ape Documentation

Release 0.4.0

Hendrik Speidel

July 31, 2015

Contents

1 Overview **3**

2 Contents **5**

2.1 Installation 5
2.2 Modes of Operation 6
2.3 Feature modules 7
2.4 Task functions 7
2.5 Tutorial 9
2.6 Changelog 9

3 Indices and tables **11**

ape is a make/rake/ant/fab-like system with support for FOSD.

ape is short for "A Productive Environment". It provides a way to organize tasks in feature modules. Feature modules can introduce new tasks and refine tasks that have been introduced by other features.

Tasks are defined as simple python functions. ape makes these task functions available from the command line and provides usage information by extracting the functions' docstrings.

CHAPTER 1

Overview

`make` looks for a `Makefile` in your current directory that defines the available tasks. In Contrast, `ape` composes the available tasks from a list of selected feature modules.

`ape` provides two basic modes of operation, that make different assumptions on how features are selected:

- in *standalone mode* (default) features need to be placed somewhere on the `PYTHONPATH`.
- in *container mode* `ape` provides a set of conventions to organize features and products in a directory structure.

For details, see *Modes of Operation*.

Specifying tasks is really simple: you implement your task as a simple python function. Other features can contain refinements for tasks. This way, it is possible to adapt the behaviour of specific tasks depending on the feature selection. For details, see *Task functions*.

CHAPTER 2

Contents

2.1 Installation

There are multiple options to get ape to run on your system. When using container mode, `ape` is installed in a virtualenv inside the container root. This is the recommended way to use `ape`.

If you need to start over, simply delete the container root and recreate it again.

2.1.1 Bootstrapping a container mode environment

Make sure you have `pip` and `virtualenv` installed. If not, try:

$ easy_install pip $ pip install virtualenv

On Debian/Ubuntu, these are also available as system packages `python-pip` and `python-virtualenv`.

Fetch the script to bootstrap `ape` in container mode:

```
$ wget https://raw.github.com/henzk/ape/master/bin/bootstrape
$ chmod 0755 bootstrape
```

Now, run `bootstrape` to create the container structure in a folder called `aperoot`(change this to your liking):

```
$ bootstrape aperoot
```

To install the development version, run this instead:

$ bootstrape –dev aperoot

After the process has finished a folder `aperoot` is available. It contains all necessary files and dependencies. A virtualenv has been created at `aperoot/_ape/venv`. To activate the container mode, source the activation script for `ape` in your shell (requires bash):

```
$ . aperoot/_ape/activape
```

Congratulations! `ape` is now installed and container mode is activated.

Note: The `bootstrape` script creates the necessary folder structure, creates a virtualenv, and installs `ape` into that environment. There, the latest stable version of `ape` is used — even if you used the `bootstrape` script of another version!

To use a custom version of `ape` in container mode, simply uninstall `ape` from the virtualenv and install your custom version instead.

2.1.2 Installing ape globally

Warning: The preferred method of installing ape is to use bootstrape as described above. Installing ape globally means you have to create the ape root directory manually. Use this only if you want to hack on ape.

To install the latest version of `ape`, use `pip`:

```
$ pip install ape
```

To get the development version of `ape` directly from github, use:

```
$ pip install git+https://github.com/henzk/ape.git#egg=ape
```

Now, `ape` should be available on your `PATH`:

```
$ ape
Error running ape:
Either the PRODUCT_EQUATION or PRODUCT_EQUATION_FILENAME environment variable needs to be set!
```

2.2 Modes of Operation

container mode

In container mode, `ape` manages your features and products in a directory structure. It provides special tasks to switch between products.

standalone mode

In standalone mode, `ape` checks the environment variable `PRODUCT_EQUATION` to get the list of features in composition order. Use container mode! This is for development of ape.

2.2.1 Container Mode

In container mode, `ape` manages your features and products in a directory structure. It provides special tasks to switch between products.

All files managed by `ape` are placed inside a directory that we will refer to as `APE_ROOT_DIR` in the following. If you followed the installation instructions, this folder is called `aperoot`.

It contains:

- a directory named `_ape`. This directory contains the activation script and other global ressources like the global virtualenv.
- multiple *SPL Containers*. These are installations of specific versions of software product lines you manage using `ape`.

Tasks in container mode

cd *poi*

cd to a container or product inside a container. `poi` is a string in one of these formats:

- `<container_name>` e.g. `ape cd mycontainer`
- `<container_name>:<product_name>` e.g. `ape cd mycontainer:myproduct`

switch *poi*

activate the environment of product specified by `poi` `poi` is a string in this format:

- `<container_name>:<product_name>` e.g. `ape switch mycontainer:myproduct`

teleport *poi*

`ape switch` and `ape cd` in one operation.

