APScheduler Documentation

Release 2.1.2

Alex Grönholm

September 06, 2014

Contents

Contents

CHAPTER 1

Introduction

Advanced Python Scheduler (APScheduler) is a light but powerful in-process task scheduler that lets you schedule functions (or any other python callables) to be executed at times of your choosing.

This can be a far better alternative to externally run cron scripts for long-running applications (e.g. web applications), as it is platform neutral and can directly access your application's variables and functions.

The development of APScheduler was heavily influenced by the Quartz task scheduler written in Java. APScheduler provides most of the major features that Quartz does, but it also provides features not present in Quartz (such as multiple job stores).

CHAPTER 2

Features

- No (hard) external dependencies
- Thread-safe API
- Excellent test coverage (tested on CPython 2.5 - 2.7, 3.2 - 3.3, Jython 2.5.3, PyPy 2.2)
- Configurable scheduling mechanisms (triggers):
  - Cron-like scheduling
  - Delayed scheduling of single run jobs (like the UNIX "at" command)
  - Interval-based (run a job at specified time intervals)
- Multiple, simultaneously active job stores:
  - RAM
  - File-based simple database (`shelve`)
  - SQLAlchemy (any supported RDBMS works)
  - MongoDB
  - Redis

CHAPTER 3

Usage

3.1 Installing APScheduler

The preferred installation method is by using pip:

```
$ pip install apscheduler
```

or easy_install:

```
$ easy_install apscheduler
```

If that doesn't work, you can manually download the APScheduler distribution from PyPI, extract and then install it:

```
$ python setup.py install
```

3.2 Starting the scheduler

To start the scheduler with default settings:

```
from apscheduler.scheduler import Scheduler

sched = Scheduler()
sched.start()
```

The constructor takes as its first, optional parameter a dictionary of "global" options to facilitate configuration from .ini files. All APScheduler options given in the global configuration must begin with "apscheduler." to avoid name clashes with other software. The constructor also takes options as keyword arguments (without the prefix).

You can also configure the scheduler after its instantiation, if necessary. This is handy if you use the decorators for scheduling and must have a Scheduler instance available from the very beginning:

```
from apscheduler.scheduler import Scheduler

sched = Scheduler()

@sched.interval_schedule(hours=3)
def some_job():
    print "Decorated job"

sched.configure(options_from_ini_file)
sched.start()
```

3.3 Scheduling jobs

The simplest way to schedule jobs using the built-in triggers is to use one of the shortcut methods provided by the scheduler:

3.3.1 Simple date-based scheduling

This is the simplest possible method of scheduling a job. It schedules a job to be executed once at the specified time. This is the in-process equivalent to the UNIX "at" command.

```
from datetime import date
from apscheduler.scheduler import Scheduler

# Start the scheduler
sched = Scheduler()
sched.start()

# Define the function that is to be executed
def my_job(text):
    print text

# The job will be executed on November 6th, 2009
exec_date = date(2009, 11, 6)

# Store the job in a variable in case we want to cancel it
job = sched.add_date_job(my_job, exec_date, ['text'])
```

We could be more specific with the scheduling too:

```
from datetime import datetime

# The job will be executed on November 6th, 2009 at 16:30:05
job = sched.add_date_job(my_job, datetime(2009, 11, 6, 16, 30, 5), ['text'])
```

You can even specify a date as text, with or without the time part:

```
job = sched.add_date_job(my_job, '2009-11-06 16:30:05', ['text'])

# Even down to the microsecond level, if you really want to!
job = sched.add_date_job(my_job, '2009-11-06 16:30:05.720400', ['text'])
```

3.3.2 Interval-based scheduling

This method schedules jobs to be run on selected intervals. The execution of the job starts after the given delay, or on `start_date` if specified. After that, the job will be executed again after the specified delay. The `start_date` parameter can be given as a date/datetime object or text. See the *Date-based scheduling section* for more examples on that.

```
from datetime import datetime

from apscheduler.scheduler import Scheduler

# Start the scheduler
sched = Scheduler()
sched.start()
```

```
def job_function():
    print "Hello World"

# Schedule job_function to be called every two hours
sched.add_interval_job(job_function, hours=2)

# The same as before, but start after a certain time point
sched.add_interval_job(job_function, hours=2, start_date='2010-10-10 09:30')
```

