buildstrap Documentation

Release 0.4.0

Bernard ‘Guyzmo‘ Pratz

Jun 26, 2016

Contents

For those who want to hack on any project without having to hack around your shell environment, mess with your python tools, pollute your home dotfiles, buildstrap is for you.

This project will bring the power of buildout, by generating in a simple command all you need to setup a buildout configuration, that will then create a self contained python environment for all your hacking needs.

It's as simple as:

```
% git clone https://github.com/guyzmo/buildstrap
% cd buildstrap
% buildstrap run buildstrap requirements.txt
...
% bin/buildstrap --version
0.1.1
```

What is being done here, is that you tell buildstrap the package's name, and the requirements files to parse, and it will generate the the following `buildout.cfg` file:

```
[buildout]
newest = false
parts = buildstrap
develop = .
eggs-directory = ${buildout:directory}/var/eggs
develop-eggs-directory = ${buildout:directory}/var/develop-eggs
develop-dir = ${buildout:directory}/var/develop
parts-directory = ${buildout:directory}/var/parts
requirements = ${buildout:develop}/requirements.txt

[buildstrap]
eggs = ${buildout:requirements-eggs}
    buildstrap
recipe = zc.recipe.egg
```

That file is then used to configure buildout so it creates the environment in your project's directory. You'll find all your dependencies downloaded into `/var` , and all the scripts you need populated in `/bin` .

So, it's only two directories to add to your `.gitignore` , and to delete when you want to make your workspace clean again. Then you can choose to either keep (and eventually tweak) your `buildout.cfg` file, or throw it away.

Yes, it's as easy as it sounds!

CHAPTER 1

Buildstrap: generate a buildout config for any *env project

There's pyenv, pyvenv, venv, virtualenv… and who knows how many other ways to deal with development of python programs in a per-project self-contained manner.

While most of the python community tried to keep up, and got their shell configuration or global pip changing regularly, some have been quietly enjoying python development the same way for the last ten years, using buildout for their development.

Though, it's a fact that buildout is not the standard way to do things, even if it's a very convenient tool. So to keep your repositories compatible with most *env tools available — or get buildout with other projects. I wrote this tool to make it easy to create a buildout environment within the project.

CHAPTER 2

Quickstart Guide

Here we'll see the most common usages, and refer to the full documentation for more details.

2.1 Usage

when you got a repository that has requirements files, at the root of your project's directory, call buildstrap using:

```
% buildstrap run project requirements.txt
```

where `project` as second argument is the name of the package as you've set it up in your `setup.py` — and as you'd import it from other python code.

Running that command will generate the `buildout.cfg` file, and run `buildout` in your current directory. Then you'll find all your scripts available in the newly created `bin` directory of your project.

If you have several `requirements.txt` files, depending on the task you want to do, it's easy:

```
% buildstrap run project -p pytest -p sphinx requirements.txt requirements-test.txt␣
↪requirements-doc.txt
```

which will create three sections in your `buildout.cfg` file, and get all the appropriate dependencies.

Here's a real life example:

```
% git hub clone kennethreitz/requests    # cf 'Nota Bene'
% cd requests
% buildstrap run requests requirements.txt
...
% bin/py.test
... (look at the tests result)
% bin/python3
>>> import requests
>>>
```

or another one:

```
% git hub clone jkbrzt/httpie            # cf 'Nota Bene'
% cd httpie
% buildstrap run httpie requirements-dev.txt
...
% bin/py.test
... (look at the tests result)
```

```
% bin/http --version
1.0.0-dev
```

2.2 Installation

it's as easy as any other python program:

```
% pip install buildstrap
```

or from the sources:

```
% git hub clone guyzmo/buildstrap
% cd buildstrap
% python3 setup.py install
```

2.3 Development

for development you just need to do:

```
% pip install buildstrap
% git clone https://github.com/guyzmo/buildstrap
% cd buildstrap
% builstrap run buildstrap -p pytest -p sphinx requirements.txt requirement-test.txt␣
↪requirement-doc.txt
...
% bin/buildstrap
```

Yeah, I'm being evil here

You can have a look at the sources documentation.

2.4 Nota Bene

You might wonder where does the `git hub clone` command comes from, and I'm using here another project I wrote: guyzmo/git-repo.

Simply put, `git hub clone user/project` is equivalent to `git clone https://github.com/user/project`.

