Plan Documentation

Release 0.4

Shipeng Feng

July 30, 2014

Contents

Welcome to Plan's documentation. This documentation is mainly divided into two parts. You can get started with *Installation* and then have a look at *Quickstart* in order to have a feel about how plan looks like. You can head over to *Tutorial* if you want to know how to use Plan in real life. The rest part of the user's guide is about how to define one specific job or plan object and how to run one plan object so that it talks with your crontab process. If you are interested in the internals of Plan, go to the *API* documentation.

CHAPTER 1

User's Guide

This part focuses on instructions for Cron jobs management with Plan.

1.1 Installation

You will need Python 2.6 or newer to get started, so firstly you should have an up to date Python 2.x installation. If you want to use Plan with Python 3 have a look at *Python 3 Support*.

1.1.1 virtualenv

Virtualenv is really great, maybe it is what you want to use during development. Why virtualenv? There are chances that you use Python for other projects besides Plan-managed cron jobs. It is likely that you will be working with different versions of Python, or libraries. And virtualenv is the solution if two of your projects have conflicting dependencies.

You can install virtualenv by the following commands:

```
$ sudo easy_install virtualenv
```

or better:

```
$ sudo pip install virtualenv
```

Once you have virtualenv installed, it is easy to set up one working environment. Let's create one project folder and one *venv* folder for example:

```
$ mkdir myproject
$ cd myproject
$ virtualenv venv
New python executable in venv/bin/python
Installing setuptools, pip...done.
```

Now if you want to activate the corresponding environment, just do:

```
$ . venv/bin/activate
```

1.1.2 Install Plan

You can just type the following command to get Plan:

```
$ pip install plan
```

If you are not using *virtualenv*, you will have to do one system-wide installation:

```
$ sudo pip install plan
```

1.1.3 Development Version

Get the source code from github and run it in development mode:

```
$ git clone https://github.com/fengsp/plan.git
$ cd plan
$ virtualenv venv
New python executable in venv/bin/python
Installing setuptools, pip...done.
$ . venv/bin/activate
$ python setup.py develop
...
Finished processing dependencies for Plan
```

Then you can use `git pull origin` to update to the latest version.

1.2 Quickstart

This page gives a good introduction to Plan. If you do not have Plan installed, check out *Installation* section.

1.2.1 A Minimal Example

A simple usage looks like this:

```
from plan import Plan

cron = Plan()

cron.command('ls /tmp', every='1.day', at='12:00')
cron.command('pwd', every='2.month')
cron.command('date', every='weekend')

if __name__ == '__main__':
    cron.run()
```

Just save it as *schedule.py* (or whatever you want) and run it with your Python interpreter. Make sure to not name it *plan.py* because it would conflict with Plan itself.

Now we do not run with one explicit run_type, default run_type check will be used, or you can just get your cron syntax content by access `cron_content`, you should see your cron syntax jobs in the output of terminal:

```
# Begin Plan generated jobs for: main
0 12 * * * ls /tmp
0 0 1 1,3,5,7,9,11 * pwd
0 0 * * 6,0 date
# End Plan generated jobs for: main
```

It seems everything goes fine, we can write it to crontab now:

```
if __name__ == '__main__':
    cron.run('write')
```

1.2.2 Explanation

So what did the above code do?

1. First we imported the `Plan` class. An instance of this class will be one group of cron jobs.
2. Next we create an instance of this class. We are not passing any arguments here, though, the first argument is the name of this group of cron jobs. In our case, we have the default name 'main' here. For more information have a look at the *Plan Definition* documentation.
3. We then use the `command()` to register three command jobs on this Plan object.
4. Finally we run this Plan object and check your cron syntax jobs or write it to your crontab, see *Run Types* for more details.

1.3 Tutorial

You want to manage your cron jobs with Python and Plan? Here you can learn that by this tutorial. In this tutorial I will show you how to use Plan to do cron jobs in Python. After this you should be easily creating your own cron jobs and learn the common patterns of Plan.

If you want the full source code, check out example source.

1.3.1 Introducing

We will explain our needs here, basically we want to do the following things:

1. We want to log our server running status every 4 hours.
2. We are running one Python web application and want to run a few scripts at different times.
3. As time goes on, I have a few more commands and scripts to run.