Decorator syntax

As a convenience, there is an alternative syntax for using interval-based schedules. The `interval_schedule()` decorator can be attached to any function, and has the same syntax as `add_interval_job()`, except for the `func` parameter, obviously.

```
from apscheduler.scheduler import Scheduler

# Start the scheduler
sched = Scheduler()
sched.start()

# Schedule job_function to be called every two hours
@sched.interval_schedule(hours=2)
def job_function():
    print "Hello World"
```

If you need to unschedule the decorated functions, you can do it this way:

```
scheduler.unschedule_job(job_function.job)
```

3.3.3 Cron-style scheduling

This is the most powerful scheduling method available in APScheduler. You can specify a variety of different expressions on each field, and when determining the next execution time, it finds the earliest possible time that satisfies the conditions in every field. This behavior resembles the "Cron" utility found in most UNIX-like operating systems.

You can also specify the starting date for the cron-style schedule through the `start_date` parameter, which can be given as a date/datetime object or text. See the *Date-based scheduling section* for examples on that.

Unlike with crontab expressions, you can omit fields that you don't need. Fields greater than the least significant explicitly defined field default to `*` while lesser fields default to their minimum values except for `week` and `day_of_week` which default to `*`. For example, if you specify only `day=1, minute=20`, then the job will execute on the first day of every month on every year at 20 minutes of every hour. The code examples below should further illustrate this behavior.

Note: The behavior for omitted fields was changed in APScheduler 2.0. Omitted fields previously always defaulted to `*`.

Available fields

Field	Description
year	4-digit year number
month	month number (1-12)
day	day of the month (1-31)
week	ISO week number (1-53)
day_of_week	number or name of weekday (0-6 or mon,tue,wed,thu,fri,sat,sun)
hour	hour (0-23)
minute	minute (0-59)
second	second (0-59)

Note: The first weekday is always **monday**.

Expression types

The following table lists all the available expressions applicable in cron-style schedules.

Expression	Field	Description
*	any	Fire on every value
*/a	any	Fire every a values, starting from the minimum
a-b	any	Fire on any value within the a-b range (a must be smaller than b)
a-b/c	any	Fire every c values within the a-b range
xth y	day	Fire on the x -th occurrence of weekday y within the month
last x	day	Fire on the last occurrence of weekday x within the month
last	day	Fire on the last day within the month
x,y,z	any	Fire on any matching expression; can combine any number of any of the above expressions

Example 1

```
from apscheduler.scheduler import Scheduler

# Start the scheduler
sched = Scheduler()
sched.start()

def job_function():
    print "Hello World"

# Schedules job_function to be run on the third Friday
# of June, July, August, November and December at 00:00, 01:00, 02:00 and 03:00
sched.add_cron_job(job_function, month='6-8,11-12', day='3rd fri', hour='0-3')
```

Example 2

```
# Initialization similar as above, the backup function defined elsewhere

# Schedule a backup to run once from Monday to Friday at 5:30 (am)
sched.add_cron_job(backup, day_of_week='mon-fri', hour=5, minute=30)
```

Decorator syntax

As a convenience, there is an alternative syntax for using cron-style schedules. The `cron_schedule()` decorator can be attached to any function, and has the same syntax as `add_cron_job()`, except for the `func` parameter, obviously.

```
@sched.cron_schedule(day='last sun')
def some_decorated_task():
    print "I am printed at 00:00:00 on the last Sunday of every month!"
```

If you need to unschedule the decorated functions, you can do it this way:

```
scheduler.unschedule_job(job_function.job)
```

These shortcuts cover the vast majority of use cases. However, if you need to use a custom trigger, you need to use the `add_job()` method.

When a scheduled job is triggered, it is handed over to the thread pool for execution.

You can request a job to be added to a specific job store by giving the target job store's alias in the `jobstore` option to `add_job()` or any of the above shortcut methods.

You can schedule jobs on the scheduler **at any time**. If the scheduler is not running when the job is added, the job will be scheduled *tentatively* and its first run time will only be computed when the scheduler starts. Jobs will not run retroactively in such cases.

Warning: Scheduling new jobs from existing jobs is not currently reliable. This will likely be fixed in the next major release.

3.4 Job options

The following options can be given as keyword arguments to `add_job()` or one of the shortcut methods, including the decorators.