2.5 License

```
Copyright © 2016 Bernard `Guyzmo` Pratz <guyzmo+buildstrap+pub@m0g.net>
This work is free. 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 the LICENSE file for more details.
```

CHAPTER 3

Buildout configuration

Before going into more details, let's have briefly a look at a `buildout.cfg` configuration. Each section of the configuration file are called `part` s in buildout slang. The part configures a directive to run using a recipe. There are many recipes you can lookup, but in a `buildout.cfg` freshly baked by buildstrap, you'll only see two:

- `zc.recipe.egg` : which takes care of downloading and installing dependencies into the self-contained environment ;
- `gp.vcsdevelop` : which parses a `requirements.txt` file and exposes the dependencies, so `zc.recipe.egg` can do its job (in the context of buildstrap).

Then, to setup the environment, there's a section named `[buildout]` that contains everything needed to setup the self contained environment, like the list of parts to run (remove one from there and it'll be ignored), the paths to the used directories…

Once the buildout configuration file, `buildout.cfg` has been generated, you can tweak it as much as you like to suit your needs. Buildout is much more than just setting up a self contained environment!

If you want to read more about buildout, check its documentation, or for more in depth info, check `buildout.cfg` manual.

CHAPTER 4

Usage

```
Usage: buildstrap [-v...] [options] [run|show|debug|generate] [-p part...]<package>
↪<requirements>...

Options:
    run                            run buildout once buildout.cfg has been generated
    show                           show the buildout.cfg (same as using `-o -`)
    debug                          print internal representation of buildout config
    generate                       create the buildout.cfg file (default action)
    <package>                      use this name for the package being developed
    <requirements>                 use this requirements file as main requirements
    -p,--part <part>               choose part template to use (use "list" to show all)
    -i,--interpreter <python>      use this python version
    -o,--output <buildout.cfg>     file to output [default: buildout.cfg]
    -r,--root <path>               path to the project root (where buildout.cfg will
                be generated) (defaults to ./)
    -s,--src <path>                path to the sources (default is same as root path)
                relative to the root path if not absolute
    -e,--env <path>                path to the environment data [default: var]
                relative to directory if not absolute
    -b,--bin <path>                path to the bin directory [default: bin]
                relative to directory if not absolute
    -f,--force                     force overwrite output file if it exists
    -c,--config <path>             path to the configuration directory
                [default: ~/.config/buildstrap]
    -v,--verbose                   increase verbosity
    -h,--help                      show this message
    --version                      show version
```

CHAPTER 5

Multiple requirements.txt

Many projects offer multiple `requirements.txt` files, one for each task of the development cycle (which usually are running, testing, documenting).

Well, just tell buildstrap what the extra requirements are:

```
% buildstrap run buildstrap -p pytest -p sphinx requirements.txt requirements-doc.txt␣
↪requirements-test.txt
```

and that will generate the following buildout.cfg configuration:

```
[buildout]
newest = false
parts = buildstrap
        pytest
        sphinx
package = buildstrap
extensions = gp.vcsdevelop
develop = .
eggs-directory = ${buildout:directory}/var/eggs
develop-eggs-directory = ${buildout:directory}/var/develop-eggs
parts-directory = ${buildout:directory}/var/parts
develop-dir = ${buildout:directory}/var/develop
bin-directory = ${buildout:directory}/bin
requirements = ${buildout:develop}/requirements.txt
        ${buildout:develop}/requirements-doc.txt
        ${buildout:develop}/requirements-test.txt

[buildstrap]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
        buildstrap

[pytest]
arguments = ['--cov={}/{}'.format('${buildout:develop}', package) for package in '$
↪{buildout:pack
age}'.split(',')] \
        +['--cov-report', 'term-missing', 'tests']+sys.argv[1:]
eggs = ${buildout:requirements-eggs}
recipe = zc.recipe.egg

[sphinx]
eggs = ${buildout:requirements-eggs}
source = ${buildout:directory}/doc
```

```
recipe = collective.recipe.sphinxbuilder
build = ${buildout:directory}/doc/_build
```

and you'll find all the tools you'll need in bin:

```
% ls bin
buildout    cm2html   cm2man        cm2xetex  py.test      sphinx         sphinx-
↪autogen  sphinx-quickstart
buildstrap  cm2latex  cm2pseudoxml  cm2xml    py.test-2.7  sphinx-apidoc  sphinx-build
```