`poi` is a string in this format:

- `<container_name>:<product_name>` e.g. `ape teleport mycontainer:myproduct`

Since `teleport` is quite long, and it's all about productivity, `zap` is available as an alias for `teleport` ;)

2.2.2 Standalone Mode

In standalone mode, `ape` checks the environment variable `PRODUCT_EQUATION` to get the list of features in composition order.

It needs to contain the names of the features seperated by spaces, e.g. `"basic_tasks extra_tasks my_adjustments"`. For details on how features are specified, see *Feature modules*.

Feature modules need to be placed on the `PYTHONPATH` so `ape` can find them. In container mode, `ape` can manage that for you.

2.3 Feature modules

2.3.1 Layout

Feature modules are Python Packages that contain a module called `tasks`, so a minimal feature called `myfeature` would look like this:

```
myfeature/
    __init__.py
    tasks.py
```

Feature modules need to be placed on the `PYTHONPATH`.

When using container mode, this part is managed for you.

2.4 Task functions

Tasks are python functions that are typically defined in the `tasks` module of a feature. Other features can refine tasks to adapt their behaviour. `ape` automatically creates a command line parser for every task, by inspecting the function signature.

`ape` concentrates solely on task definition, refinement, and invokation from the command line. It does not contain any helpers for implementing your tasks — there are plenty good packages out there to support you with that:

- `subprocess`, `os`, `sys` in the Python standard library
- http://amoffat.github.com/sh/
- Fabric, https://github.com/sebastien/cuisine
- ... lots of others

2.4.1 Introducing tasks

Tasks are introduced by implementing the task as a python function in your feature's `tasks` module. The function must be decorated with `ape.tasks.register` to become a task:

```
#myfeature/tasks.py

from ape import tasks

@tasks.register
def mynewtask(a, b, c=1):
    """description of mynewtask"""
    print a, b, c
```

To make the task available from the command line, we need to select `myfeature` by setting the `PRODUCT_EQUATION` shell variable:

```
$ export PRODUCT_EQUATION="myfeature"
```

Now, make sure `mynewtask` is listed in the output of:

```
$ ape help
```

Finally, let's invoke it:

```
$ ape mynewtask 1 2 --c 5
1 2 5
$ ape mynewtask 1 2
1 2 1
```

2.4.2 Refining a task

Here, we refine `mynewtask` in another feature. To do that, we need to specify a higher order function called `refine_mynewtask` that returns the refined task function. The refined function can access the original implementation as `original`.

```
#anotherfeature/tasks.py

def refine_mynewtask(original):
    def mynewtask(a, b, c=7):
        """updated description of mynewtask"""
        print 'refined task'
        original(a, b, c=c)
    return mynewtask
```

So, to refine a task, we need to define a "factory" that accepts the original task as parameter and returns a wrapper —the refined task. The factory must be named `refine_` followed by the name of the task to refine — in the example above, this results in `refine_mynewtask`. When the feature composer encounters this function, it applies the refinement.

`ape` uses `featuremonkey` to compose the feature modules. For a more detailed description on the composition process, please see the `featuremonkey` documentation at http://featuremonkey.readthedocs.org.

2.4.3 Calling other tasks

Often, tasks need to call other tasks. Functions decorated with `ape.tasks.register` cannot be called directly. This is a safety mechanism to protect against calling partially composed tasks.

To call another task called `mynewtask` call `ape.tasks.mynewtask`:

```
#somefeature/tasks.py

@tasks.register
def taskcallingtask():
    #call mynewtask
    tasks.mynewtask(1, 2, c=3)
```

2.4.4 Running ape

2.5 Tutorial

FIXME

Suppose you have multiple computers with slightly different installations:

Office Computer

- you use `acroread` to view pdf documents
- documents are released by placing them into `~/public-html/documents/`

Home Computer

Notebook

2.6 Changelog

0.4

- better errorhandling if virualenv is not installed on debian systems.
- `bootstrape --dev` to create an ape container using the development version from github.
- place a file called `initenv` in `APE_ROOT_DIR` to customize shell environment. The file is sourced on `activape`.
- spl containers may use their own virtualenv. `ape` looks for it in `_lib/venv` inside the `CONTAINER_DIR`.
- added `aperun` script to activate a product and call a task in a single step (useful for scripts).
- `activape mysplcontainer:myproduct` activates and zaps to mysplcontainer:myproduct in a single step.

0.3

- be less verbose
- cleanup environments properly
- improved error propagation
- made the info task a little bit nicer

0.2

- first PyPI release

0.1

- initial version

CHAPTER 3

Indices and tables

- genindex
- modindex
- search