Option	Definition
name	Name of the job (informative, does not have to be unique)
misfire_grace_time	Time in seconds that the job is allowed to miss the the designated run time before being considered to have misfired (see *Missed job executions and coalescing*) (overrides the global scheduler setting)
coalesce	Run once instead of many times if the scheduler determines that the job should be run more than once in succession (see *Missed job executions and coalescing*) (overrides the global scheduler setting)
max_runs	Maximum number of times this job is allowed to be triggered before being removed
max_instances	Maximum number of concurrently running instances allowed for this job (see *_max_instances*)

3.5 Shutting down the scheduler

To shut down the scheduler:

```
sched.shutdown()
```

By default, the scheduler shuts down its thread pool and waits until all currently executing jobs are finished. For a faster exit you can do:

```
sched.shutdown(wait=False)
```

This will still shut down the thread pool but does not wait for any running tasks to complete. Also, if you gave the scheduler a thread pool that you want to manage elsewhere, you probably want to skip the thread pool shutdown altogether:

```
sched.shutdown(shutdown_threadpool=False)
```

This implies `wait=False`, since there is no way to wait for the scheduler's tasks to finish without shutting down the thread pool.

A neat trick to automatically shut down the scheduler is to use an `atexit` hook for that:

```
import atexit

sched = Scheduler(daemon=True)
atexit.register(lambda: sched.shutdown(wait=False))
# Proceed with starting the actual application
```

3.6 Scheduler configuration options

Directive	Default	Definition
misfire_grace_time	1	Maximum time in seconds for the job execution to be allowed to delay before it is considered a misfire (see *Missed job executions and coalescing*)
coalesce	False	Roll several pending executions of jobs into one (see *Missed job executions and coalescing*)
standalone	False	If set to `True`, `start()` will run the main loop in the calling thread until no more jobs are scheduled. See *Operating modes: embedded vs standalone* for more information.
daemonic	True	Controls whether the scheduler thread is daemonic or not. This option has no effect when `standalone` is `True`. If set to `False`, then the scheduler must be shut down explicitly when the program is about to finish, or it will prevent the program from terminating. If set to `True`, the scheduler will automatically terminate with the application, but may cause an exception to be raised on exit. Jobs are always executed in non-daemonic threads.
threadpool	(built-in)	Instance of a **PEP 3148** compliant thread pool or a dot-notation (`x.y.z:varname`) reference to one
threadpool.core_threads	0	Maximum number of persistent threads in the pool
threadpool.max_threads	20	Maximum number of total threads in the pool
threadpool.keepalive	1	Seconds to keep non-core worker threads waiting for new tasks
jobstore.X.class		Class of the jobstore named X (specified as module.name:classname)
jobstore.X.Y		Constructor option Y of jobstore X

3.7 Operating modes: embedded vs standalone

The scheduler has two operating modes: standalone and embedded. In embedded mode, it will spawn its own thread when `start()` is called. In standalone mode, it will run directly in the calling thread and will block until there are no more pending jobs.

The embedded mode is suitable for running alongside some application that requires scheduling capabilities. The standalone mode, on the other hand, can be used as a handy cross-platform cron replacement for executing Python code. A typical usage of the standalone mode is to have a script that only adds the jobs to the scheduler and then calls `start()` on the scheduler.

All of the examples in the examples directory demonstrate usage of the standalone mode, with the exception of `threaded.py` which demonstrates the embedded mode (where the "application" just prints a line every 2 seconds).

3.8 Job stores

APScheduler keeps all the scheduled jobs in *job stores*. Job stores are configurable adapters to some back-end that may or may not support persisting job configurations on disk, database or something else. Job stores are added to the scheduler and identified by their aliases. The alias `default` is special in that if the user does not explicitly specify a job store alias when scheduling a job, it goes to the `default` job store. If there is no job store in the scheduler by that name when the scheduler is started, a new job store of type `RAMJobStore` is created to serve as the default.

The other built-in job stores are:

- `ShelveJobStore`
- `SQLAlchemyJobStore`
- `MongoDBJobStore`
- `RedisJobStore`

Job stores can be added either through configuration options or the `add_jobstore()` method. The following are therefore equal:

```
config = {'apscheduler.jobstores.file.class': 'apscheduler.jobstores.shelve_store:ShelveJobStore',
          'apscheduler.jobstores.file.path': '/tmp/dbfile'}
sched = Scheduler(config)
```

and:

```
from apscheduler.jobstores.shelve_store import ShelveJobStore

    sched = Scheduler()
    sched.add_jobstore(ShelveJobStore('/tmp/dbfile'), 'file')
```

The example configuration above results in the scheduler having two job stores – one `RAMJobStore` and one `ShelveJobStore`.