Multiple packages

Some projects will include several packages in the sources, so to support that, just list all your packages as a comma seperated list, and they will all be included:

```
% buildstrap show dent,prefect,beeblebox requirements.txt
[buildout]
newest = false
parts = dent
package = dent prefect beeblebox
extensions = gp.vcsdevelop
develop = .
eggs-directory = ${buildout:directory}/var/eggs
develop-eggs-directory = ${buildout:directory}/var/develop-eggs
parts-directory = ${buildout:directory}/var/parts
develop-dir = ${buildout:directory}/var/develop
bin-directory = ${buildout:directory}/bin
requirements = ${buildout:develop}/requirements.txt

[dent]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
        dent
        prefect
        beeblebox
```

6.1 Control the output

If you want to only generate the `buildout.cfg` file, simply use buildstrap with no subcommand, and you'll get it in your current directory!

```
% buildstrap slartibartfast requirements.txt
% cat buildout.cfg
[buildout]
newest = false
parts = slartibartfast
package = slartibartfast
extensions = gp.vcsdevelop
develop = .
eggs-directory = ${buildout:directory}/var/eggs
develop-eggs-directory = ${buildout:directory}/var/develop-eggs
parts-directory = ${buildout:directory}/var/parts
```

```
develop-dir = ${buildout:directory}/var/develop
bin-directory = ${buildout:directory}/bin
requirements = ${buildout:develop}/requirements.txt

[slartibartfast]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
        slartibartfast
```

but if you want to just test the command and print the configuration to stdout, without it doing nothing, use the `show` subcommand:

```
% buildstrap show slartibartfast requirements.txt
[buildout]
newest = false
parts = slartibartfast
package = slartibartfast
extensions = gp.vcsdevelop
develop = .
eggs-directory = ${buildout:directory}/var/eggs
develop-eggs-directory = ${buildout:directory}/var/develop-eggs
parts-directory = ${buildout:directory}/var/parts
develop-dir = ${buildout:directory}/var/develop
bin-directory = ${buildout:directory}/bin
requirements = ${buildout:develop}/requirements.txt

[slartibartfast]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
        slartibartfast
```

and if you want to write the `buildout.cfg` as another file, you can either redirect the show command with a pipe, or use the `--output` argument:

```
% buildstrap -o foobar.cfg slartibartfast requirements.txt
% cat foobar.cfg
[buildout]
newest = false
parts = slartibartfast
package = slartibartfast
extensions = gp.vcsdevelop
develop = .
eggs-directory = ${buildout:directory}/var/eggs
develop-eggs-directory = ${buildout:directory}/var/develop-eggs
parts-directory = ${buildout:directory}/var/parts
develop-dir = ${buildout:directory}/var/develop
bin-directory = ${buildout:directory}/bin
requirements = ${buildout:develop}/requirements.txt

[slartibartfast]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
        slartibartfast
```

N.B.: the show command is equivalent to `--output -`.

CHAPTER 7

Configure the path

For your project, there are three important path to configure:

- where your project root is,
- where your sources are (within your project),
- where your environment will be.

When you're using buildstrap on a project, the default are safe, as long as you're running while you're doing it within the sources of the project. Then what you'll have is:

- `root_path` → '.'
- `src_path` → `{root_path}` → '.'
- `env_path` → `{root_path}/var` → './var'
- `bin_path` → `{root_path}/bin` → './bin'

But sometimes, you want to change the defaults, for the best (or the worst — most often, the worst, though).

So, you can set all those paths to values other than the default, and have it all in a very different setup than the default.

7.1 Root path: `--root`

The project's root is where typically all other paths are being relative to. It's where you'll expect to find the `buildout.cfg` file, and where the environment directory will be.

When passed, it's setting up the `directory` directive of the `buildout.cfg` file, otherwise it's keeping the default.

```
% buildstrap -r /tmp/buildstrap-env/ show buildstrap requirements.txt
[buildout]
newest = false
parts = buildstrap
package = buildstrap
extensions = gp.vcsdevelop
directory = /tmp/buildstrap-env/
develop = .
eggs-directory = ${buildout:directory}/var/eggs
develop-eggs-directory = ${buildout:directory}/var/develop-eggs
parts-directory = ${buildout:directory}/var/parts
develop-dir = ${buildout:directory}/var/develop
bin-directory = ${buildout:directory}/bin
```

```
requirements = ${buildout:develop}/requirements.txt

[buildstrap]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
        buildstrap
```

7.2 Sources path: `--src`

Though, if you change the root directory, chances are (like in the former example) that it won't be where your sources are. Then, running `buildout` will end up in throwing an exception:

```
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/builstrap-env/./setup.py
↪'
```

