schedule Documentation

Release 0.4.0

Daniel Bader

Jan 23, 2019

Contents

Python job scheduling for humans.

An in-process scheduler for periodic jobs that uses the builder pattern for configuration. Schedule lets you run Python functions (or any other callable) periodically at pre-determined intervals using a simple, human-friendly syntax.

Inspired by Adam Wiggins' article "Rethinking Cron" and the clockwork Ruby module.

CHAPTER 1

Features

- A simple to use API for scheduling jobs.
- Very lightweight and no external dependencies.
- Excellent test coverage.
- Tested on Python 2.7, 3.5, and 3.6

CHAPTER 2

Usage

```
$ pip install schedule
```

```
import schedule
import time

def job():
    print("I'm working...")

schedule.every(10).minutes.do(job)
schedule.every().hour.do(job)
schedule.every().day.at("10:30").do(job)
schedule.every().monday.do(job)
schedule.every().wednesday.at("13:15").do(job)
schedule.every().minute.at(":17").do(job)

while True:
    schedule.run_pending()
    time.sleep(1)
```

CHAPTER 3

API Documentation

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

3.1 Developer Interface

This part of the documentation covers all the interfaces of schedule. For parts where schedule depends on external libraries, we document the most important right here and provide links to the canonical documentation.

3.1.1 Main Interface

schedule.**default_scheduler = <schedule.Scheduler object>**
Default *Scheduler* object

schedule.**jobs = []**
Default *Jobs* list

schedule.**every**(*interval=1*)
Calls *every* on the *default scheduler instance*.

schedule.**run_pending**()
Calls *run_pending* on the *default scheduler instance*.

schedule.**run_all**(*delay_seconds=0*)
Calls *run_all* on the *default scheduler instance*.

schedule.**clear**(*tag=None*)
Calls *clear* on the *default scheduler instance*.

schedule.**cancel_job**(*job*)
Calls *cancel_job* on the *default scheduler instance*.

schedule.**next_run**()
Calls *next_run* on the *default scheduler instance*.

schedule.**idle_seconds**()
Calls *idle_seconds* on the *default scheduler instance*.

3.1.2 Exceptions

exception schedule.**CancelJob**
Can be returned from a job to unschedule itself.

3.1.3 Classes

class schedule.**Scheduler**
Objects instantiated by the *Scheduler* are factories to create jobs, keep record of scheduled jobs and handle their execution.

run_pending()
Run all jobs that are scheduled to run.

Please note that it is *intended behavior that run_pending() does not run missed jobs*. For example, if you've registered a job that should run every minute and you only call run_pending() in one hour increments then your job won't be run 60 times in between but only once.

run_all(*delay_seconds=0*)
Run all jobs regardless if they are scheduled to run or not.

A delay of *delay* seconds is added between each job. This helps distribute system load generated by the jobs more evenly over time.

Parameters **delay_seconds** – A delay added between every executed job

clear(*tag=None*)
Deletes scheduled jobs marked with the given tag, or all jobs if tag is omitted.

Parameters **tag** – An identifier used to identify a subset of jobs to delete

cancel_job(*job*)
Delete a scheduled job.

Parameters **job** – The job to be unscheduled

every(*interval=1*)
Schedule a new periodic job.

Parameters **interval** – A quantity of a certain time unit

Returns An unconfigured *Job*

next_run
Datetime when the next job should run.

Returns A `datetime` object

idle_seconds

Returns Number of seconds until *next_run*.

class schedule.**Job**(*interval*, *scheduler=None*)
A periodic job as used by *Scheduler*.

Parameters

- **interval** – A quantity of a certain time unit

- **scheduler** – The *Scheduler* instance that this job will register itself with once it has been fully configured in *Job.do()*.

Every job runs at a given fixed time interval that is defined by:

- a *time unit*
- a quantity of *time units* defined by *interval*

A job is usually created and returned by *Scheduler.every()* method, which also defines its *interval*.

second

seconds

minute

minutes

hour

hours

day

days

week

weeks

monday

tuesday

wednesday

thursday

friday

saturday

sunday

tag(**tags*)

Tags the job with one or more unique indentifiers.

Tags must be hashable. Duplicate tags are discarded.

Parameters **tags** – A unique list of `Hashable` tags.

