configuration

Returns None

pkglts.local.**pkg_full_name**(*cfg*)
Compute name of src dir according to pkgname and namespace in environment

Parameters **cfg** (Config) – current package configuration

Returns (str)

pkglts.local.**src_dir**(*cfg*)
Compute name of src dir according to pkgname and namespace in environment

Parameters **cfg** (Config) – current package configuration

Returns (str)

pkglts.logging_tools module

pkglts.manage module

Contains functions to manage the structure of the package.

Use 'setup.py' for common tasks.

pkglts.manage.**add_option**(*name*, *cfg*)
Add a new option to this package.

Notes: See the list of available options online

Parameters

- **name** (*str*) – name of option to add
- **cfg** (Config) – current package configuration

Returns updated package configuration

Return type (*Config*)

pkglts.manage.**clean**(*rep='.'*)
Thorough cleaning of all arborescence rooting at rep.

Todo: exception list instead of hardcoded one

Parameters **rep** (*str*) – default ".", top directory to clean

Returns None

pkglts.manage.**init_pkg**(*rep='.'*)
Initialise a package in given directory.

Parameters **rep** (*str*) – directory to create pkg into, default current

Returns None

pkglts.manage.**install_example_files**(*option*, *cfg*, *target='.'*)
Install example files associated to an option.

Parameters

- **option** (*str*) – name of option
- **cfg** (Config) – current package configuration
- **target** (*str*) – target directory to write into

Returns whether operation succeeded or not

Return type (bool)

pkglts.manage.**regenerate_option**(*cfg*, *name*, *target='.'*, *overwrite=False*)
Call the regenerate function of a given option

Parameters

- **cfg** (Config) – current package configuration
- **name** – (str) name of option
- **target** – (str) target directory to write into
- **overwrite** (*bool*) – default False, whether or not to overwrite user modified files

Returns None

pkglts.manage.**regenerate_package**(*cfg*, *target='.'*, *overwrite=False*)
Rebuild all automatically generated files.

Parameters

- **cfg** (Config) – current package configuration
- **target** (*str*) – target directory to write into
- **overwrite** (*bool*) – default False, whether or not to overwrite user modified files

Returns None

pkglts.manage_script module

pkglts.manage_script.**action_add**(**args*, ***kwds*)
Add new options in the package.

pkglts.manage_script.**action_clean**(**args*, ***kwds*)
Clean package of all un necessary files.

pkglts.manage_script.**action_clear**(**args*, ***kwds*)
Attempt to free the package from pkglts interactions.

pkglts.manage_script.**action_example**(**args*, ***kwds*)
Install example files associated with options.

pkglts.manage_script.**action_info**(**args*, ***kwds*)
Display info on package for debug purpose.

pkglts.manage_script.**action_init**(**args*, ***kwds*)
Initialize environment for use of pkglts.

pkglts.manage_script.**action_regenerate**(**args*, ***kwds*)
Regenerate all files in the package.

pkglts.manage_script.**action_remove**(**args*, ***kwds*)
Remove options from the package.

pkglts.manage_script.**action_update**(**args*, ***kwds*)
Check if a new version of pkglts is available.

pkglts.manage_script.**main**()

pkglts.manage_tools module

Specific helper function for manage script

pkglts.manage_tools.**check_option_parameters**(*name*, *cfg*)
Check that the parameters associated to an option are valid.

Try to import Check function in option dir.

Parameters

- **name** (*str*) – option name
- **cfg** (Config) – current package configuration

pkglts.manage_tools.**ensure_installed_packages**(*requirements*, *msg*, *cfg*)
Ensure all packages in requirements are installed.

If not, ask user permission to install them.

Parameters

- **requirements** (*list of str*) – list of package names to install if needed
- **msg** (*str*) – error message to print
- **cfg** (Config) – current package configuration

Returns whether all required packages are installed or not

Return type (bool)

pkglts.manage_tools.**regenerate_dir**(*src_dir*, *tgt_dir*, *cfg*, *overwrite_file*)
Walk all files in src_dir and create/update them on tgt_dir

Parameters

- **src_dir** (*str*) – path to reference files
- **tgt_dir** (*str*) – path to target where files will be written
- **cfg** (Config) – current package configuration
- **overwrite_file** (*dict of str, bool*) – whether or not to overwrite some files

Returns hash key of preserved sections

Return type (dict of str, map)

pkglts.manage_tools.**update_opt**(*name*, *cfg*)
Update an option of this package.

Notes: If the option does not exists yet, add it first. See the list of available option online

Parameters

- **name** (*str*) – name of option to add
- **cfg** (Config) – current package configuration

pkglts.option_tools module

Some helpers for options

class pkglts.option_tools.**Option**
Bases: object

Base class to store information associated with an option

check(**args*, ***kwds*)

environment_extensions(**args*, ***kwds*)

files_dir()

from_entry_point(*ep*)

regenerate(**args*, ***kwds*)

require(**args*, ***kwds*)

update_parameters(*cfg*)

pkglts.option_tools.**find_available_options**()

pkglts.option_tools.**get_user_permission**(*action_name*, *default_true=True*)

pkglts.templating module

class pkglts.templating.**TplBlock**
Bases: object

Simple container for templated blocks.

pkglts.templating.**parse_source**(*txt*)
Parse text to find preserved blocks

Parameters **txt** (*str*) – full text to parse

Returns

ordered list of blocks:

- block_id (or None if content is not preserved
- line start before start for preserved content
- content
- line start before end for preserved content

Return type (list of [str, str, str, str])

pkglts.templating.**render**(*cfg*, *src_pth*, *tgt_pth*)
Render src_pth templated file into tgt_pth

Notes: keeps 'preserved' block structure

Parameters

- **cfg** (Config) – current package configuration

- **src_pth** (*str*) – path to reference file
- **tgt_pth** (*str*) – path to potentially non existent yet target file

Returns key, cnt for preserved blocks

Return type (list of [str, str])

pkglts.tree_ascii_fmt module

pkglts.tree_ascii_fmt.**fmt_tree**(*dname*)

pkglts.tree_ascii_fmt.**nn**(*cfg*, *pth*)

pkglts.tree_ascii_fmt.**tree**(*dname*, *padding*, *txt*)

pkglts.version module

pkglts.version.**major = 2**
(int) Version major component.

pkglts.version.**minor = 1**
(int) Version minor component.

pkglts.version.**post = 0**
(int) Version post or bugfix component.

Module contents

10.1.2 pkglts_data package

Subpackages

pkglts_data.base package

Module contents

Module contents

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

T

U

W