The source path is where you'll have your `setup.py` file that defines your project. So, if your `setup.py` is not at the root of your project, you definitely want to use the `--src` argument.

```
% buildstrap -r /tmp/buildstrap-build -s `pwd`/buildstrap show buildstrap␣
↪requirements.txt
[buildout]
newest = false
parts = buildstrap
package = buildstrap
extensions = gp.vcsdevelop
directory = /tmp
develop = /absolute/path/to/buildstrap
eggs-directory = ${buildout:directory}/var/eggs
develop-eggs-directory = ${buildout:directory}/var/develop-eggs
parts-directory = ${buildout:directory}/var/parts
develop-dir = ${buildout:directory}/var/develop
bin-directory = ${buildout:directory}/bin
requirements = ${buildout:develop}/requirements.txt

[buildstrap]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
        buildstrap
```

Nota Bene: if you do not want to use a path relative to the `root` path, then use an absolute path, or you'll have surprises! As you can see in the example above the path is made absolute by using the `pwd` command.

So running this command with buildout will do:

```
% buildout
Creating directory '/tmp/buildstrap-build/var/eggs'.
Getting distribution for 'gp.vcsdevelop'.
warning: no previously-included files matching '*' found under directory 'docs/_build'
Got gp.vcsdevelop 2.2.3.
Creating directory '/tmp/buildstrap-build/bin'.
Creating directory '/tmp/buildstrap-build/var/parts'.
Creating directory '/tmp/buildstrap-build/var/develop-eggs'.
Develop: '/home/guyzmo/Workspace/Projects/buildstrap'
Getting distribution for 'zc.recipe.egg>=2.0.0a3'.
Got zc.recipe.egg 2.0.3.
```

```
Unused options for buildout: 'package'.
Installing buildstrap.
Generated script '/tmp/buildstrap-build/bin/buildout'.
Generated script '/tmp/buildstrap-build/bin/buildstrap'.
```

7.3 Environment path: `--env`

As seen in the previous example, the script is generating a bunch of directories used for setting up the environment in `{root_path}/var/`. You might want them to be named differently, so they're not seen in listings for example:

```
% buildstrap -r /tmp -s `pwd`/buildstrap -e .var show buildstrap requirements.txt
[buildout]
newest = false
parts = buildstrap
package = buildstrap
extensions = gp.vcsdevelop
directory = /tmp
develop = /home/guyzmo/Workspace/Projects/buildstrap/buildstrap
eggs-directory = ${buildout:directory}/.var/eggs
develop-eggs-directory = ${buildout:directory}/.var/develop-eggs
parts-directory = ${buildout:directory}/.var/parts
develop-dir = ${buildout:directory}/.var/develop
bin-directory = ${buildout:directory}/bin
requirements = ${buildout:develop}/requirements.txt

[buildstrap]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
        buildstrap
```

or you might want to put it at any other place, by using an absolute path:

```
% buildstrap -r /tmp -s `pwd`/buildstrap -e /tmp/buildstrap-var show buildstrap␣
↪requirements.txt
[buildout]
directory = /tmp
develop = /home/guyzmo/Workspace/Projects/buildstrap/buildstrap
eggs-directory = /tmp/buildstrap-var/eggs
develop-eggs-directory = /tmp/buildstrap-var/develop-eggs
parts-directory = /tmp/buildstrap-var/parts
develop-dir = /tmp/buildstrap-var/develop
bin-directory = ${buildout:directory}/bin
...
```

7.4 Bin path: `--bin`

Finally, you might not like the default of having the `bin` directory at the `root` path position, so you can put it within var the following way:

```
% buildstrap -b var/bin show buildstrap requirements.txt
[buildout]
develop = .
eggs-directory = ${buildout:directory}/var/eggs
```

```
develop-eggs-directory = ${buildout:directory}/var/develop-eggs
parts-directory = ${buildout:directory}/var/parts
develop-dir = ${buildout:directory}/var/develop
bin-directory = ${buildout:directory}/var/bin
...
```

or same as before, to somewhere other place non relative to the sources:

```
% buildstrap -r /tmp -s `pwd`/buildstrap -e /tmp/buildstrap-var -b /tmp/buildstrap-
↪bin show buildstrap requirements.txt
[buildout]
develop = .
directory = /tmp/buildstrap-env/
eggs-directory = /tmp/buildstrap-var/eggs
develop-eggs-directory = /tmp/buildstrap-var/develop-eggs
parts-directory = /tmp/buildstrap-var/parts
develop-dir = /tmp/buildstrap-var/develop
bin-directory = /tmp/buildstrap-bin
...
```