1.3.2 Basic

Let's get our basic schedule file:

```
$ mkdir schedule
$ cd schedule
$ plan-quickstart
```

Now you can see a *schedule.py* under your schedule directory, `plan-quickstart` is the command that comes with Plan for creating one example file, you can run `plan-quickstart --help` for more details. Then you can modify this file for your own needs, the file looks like this:

```
# -*- coding: utf-8 -*-

# Use this file to easily define all of your cron jobs.
#
# It's helpful to understand cron before proceeding.
# http://en.wikipedia.org/wiki/Cron
#
```

```
# Learn more: http://github.com/fengsp/plan

from plan import Plan

cron = Plan()

# register one command, script or module
# cron.command('command', every='1.day')
# cron.script('script.py', path='/web/yourproject/scripts', every='1.month')
# cron.module('calendar', every='feburary', at='day.3')

if __name__ == "__main__":
    cron.run()
```

1.3.3 One Command

Let's begin with one little command, quite clear:

```
cron.command('top', every='4.hour', output=
             dict(stdout='/tmp/top_stdout.log',
                  stderr='/tmp/top_stderr.log'))
```

Run `python schedule.py` and run it with check mode, you will see the following cron syntax content:

```
# Begin Plan generated jobs for: main
0 0,4,8,12,16,20 * * * top >> /tmp/top_stdout.log 2>> /tmp/top_stderr.log
# End Plan generated jobs for: main
```

When you call `run()`, you can choose which run-type(default to be check) you want to use, for more details on run types, check *Run Types* out.

1.3.4 Scripts

I have one script I want to run every day:

```
cron.script('script.py', every='1.day', path='/web/yourproject/scripts',
            environment={'YOURAPP_ENV': 'production'})
```

And now we have one more cron entry:

```
0 0 * * * cd /web/yourproject/scripts && YOURAPP_ENV=production python script.py
```

1.3.5 More Jobs

As time goes on, I have more cron jobs. For example, I have 10 more scripts under */web/yourproject/scripts* and 10 more commands to run. Now we have to think about how to manage these tons of jobs, first we do not want to put these jobs in one place, second we do not want to repeat the same path and environment parameter on all script jobs. Luckily, you can do that easily with Plan, basically, every `Plan` instance is a group of cron jobs:

```
$ cp schedule.py schedule_commands.py
$ cp schedule.py schedule_scripts.py
```

Now we modify schedule_commands.py:

```
from plan import Plan

cron = Plan("commands")

cron.command('top', every='4.hour', output=
             dict(stdout='/tmp/top_stdout.log',
                  stderr='/tmp/top_stderr.log'))
cron.command('yourcommand', every='sunday', at='hour.12 minute.0 minute.30')
# more commands here

if __name__ == "__main__":
    cron.run()
```

Then schedule_scripts.py:

```
from plan import Plan

cron = Plan("scripts", path='/web/yourproject/scripts',
            environment={'YOURAPP_ENV': 'production'})

cron.script('script.py', every='1.day')
cron.script('script_2.py', every='1.month', at='hour.12 minute.0')
# more scripts here

if __name__ == "__main__":
    cron.run()
```

A problem arises, how do you update your crontab content when you have two schedule files, it is simple, do not use `write` run-type, instead use `update` run-type here. `write` run-type will replace the whole crontab cronfile content with that Plan object's cron content, `update` will just add or update the corresponding block distinguished by your Plan object name (here is `"commands"` and `"scripts"`).

If you are still interested, now is your time to move on to the next part.

1.4 Job Definition

This part shows you how to define a cron job in Plan. One job takes the following parameters `task`, `every`, `at`, `path`, `environment` and `output`, you can have a look at `Job` for more details. Here is one example:

```
from plan import Job

job = Job('ruby script.rb', every='1.month', at='day.5',
          path='/web/scripts', output='null',
          environment={'RAILS_ENV': 'production'})
```

1.4.1 Every

Every is used to define how often the job runs. It takes the following values:

```
[1-60].minute
[1-24].hour
[1-31].day
[1-12].month
jan feb mar apr may jun jul aug sep oct nov dec
and all of those full month names(case insensitive)
```

```
sunday, monday, tuesday, wednesday, thursday, friday, saturday
weekday, weekend (case insensitive)
[1].year
```

There might be some cron time intervals that you cannot describe with Plan because of the limited supported syntax. No worries, every takes raw cron syntax time definition, and in this case, your at value will be ignored. For example, I can do something like this:

```
job = Job('demo', every='1,2 5,6 * * 3,4')
```