3.9 Job persistency

The built-in job stores (other than `RAMJobStore`) store jobs in a durable manner. This means that when you schedule jobs in them, shut down the scheduler, restart it and readd the job store in question, it will load the previously scheduled jobs automatically.

Persistent job stores store a reference to the target callable in text form and serialize the arguments using pickle. This unfortunately adds some restrictions:

- You cannot schedule static methods, inner functions or lambdas.
- You cannot update the objects given as arguments to the callable.

Technically you *can* update the state of the argument objects, but those changes are never persisted back to the job store.

Note: None of these restrictions apply to `RAMJobStore`.

3.10 Limiting the number of concurrently executing instances of a job

By default, no two instances of the same job will be run concurrently. This means that if the job is about to be run but the previous run hasn't finished yet, then the latest run is considered a misfire. It is possible to set the maximum number of instances for a particular job that the scheduler will let run concurrently, by using the `max_instances` keyword argument when adding the job.

3.11 Missed job executions and coalescing

Sometimes the scheduler may be unable to execute a scheduled job at the time it was scheduled to run. The most common case is when a job is scheduled in a persistent job store and the scheduler is shut down and restarted after the job was supposed to execute. When this happens, the job is considered to have "misfired". The scheduler will then check each missed execution time against the job's `misfire_grace_time` option (which can be set on per-job basis or globally in the scheduler) to see if the execution should still be triggered. This can lead into the job being executed several times in succession.

If this behavior is undesirable for your particular use case, it is possible to use *coalescing* to roll all these missed executions into one. In other words, if coalescing is enabled for the job and the scheduler sees one or more queued executions for the job, it will only trigger it once. The "bypassed" runs of the job are not considered misfires nor do they count towards any maximum run count of the job.

3.12 Scheduler events

It is possible to attach event listeners to the scheduler. Scheduler events are fired on certain occasions, and may carry additional information in them concerning the details of that particular event. It is possible to listen to only particular types of events by giving the appropriate `mask` argument to `add_listener()`, OR'ing the different constants together. The listener callable is called with one argument, the event object. The type of the event object is tied to the event code as shown below:

Constant	Event class	Triggered when...
EVENT_SCHEDULER_START	SchedulerEvent	The scheduler is started
EVENT_SCHEDULER_SHUTDOWN	SchedulerEvent	The scheduler is shut down
EVENT_JOBSTORE_ADDED	JobStoreEvent	A job store is added to the scheduler
EVENT_JOBSTORE_REMOVED	JobStoreEvent	A job store is removed from the scheduler
EVENT_JOBSTORE_JOB_ADDED	JobStoreEvent	A job is added to a job store
EVENT_JOBSTORE_JOB_REMOVED	JobStoreEvent	A job is removed from a job store
EVENT_JOB_EXECUTED	JobEvent	A job is executed successfully
EVENT_JOB_ERROR	JobEvent	A job raised an exception during execution
EVENT_JOB_MISSED	JobEvent	A job's execution time is missed

See the documentation for the `events` module for specifics on the available event attributes.

Example:

```
def my_listener(event):
    if event.exception:
        print 'The job crashed :('
    else:
        print 'The job worked :)'

scheduler.add_listener(my_listener, EVENT_JOB_EXECUTED | EVENT_JOB_ERROR)
```

3.13 Getting a list of scheduled jobs

If you want to see which jobs are currently added in the scheduler, you can simply do:

```
sched.print_jobs()
```

This will print a human-readable listing of scheduled jobs, their triggering mechanisms and the next time they will fire. If you supply a file-like object as an argument to this method, it will output the results in that file.

To get a machine processable list of the scheduled jobs, you can use the `get_jobs()` scheduler method. It will return a list of `Job` instances.

CHAPTER 4

Extending APScheduler

It is possible to extend APScheduler to support alternative job stores and triggers. See the `Extending APScheduler` document for details.

CHAPTER 5

FAQ

Q: Why do my processes hang instead of exiting when they are finished?

A: A scheduled job may still be executing. APScheduler's thread pool is wired to wait for the job threads to exit before allowing the interpreter to exit to avoid unpredictable behavior caused by the shutdown procedures of the Python interpreter. A more thorough explanation can be found here.