buildstrap package

8.1 Submodules

8.2 buildstrap.buildstrap module

Buildstrap: generate and run buildout in your projects

```
Usage: {} [-v...] [options] [run|show|debug|generate] [-p part...]<package>
 ↪<requirements>...

Options:
    run                         run buildout once buildout.cfg has been generated
    show                        show the buildout.cfg (same as using `-o -`)
    debug                       print internal representation of buildout config
    generate                    create the buildout.cfg file (default action)
    <package>                   use this name for the package being developed
    <requirements>              use this requirements file as main requirements
    -p,--part <part>            choose part template to use (use "list" to show all)
    -i,--interpreter <python>   use this python version
    -o,--output <buildout.cfg>  file to output [default: buildout.cfg]
    -r,--root <path>            path to the project root (where buildout.cfg will
                                be generated) (defaults to ./)
    -s,--src <path>             path to the sources (default is same as root path)
                                relative to the root path if not absolute
    -e,--env <path>             path to the environment data [default: var]
                                relative to directory if not absolute
    -b,--bin <path>             path to the bin directory [default: bin]
                                relative to directory if not absolute
    -f,--force                  force overwrite output file if it exists
    -c,--config <path>          path to the configuration directory
                                [default: ~/.config/buildstrap]
    -v,--verbose                increase verbosity
    -h,--help                   show this message
    --version                   show version
```

For more detailed help, please read the documentation on https://readthedocs.org/buildstrap

class `buildstrap.buildstrap.` **`ListBuildout`**

Bases: `list`

Makes it possible to print a list the way buildout expects it.

Because buildout uses a custom built parser for parsing ini style files, that has a major difference in list handling from the standard config parser.

The standard configparser doesn't anything about lists, and outputs python's internal representation of strings: `['a','b','c']` as values. And the standard configparser consider multiline values as a multiline string.

On the other hand, buildout considers multiline values as lists, one value per line.

This class uses a context manager to define the behaviour of the string conversion method. The default behaviour is the same as the standard list. But when within the context of the `generate_context` method, it prints lists as multiline string, one value per line, the way buildout expects it.

classmethod **`generate_context`** `()`

Context manager to change the string conversion behaviour on this class

`buildstrap.buildstrap.`**`build_part_buildout`**`(`*root_path=None*, *src_path=None*, *env_path=None*, *bin_path=None*`)`

Generates the buildout part

This part is the entry point of a buildout configuration file, setting up general values for the environment. Here we setup paths and defaults for buildout's behaviour. Please refer to buildout documentation for more.

This will output a buildout header that can be considered as a good start:

```
[buildout]
newest=false
parts=
package=
extensions=gp.vcsdevelopc
directory=.
develop=${buildout:directory}
eggs-directory=${buildout:directory}/var/eggs
develop-eggs-directory=${buildout:directory}/var/develop-eggs
develop-dir=${buildout:directory}/var/develop
parts-directory=${buildout:directory}/var/parts
requirements=
```

Parameter `root_path` will change the path to the project's root, which is where the enviroment will be based on. If you're placing the `buildout.cfg` file in another directory than the root of the project, set it to the path that can get you from the buildout.cfg into the project, and it will all work ok.

Parameter `src_path` will change the path to the sources, so if you've got your sources in `./src`, you can set it up to src and it will generate:

```
develop=./src
```

Beware that all non-absolute paths given to `src_path` are relative to the `root_path`.

For parameter `bin_path` and `env_path`, it will respectively change path to the generated `bin` directory and `env` directory, after running buildout.

Parameters

- **`root_path`** – path string to the root of the project (from which all other paths are relative to)
- **`src_path`** – path string to the sources (where `setup.py` is)
- **`env_path`** – path string to the environment (where dependencies are downloaded)
- **`bin_path`** – path string to the runnable scripts

Returns the buildout part as a dict

buildstrap.buildstrap.**build_part_target** (*target*, *packages=[]*, *interpreter=None*)

Generates a part to run the currenctly develop package

This will output a part, that will make a script based on the current package developed, using the `zc.recipe.egg` recipe, to populate the environment. The generated part follows the following template:

```
[<target>]
recipe=zc.recipe.egg
eggs=<package>
interpreter=<interpreter>
```

If no `packages` argument is given, the list only contains the reference to the requirements egg list, otherwise the list of packages gets appended. If no interpreter argument is given, the directive is ignored.