Also, every can be special predefined values, and in this case, your at value will be ignored too, they are:

```
"yearly"    # Run once a year at midnight on the morning of January 1
"monthly"   # Run once a month at midnight on the morning of the first day
            # of the month
"weekly"    # Run once a week at midnight on Sunday morning
"daily"     # Run once a day at midnight
"hourly"    # Run once an hour at the beginning of the hour
"reboot"    # Run at startup
```

1.4.2 At

At value is used to define when the job runs. It takes the following values:

```
minute.[0-59]
hour.[0-23]
hour:minute
day.[1-31]
sunday, monday, tuesday, wednesday, thursday, friday, saturday
weekday, weekend (case insensitive)
```

How about multiple at values, you can do that by using one space to seperate multiple values, for example I want to run one job every day at 12:15 and 12:45, I can define it like this:

```
job = Job('onejob', every='1.day', at='hour.12 minute.15 minute.45')
# or even better
job = Job('onejob', every='1.day', at='12:15 12:45')
```

1.4.3 Path

The path you want to change to before the task is executed, defaults to the current working directory. For job types that do not need one path, this will be ignored, for example, `CommandJob`.

1.4.4 Environment

The bash environment you want to run the task on. You should use one Python dictionary to define your environment key values pairs.

1.4.5 Output

The output redirection for the task. It takes following values:

```
"null"
any raw output string
one dictionary to define your stdout and stderr
```

For example:

```
job = Job('job', every='1.day', output='null')
job = Job('job', every='1.day', output='> /tmp/stdout.log 2> /tmp/stderr.log')
job = Job('job', every='1.day', output=
          dict(stdout='/tmp/stdout.log', stderr='/tmp/stderr.log'))
```

1.5 Plan Definition

This part shows how to initialize one plan object. One plan object takes parameters `name`, `path`, `environment`, `output` and `user`, for more details check out `Plan`. Path, environment and output are the same with *Job Definition*, they are used to set global parameters for all jobs registered on this plan object. If a job does not set one parameter, and a global parameter is set, the global one will be used.

1.5.1 Name

The name of one plan object. Should be passed as the first parameter.

1.5.2 User

If you want to run *crontab* command with a certain user, remember to set this parameter.

1.5.3 Bootstrap

Maybe you want to do some bootstrap work before you run your plan object, like you used one third party library in your Python script, you need to install it before running. You can do that as simple like this:

```
cron = Plan()
cron.bootstrap('pip install requests')
cron.bootstrap(['pip install Sphinx', 'sphinx-quickstart'])
cron.script('crawl.py', every='1.day', path='/tmp')
cron.run('check')
```

Bootstrap takes one command or a list of commands, for more details check out `bootstrap()`.

1.5.4 Patterns

One Plan object should be a group of cron jobs. The name of the plan object should be unique so that this object can be distinguished from another object. Now you can have multiple plan object around and run it with update run_type, only the corresponding content block will be renewed in the cronfile, for more details go to *Run Types* part.

1.6 Job Types

There are several Plan built-in cron job types.

1.6.1 Command Job

One command is a common executable program, check out `CommandJob`. Plan comes with one shortcut for register CommandJob `command()`.

1.6.2 Script Job

One script should be one Python pyfile, check out `ScriptJob`. Plan comes with one shortcut for register ScriptJob `script()`.

1.6.3 Module Job

One module should be one Python module, check out `ModuleJob`. Plan comes with one shortcut for register ModuleJob `module()`.

1.6.4 Raw Job

Maybe these job types are not what you want, you can define your job with raw cron syntax:

```
cron = Plan()
cron.raw('cd /tmp && ruby script.rb > /dev/null 2>&1', every='1.day')

# In this particular case, you should try Job
job = Job('ruby script.rb', every='1.day', path='/tmp', output='null')
cron.job(job)
```

1.6.5 Define Your Own Job Types

You can define your own job types if you want. Before we talk about how to do that, let's have a look on what a shortcut like `command()` do:

```
plan = Plan()
job = CommandJob(*args, **kwargs)
plan.job(job)
```

What `job()` does is registering one job on this plan object. If you want to write one own job type, just define one subclass of `Job` and override `task_template()`, let's see what *CommandJob* looks like inside Plan:

```
class CommandJob(Job):

    def task_template(self):
        return '{task} {output}'
```