CHAPTER 6

Reporting bugs

A bug tracker is provided by bitbucket.org.

CHAPTER 7

Getting help

If you have problems or other questions, you can either:

- Ask on the APScheduler Google group, or
- Ask on the `#apscheduler` channel on Freenode IRC

CHAPTER 8

Version history

To find out how to migrate your application from a previous version of APScheduler, see the `migration section`.

8.1 2.1.2

- Fixed shelve store losing data after on an unclean shutdown
- Worked around an obscure Linux kernel bug that causes IOErrors in the main loop

8.2 2.1.1

- Fixed shutdown() in standalone mode
- Fixed the default value of the "db" parameter in the redis job store
- Switched to PyPy 2.0 for PyPy compatibility testing

8.3 2.1.0

- Added Redis job store
- Added a "standalone" mode that runs the scheduler in the calling thread
- Fixed disk synchronization in ShelveJobStore
- Switched to PyPy 1.9 for PyPy compatibility testing
- Dropped Python 2.4 support
- Fixed SQLAlchemy 0.8 compatibility in SQLAlchemyJobStore
- Various documentation improvements

8.4 2.0.3

- The scheduler now closes the job store that is being removed, and all job stores on shutdown() by default
- Added the `last` expression in the day field of CronTrigger (thanks rcaselli)

- Raise a TypeError when fields with invalid names are passed to CronTrigger (thanks Christy O'Reilly)
- Fixed the persistent.py example by shutting down the scheduler on Ctrl+C
- Added PyPy 1.8 and CPython 3.3 to the test suite
- Dropped PyPy 1.4 - 1.5 and CPython 3.1 from the test suite
- Updated setup.cfg for compatibility with distutils2/packaging
- Examples, documentation sources and unit tests are now packaged in the source distribution

8.5 2.0.2

- Removed the unique constraint from the "name" column in the SQLAlchemy job store
- Fixed output from Scheduler.print_jobs() which did not previously output a line ending at the end

8.6 2.0.1

- Fixed cron style jobs getting wrong default values

8.7 2.0.0

- Added configurable job stores with several persistent back-ends (shelve, SQLAlchemy and MongoDB)
- Added the possibility to listen for job events (execution, error, misfire, finish) on a scheduler
- Added an optional start time for cron-style jobs
- Added optional job execution coalescing for situations where several executions of the job are due
- Added an option to limit the maximum number of concurrenctly executing instances of the job
- Allowed configuration of misfire grace times on a per-job basis
- Allowed jobs to be explicitly named
- All triggers now accept dates in string form (YYYY-mm-dd HH:MM:SS)
- Jobs are now run in a thread pool; you can either supply your own PEP 3148 compliant thread pool or let APScheduler create its own
- Maximum run count can be configured for all jobs, not just those using interval-based scheduling
- Fixed a v1.x design flaw that caused jobs to be executed twice when the scheduler thread was woken up while still within the allowable range of their previous execution time (issues #5, #7)
- Changed defaults for cron-style jobs to be more intuitive – it will now default to all minimum values for fields lower than the least significant explicitly defined field

8.8 1.3.1

- Fixed time difference calculation to take into account shifts to and from daylight saving time

8.9 1.3.0

- Added __repr__() implementations to expressions, fields, triggers, and jobs to help with debugging
- Added the dump_jobs method on Scheduler, which gives a helpful listing of all jobs scheduled on it
- Fixed positional weekday (3th fri etc.) expressions not working except in some edge cases (fixes #2)
- Removed autogenerated API documentation for modules which are not part of the public API, as it might confuse some users

Note: Positional weekdays are now used with the **day** field, not **weekday**.

8.10 1.2.1

- Fixed regression: add_cron_job() in Scheduler was creating a CronTrigger with the wrong parameters (fixes #1, #3)
- Fixed: if the scheduler is restarted, clear the "stopped" flag to allow jobs to be scheduled again

8.11 1.2.0

- Added the `week` option for cron schedules
- Added the `daemonic` configuration option
- Fixed a bug in cron expression lists that could cause valid firing times to be missed
- Fixed unscheduling bound methods via unschedule_func()
- Changed CronTrigger constructor argument names to match those in Scheduler

8.12 1.01

- Fixed a corner case where the combination of hour and day_of_week parameters would cause incorrect timing for a cron trigger