Parameters

- **target** – name to be used for the part
- **interpreter** – if given, setup the interpreter directive, using the name of a python interpreter as a string.
- **packages** – if given, adds that package to the list of requirements.

Raises `TypeError` – if the packages is not a list of string

Returns dict representation of the part

buildstrap.buildstrap.**build_part_template** (*name*, *config_path*)

Creates a part out of a template file

Will resolve a part file based on its name, by looking through both package's static directory, and through user defined configuration path.

The template file will feature a section (which name is the same as the file name) and will be parsed, and then added to the buildout file *as is*. It will also be named with the `.part.cfg` extension.

Parameters

- **name** – name of the template file (without extension)
- **config_path** – directory where to look for the template file

Returns dict representation of a part

Raises FileNotFoundError if no template can be found.

buildstrap.buildstrap.**build_parts** (*packages*, *requirements*, *part_templates=[]*, *interpreter=None*, *config_path=None*, *root_path='.'*, *src_path=None*, *env_path=None*, *bin_path=None*)

Builds up the different parts of the buildout configuration

this is the workhorse of this code. It will build and return an internal dict representation of a buildout configuration, following the values given by the arguments. The buildout configuration can be seen as a succession of parts, each one being a section in the configuration file. For more, please refer to buildout's documentation.

First, it generates the `[buildout]` part within the dict representation. Within it, it will setup the `packages` value so we keep track of which packages you want to build, the `requirements` value will be used to find and download all the eggs that are needed as dependencies. the `parts` list will keep track of each generated part, only one part being generated for the code under development (even if there are several packages).

The first argument will define the first part's name (the one that will be used to generate a script if an entry point has been defined within the `setup.py`). Thus, it will append the package name to the list of packages within the `[buildout]` section, and be added to the list of eggs that will be run:

```
[buildout]
package = marvin
parts = marvin
...


[marvin]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
     marvin
```

The second argument is the list of requirements to be parsed and fed to `gp.vcsdevelop` so it can work out downloading all your dependencies:

```
[buildout]
requirements = requirements.txt
...
```

Both can be lists (or comma separated list — as a *string*) of package names and requirements files, so if you give packages and requirements being respectively:

- `dent,prefect,beeblebrox` and
- `requirements.txt,requirements-dev.txt`

it will generate:

```
[buildout]
...
parts = marvin
package = marvin prefect beeblebrox
requirements = requirements.txt
        requirements-dev.txt

[marvin]
recipe = zc.recipe.egg
eggs = ${buildout:requirements-eggs}
     marvin
```

The third argument enables to load a part template. It will load the part from the static path within the package, or from `config_path` , which defaults to the user's home config directory.

Parameters

- **`packages`** – the list of packages to target as first part (list or comma separated string)
- **`requirements`** – the list of requirements to target as first part (list or comma separated string)
- **`part_templates`** – list of templates to load
- **`interpreter`** – string name of the python interpreter to use
- **`config_path`** – path string to the configuration directory where to find the template parts files.
- **`root_path`** – path string to the root of the project (from which all other paths are relative to)
- **`src_path`** – path string to the sources (where `setup.py` is)
- **`env_path`** – path string to the environment (where dependencies are downloaded)

- **`bin_path`** – path string to the runnable scripts

Returns OrderedDict instance configured with all parts.

`buildstrap.buildstrap.`**`buildstrap`**`(`*args*`)`

Parses the command line arguments, build the parts, generate the config and runs buildout

refer to the __doc__ of this module for all arguments.

Parameters **`args`** – arguments to parse

Returns 0 on success, 1 otherwise

`buildstrap.buildstrap.`**`generate_buildout_config`**`(`*parts*, *output*, *force=False*`)`

Generates the buildout configuration

Using the custom `ListBuildout` context, lists will be printed as multilines. If output is set to `-` it will print to stdout the file.

Parameters

- **`parts`** – dict based representation of the buildout file to generate
- **`output`** – name of the file to output
- **`force`** – if set, it won't care whether the file exists

Raises `FileExistsError` – when a file already exists.

`buildstrap.buildstrap.`**`list_part_templates`**`(`*config_path*`)`

Iterates over the available part templates

Will get through both package's templates path and user config path to check for `.part.cfg` files.

Parameters **`config_path`** – path to the user's part template directory

Returns iterator over the list of templates

`buildstrap.buildstrap.`**`run`**`()`

Parses arguments, gets current command name and version number

8.3 Module contents

Index