Returns The invoked job instance

at(*time_str*)

Specify a particular time that the job should be run at.

Parameters **time_str** – A string in one of the following formats: *HH:MM:SS*, *HH:MM*,':MM', *:SS*. The format must make sense given how often the job is repeating; for example, a job that repeats every minute should not be given a string in the form *HH:MM:SS*. The difference between *:MM* and *:SS* is inferred from the selected time-unit (e.g. *every().hour.at(':30')* vs. *every().minute.at(':30')*).

Returns The invoked job instance

to(*latest*)

Schedule the job to run at an irregular (randomized) interval.

The job's interval will randomly vary from the value given to *every* to *latest*. The range defined is inclusive on both ends. For example, *every(A).to(B).seconds* executes the job function every N seconds such that A <= N <= B.

Parameters **latest** – Maximum interval between randomized job runs

Returns The invoked job instance

do(*job_func*, **args*, ***kwargs*)

Specifies the job_func that should be called every time the job runs.

Any additional arguments are passed on to job_func when the job runs.

Parameters **job_func** – The function to be scheduled

Returns The invoked job instance

should_run

Returns `True` if the job should be run now.

run()

Run the job and immediately reschedule it.

Returns The return value returned by the *job_func*

CHAPTER 4

Common Questions

Please check here before creating a new issue ticket.

4.1 Frequently Asked Questions

Frequently asked questions on the usage of `schedule`.

4.1.1 How to execute jobs in parallel?

I am trying to execute 50 items every 10 seconds, but from the my logs it says it executes every item in 10 second schedule serially, is there a work around?

By default, schedule executes all jobs serially. The reasoning behind this is that it would be difficult to find a model for parallel execution that makes everyone happy.

You can work around this restriction by running each of the jobs in its own thread:

```
import threading
import time
import schedule


def job():
    print("I'm running on thread %s" % threading.current_thread())


def run_threaded(job_func):
    job_thread = threading.Thread(target=job_func)
    job_thread.start()


schedule.every(10).seconds.do(run_threaded, job)
```

(continues on next page)

(continued from previous page)

```
schedule.every(10).seconds.do(run_threaded, job)
schedule.every(10).seconds.do(run_threaded, job)
schedule.every(10).seconds.do(run_threaded, job)
schedule.every(10).seconds.do(run_threaded, job)


while 1:
    schedule.run_pending()
    time.sleep(1)
```

If you want tighter control on the number of threads use a shared jobqueue and one or more worker threads:

```
import Queue
import time
import threading
import schedule


def job():
    print("I'm working")


def worker_main():
    while 1:
        job_func = jobqueue.get()
        job_func()
        jobqueue.task_done()

jobqueue = Queue.Queue()

schedule.every(10).seconds.do(jobqueue.put, job)
schedule.every(10).seconds.do(jobqueue.put, job)
schedule.every(10).seconds.do(jobqueue.put, job)
schedule.every(10).seconds.do(jobqueue.put, job)
schedule.every(10).seconds.do(jobqueue.put, job)

worker_thread = threading.Thread(target=worker_main)
worker_thread.start()

while 1:
    schedule.run_pending()
    time.sleep(1)
```

This model also makes sense for a distributed application where the workers are separate processes that receive jobs from a distributed work queue. I like using beanstalkd with the beanstalkc Python library.

4.1.2 How to continuously run the scheduler without blocking the main thread?

Run the scheduler in a separate thread. Mrwhick wrote up a nice solution in to this problem here (look for `run_continuously()`)

4.1.3 Does schedule support timezones?

Vanilla schedule doesn't support timezones at the moment. If you need this functionality please check out @imiric's work here. He added timezone support to schedule using python-dateutil.

4.1.4 What if my task throws an exception?

Schedule doesn't catch exceptions that happen during job execution. Therefore any exceptions thrown during job execution will bubble up and interrupt schedule's run_xyz function.

If you want to guard against exceptions you can wrap your job function in a decorator like this:

```
import functools

def catch_exceptions(cancel_on_failure=False):
    def catch_exceptions_decorator(job_func):
        @functools.wraps(job_func)
        def wrapper(*args, **kwargs):
            try:
                return job_func(*args, **kwargs)
            except:
                import traceback
                print(traceback.format_exc())
                if cancel_on_failure:
                    return schedule.CancelJob
        return wrapper
    return catch_exceptions_decorator

@catch_exceptions(cancel_on_failure=True)
def bad_task():
    return 1 / 0

schedule.every(5).minutes.do(bad_task)
```

Another option would be to subclass Schedule like @mplewis did in this example.