The *ScriptJob* and *ModuleJob* are almost the same, with different template:

```
# ScriptJob
return 'cd {path} && {environment} %s {task} {output}' % sys.executable
# ModuleJob
return '{environment} %s -m {task} {output}' % sys.executable
```

Now I want to have one job type to run ruby script, I can define it like this:

```
class RubyJob(Job):

    def task_template(self):
        return 'cd {path} && {environment} /usr/bin/ruby {task} {output}'
```

And use it like this:

```
plan = Plan()
job = RubyJob(*args, **kwargs)
plan.job(job)
```

Mostly If *CommandJob*, *ScriptJob* and *ModuleJob* are not what you need, you can just use `Job` or even `RawJob`.

1.7 Run Types

There are several mode Plan can run on, you shoud pass the run_type parameter when you run your plan object, check out `run()`, for example:

```
cron = Plan()
cron.run('check') # could be 'check', 'write', 'update', 'clear'
```

1.7.1 Check

Check mode will just echo your cron syntax jobs out in the terminal and your crontab file will not be updated. This is used to check whether everything goes fine as you expected.

1.7.2 Write

Write mode will erase everything from your crontab cronfile and write this plan object's cron content to the cronfile, your crontab file will be fresh.

1.7.3 Update

Update mode will find the corresponding block of this plan object in the crontab cronfile and replace it with the latest content. The other content will be keeped as they were, this is distinguished by your plan object name, so make sure it's unique if you have more than one plan object.

1.7.4 Clear

Clear mode will find the corresponding block of this plan object in the crontab cronfile and erase it. The other content will not be affected.

1.8 Commands

Command line tools that come with Plan itself.

1.8.1 Plan-Quickstart

Run `plan-quickstart` in your terminal to get a quickstart example schedule file, by default, this will add the file named `schedule.py` in your current working directory, you can use `plan-quickstart --path filepath` to set your example file path. Run `plan-quickstart --help` for help.

CHAPTER 2

API Reference

This part focuses on information on a specific function, class or method.

2.1 API

This part covers some interfaces of Plan.

2.1.1 Plan Object

class `plan.`**`Plan`**(*name='main'*, *path=None*, *environment=None*, *output=None*, *user=None*)

The central object where you register jobs. One Plan instance should manage a group of jobs.

Parameters

- **name** – the unique identity for this plan object, default to be main
- **path** – the global path you want to run the task on.
- **environment** – the global crontab job bash environment.
- **output** – the global crontab job output logfile for this object.
- **user** – the user you want to run *crontab* command with.

`bootstrap`(*command_or_commands*)

Register bootstrap commands.

Parameters **command_or_commands** – One command or a list of commands.

`command`(**args*, ***kwargs*)

Register one command.

`comment_begin`

Comment begin content for this object, this will be added before the actual cron syntax jobs content. Different name is used to distinguish different Plan object, so we can locate the cronfile content corresponding to this object.

`cron_content`

Your schedule jobs converted to cron syntax.

`crons`

Return a list of registered jobs's cron syntax content.

job(*job*)
Register one job.

Parameters **job** – one `Job` instance.

module(**args*, ***kwargs*)
Register one module.

raw(**args*, ***kwargs*)
Register one raw job.

read_crontab()
Get the current working crontab cronfile content.

run(*run_type='check'*)
Use this to do any action on this Plan object.

Parameters **run_type** – The running type, one of ("check", "write", "update", "clear"), default to be "check"

run_bootstrap_commands()
Run bootstrap commands.

script(**args*, ***kwargs*)
Register one script.

update_crontab(*update_type*)
Update the current cronfile, used by run_type *update* or *clear*. This will find the block inside cronfile corresponding to this Plan object and replace it.

Parameters **update_type** – update or clear, if you choose update, the block corresponding to this plan object will be replaced with the new cron job entries, otherwise, they will be wiped.

write_crontab()
Write the crontab cronfile with this object's cron content, used by run_type *write*. This will replace the whole cronfile.

2.1.2 Job Objects

class plan.**Job**(*task*, *every*, *at=None*, *path=None*, *environment=None*, *output=None*)
The plan job base class.