4.1.5 How can I run a job only once?

```
def job_that_executes_once():
    # Do some work ...
    return schedule.CancelJob

schedule.every().day.at('22:30').do(job_that_executes_once)
```

4.1.6 How can I cancel several jobs at once?

You can cancel the scheduling of a group of jobs selecting them by a unique identifier.

```
def greet(name):
    print('Hello {}'.format(name))

schedule.every().day.do(greet, 'Andrea').tag('daily-tasks', 'friend')
schedule.every().hour.do(greet, 'John').tag('hourly-tasks', 'friend')
schedule.every().hour.do(greet, 'Monica').tag('hourly-tasks', 'customer')
schedule.every().day.do(greet, 'Derek').tag('daily-tasks', 'guest')

schedule.clear('daily-tasks')
```

Will prevent every job tagged as `daily-tasks` from running again.

4.1.7 I'm getting an `AttributeError: 'module' object has no attribute 'every'` when I try to use schedule. How can I fix this?

This happens if your code imports the wrong `schedule` module. Make sure you don't have a `schedule.py` file in your project that overrides the `schedule` module provided by this library.

4.1.8 How can I add generic logging to my scheduled jobs?

The easiest way to add generic logging functionality to your schedule job functions is to implement a decorator that handles logging in a reusable way:

```
import functools
import time

import schedule


# This decorator can be applied to
def with_logging(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print('LOG: Running job "%s"' % func.__name__)
        result = func(*args, **kwargs)
        print('LOG: Job "%s" completed' % func.__name__)
        return result
    return wrapper

@with_logging
def job():
    print('Hello, World.')

schedule.every(3).seconds.do(job)

while 1:
    schedule.run_pending()
    time.sleep(1)
```

4.1.9 How to run a job at random intervals?

```
def my_job():
    # This job will execute every 5 to 10 seconds.
    print('Foo')

schedule.every(5).to(10).seconds.do(my_job)
```

4.1.10 How can I pass arguments to the job function?

`do()` passes extra arguments to the job function:

```
def greet(name):
    print('Hello', name)
```

(continues on next page)

(continued from previous page)

```
schedule.every(2).seconds.do(greet, name='Alice')
schedule.every(4).seconds.do(greet, name='Bob')
```

4.1.11 How can I make sure long-running jobs are always executed on time?

Schedule does not account for the time it takes the job function to execute. To guarantee a stable execution schedule you need to move long-running jobs off the main-thread (where the scheduler runs). See "How to execute jobs in parallel?" in the FAQ for a sample implementation.

CHAPTER 5

Issues

If you encounter any problems, please file an issue along with a detailed description. Please also check the *Frequently Asked Questions* and use the search feature in the issue tracker beforehand to avoid creating duplicates. Thank you

CHAPTER 6

About Schedule

Schedule was created by Daniel Bader - @dbader_org

Distributed under the MIT license. See `LICENSE.txt` for more information.

Thanks to all the wonderful folks who have contributed to schedule over the years:

- mattss <https://github.com/mattss>
- mrhwick <https://github.com/mrhwick>
- cfrco <https://github.com/cfrco>
- matrixise <https://github.com/matrixise>
- abultman <https://github.com/abultman>
- mplewis <https://github.com/mplewis>
- WoLfulus <https://github.com/WoLfulus>
- dylwhich <https://github.com/dylwhich>
- fkromer <https://github.com/fkromer>
- alaingilbert <https://github.com/alaingilbert>
- Zerrossetto <https://github.com/Zerrossetto>
- yetingsky <https://github.com/yetingsky>
- schnepp <https://github.com/schnepp> <https://bitbucket.org/saschaschnepp>
- grampajoe <https://github.com/grampajoe>
- gilbsgilbs <https://github.com/gilbsgilbs>
- Nathan Wailes <https://github.com/NathanWailes>
- Connor Skees <https://github.com/ConnorSkees>

Python Module Index

S

Index