Parameters

- **task** – this is what the job does.
- **every** – how often does the job run.
- **at** – when does the job run.
- **path** – the path you want to run the task on, default to be current working directory.
- **environment** – the environment you want to run the task under.
- **output** – the output redirection for the task.

cron
Job in cron syntax.

main_template
The main job template.

parse_at()
Parse at value into (at_type, moment) pairs.

parse_every()
Parse every value.

Returns every_type.

parse_month(*month*)
Parses month into month numbers. Month can only occur in every value.

Parameters **month** – this parameter can be the following values:

jan feb mar apr may jun jul aug sep oct nov dec and all of those full month names(case insenstive) or <int:n>.month

parse_time()
Parse every and at into cron time syntax:

```
# * * * * *  command to execute
#
# | | | | |
# | | | | |
# | | | | --- day of week (0 - 7) (0 to 6 are Sunday to Saturday)
# | | | ----- month (1 - 12)
# | | ------- day of month (1 - 31)
# | --------- hour (0 - 23)
# ----------- minute (0 - 59)
```

parse_week(*week*)
Parses day of week name into week numbers.

Parameters **week** – this parameter can be the following values:

sun mon tue wed thu fri sat sunday monday tuesday wednesday thursday friday saturday weekday weedend(case insenstive)

preprocess_at(*at*)
Do preprocess for at value, just modify "12:12" style moment into "hour.12 minute.12" style moment value.

Parameters **at** – The at value you want to do preprocess.

process_template(*template*)
Process template content. Drop multiple spaces in a row and strip it.

produce_frequency_time(*frequency*, *maximum*, *start=0*)
Translate frequency into comma separated times.

task_in_cron_syntax
Cron content task part.

task_template()
The task template. You should implement this in your own job type. The default template is:

```
'cd {path} && {environment} {task} {output}'
```

time_in_cron_syntax
Cron content time part.

validate_time()
Validate every and at value.

every can be:

```
[1-60].minute [1-24].hour [1-31].day
[1-12].month [1].year
jan feb mar apr may jun jul aug sep oct nov dec
sun mon tue wed thu fri sat weekday weekend
or any fullname of month names and day of week names
(case insensitive)
```

at:

```
when every is minute, can not be set
when every is hour, can be minute.[0-59]
when every is day of month, can be minute.[0-59], hour.[0-23]
when every is month, can be day.[1-31], day of week,
                     minute.[0-59], hour.[0-23]
when every is day of week, can be minute.[0-59], hour.[0-23]

at can also be multiple at values seperated by one space.
```

class plan.**CommandJob**(*task*, *every*, *at=None*, *path=None*, *environment=None*, *output=None*)

The command job.

task_template()

Template:

```
'{task} {output}'
```

class plan.**ScriptJob**(*task*, *every*, *at=None*, *path=None*, *environment=None*, *output=None*)

The script job.

task_template()

Template:

```
'cd {path} && {environment} %s {task} {output}' % sys.executable
```

class plan.**ModuleJob**(*task*, *every*, *at=None*, *path=None*, *environment=None*, *output=None*)

The module job.

task_template()

Template:

```
'{environment} %s -m {task} {output}' % sys.executable
```

class plan.**RawJob**(*task*, *every*, *at=None*, *path=None*, *environment=None*, *output=None*)

The raw job.

task_template()

Template:

```
'{task}'
```

CHAPTER 3

Additional Stuff

Changelog and license here if you are interested.

3.1 Python 3 Support

Plan and all of its dependencies support Python 3 so you can start managing your cron jobs in Python 3 Now. Though, Python 3 currently has very few users and a lot PyPI provided libraries do not support Python 3 yet. If you are a beginner, unless you are aware of what you are doing, I recommend using Python 2.x until the whole ecosystem is ready.

3.1.1 Requirements

If you want to use Plan with Python 3 you need to use Python 3.3 or higher. Older versions are *not* going to be supported.

3.2 Plan Changelog

3.2.1 Version 0.1

First public preview release.

3.2.2 Version 0.2

Released on June 20th 2014

- various bugfixes
- added support for Python 3.x

3.2.3 Version 0.3

Bugfix release, released on July 11th 2014

3.2.4 Version 0.4

Released on July 30th 2014

- replace SystemExit with PlanError when something went wrong

3.3 License

Plan is licensed under BSD License.

3.3.1 Authors

Plan is written by Shipeng Feng and various contributors:

See git contributors for more details.

Any suggestions or contributions are welcome.

3.3.2 Plan License

p