APScheduler Documentation

Release 3.3.1.post7

Alex Grönholm

May 17, 2017

Contents

1 Table of Contents 3

2 Indices and tables 21

Advanced Python Scheduler (APScheduler) is a Python library that lets you schedule your Python code to be executed later, either just once or periodically. You can add new jobs or remove old ones on the fly as you please. If you store your jobs in a database, they will also survive scheduler restarts and maintain their state. When the scheduler is restarted, it will then run all the jobs it should have run while it was offline[1].

Among other things, APScheduler can be used as a cross-platform, application specific replacement to platform specific schedulers, such as the cron daemon or the Windows task scheduler. Please note, however, that APScheduler is **not** a daemon or service itself, nor does it come with any command line tools. It is primarily meant to be run inside existing applications. That said, APScheduler does provide some building blocks for you to build a scheduler service or to run a dedicated scheduler process.

APScheduler has three built-in scheduling systems you can use:

- Cron-style scheduling (with optional start/end times)
- Interval-based execution (runs jobs on even intervals, with optional start/end times)
- One-off delayed execution (runs jobs once, on a set date/time)

You can mix and match scheduling systems and the backends where the jobs are stored any way you like. Supported backends for storing jobs include:

- Memory
- SQLAlchemy (any RDBMS supported by SQLAlchemy works)
- MongoDB
- Redis
- RethinkDB
- ZooKeeper

APScheduler also integrates with several common Python frameworks, like:

- asyncio (**PEP 3156**)
- gevent
- Tornado
- Twisted
- Qt (using either PyQt or PySide)

[1] The cutoff period for this is also configurable.

CHAPTER 1

Table of Contents

User guide

Installing APScheduler

The preferred installation method is by using pip:

```
$ pip install apscheduler
```

If you don't have pip installed, you can easily install it by downloading and running get-pip.py.

If, for some reason, pip won't work, you can manually download the APScheduler distribution from PyPI, extract and then install it:

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

Code examples

The source distribution contains the `examples` directory where you can find many working examples for using APScheduler in different ways. The examples can also be browsed online.

Basic concepts

APScheduler has four kinds of components:

- triggers
- job stores
- executors
- schedulers

Triggers contain the scheduling logic. Each job has its own trigger which determines when the job should be run next. Beyond their initial configuration, triggers are completely stateless.

Job stores house the scheduled jobs. The default job store simply keeps the jobs in memory, but others store them in various kinds of databases. A job's data is serialized when it is saved to a persistent job store, and deserialized when it's loaded back from it. Job stores (other than the default one) don't keep the job data in memory, but act as middlemen for saving, loading, updating and searching jobs in the backend. Job stores must never be shared between schedulers.

Executors are what handle the running of the jobs. They do this typically by submitting the designated callable in a job to a thread or process pool. When the job is done, the executor notifies the scheduler which then emits an appropriate event.

Schedulers are what bind the rest together. You typically have only one scheduler running in your application. The application developer doesn't normally deal with the job stores, executors or triggers directly. Instead, the scheduler provides the proper interface to handle all those. Configuring the job stores and executors is done through the scheduler, as is adding, modifying and removing jobs.

Choosing the right scheduler, job store(s), executor(s) and trigger(s)

Your choice of scheduler depends mostly on your programming environment and what you'll be using APScheduler for. Here's a quick guide for choosing a scheduler:

- `BlockingScheduler`: use when the scheduler is the only thing running in your process
- `BackgroundScheduler`: use when you're not using any of the frameworks below, and want the scheduler to run in the background inside your application
- `AsyncIOScheduler`: use if your application uses the asyncio module
- `GeventScheduler`: use if your application uses gevent
- `TornadoScheduler`: use if you're building a Tornado application
- `TwistedScheduler`: use if you're building a Twisted application
- `QtScheduler`: use if you're building a Qt application

Simple enough, yes?

To pick the appropriate job store, you need to determine whether you need job persistence or not. If you always recreate your jobs at the start of your application, then you can probably go with the default (`MemoryJobStore`). But if you need your jobs to persist over scheduler restarts or application crashes, then your choice usually boils down to what tools are used in your programming environment. If, however, you are in the position to choose freely, then `SQLAlchemyJobStore` on a PostgreSQL backend is the recommended choice due to its strong data integrity protection.

Likewise, the choice of executors is usually made for you if you use one of the frameworks above. Otherwise, the default `ThreadPoolExecutor` should be good enough for most purposes. If your workload involves CPU intensive operations, you should consider using `ProcessPoolExecutor` instead to make use of multiple CPU cores. You could even use both at once, adding the process pool executor as a secondary executor.

When you schedule a job, you need to choose a _trigger_ for it. The trigger determines the logic by which the dates/times are calculated when the job will be run. APScheduler comes with three built-in trigger types:

- `date`: use when you want to run the job just once at a certain point of time
- `interval`: use when you want to run the job at fixed intervals of time
- `cron`: use when you want to run the job periodically at certain time(s) of day

You can find the plugin names of each job store, executor and trigger type on their respective API documentation pages.

Configuring the scheduler

APScheduler provides many different ways to configure the scheduler. You can use a configuration dictionary or you can pass in the options as keyword arguments. You can also instantiate the scheduler first, add jobs and configure the scheduler afterwards. This way you get maximum flexibility for any environment.

The full list of scheduler level configuration options can be found on the API reference of the `BaseScheduler` class. Scheduler subclasses may also have additional options which are documented on their respective API references. Configuration options for individual job stores and executors can likewise be found on their API reference pages.

Let's say you want to run BackgroundScheduler in your application with the default job store and the default executor:

```
from apscheduler.schedulers.background import BackgroundScheduler


scheduler = BackgroundScheduler()

# Initialize the rest of the application here, or before the scheduler initialization
```

This will get you a BackgroundScheduler with a MemoryJobStore named "default" and a ThreadPoolExecutor named "default" with a default maximum thread count of 10.

Now, suppose you want more. You want to have *two* job stores using *two* executors and you also want to tweak the default values for new jobs and set a different timezone. The following three examples are completely equivalent, and will get you:

- a MongoDBJobStore named "mongo"
- an SQLAlchemyJobStore named "default" (using SQLite)
- a ThreadPoolExecutor named "default", with a worker count of 20
- a ProcessPoolExecutor named "processpool", with a worker count of 5
- UTC as the scheduler's timezone
- coalescing turned off for new jobs by default
- a default maximum instance limit of 3 for new jobs

Method 1:

```
from pytz import utc

from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.jobstores.mongodb import MongoDBJobStore
from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore
from apscheduler.executors.pool import ThreadPoolExecutor, ProcessPoolExecutor


jobstores = {
    'mongo': MongoDBJobStore(),
    'default': SQLAlchemyJobStore(url='sqlite:///jobs.sqlite')
}
executors = {
    'default': ThreadPoolExecutor(20),
    'processpool': ProcessPoolExecutor(5)
}
```

```
job_defaults = {
    'coalesce': False,
    'max_instances': 3
}
scheduler = BackgroundScheduler(jobstores=jobstores, executors=executors, job_
↪defaults=job_defaults, timezone=utc)
```

Method 2:

```
from apscheduler.schedulers.background import BackgroundScheduler


# The "apscheduler." prefix is hard coded
scheduler = BackgroundScheduler({
    'apscheduler.jobstores.mongo': {
         'type': 'mongodb'
    },
    'apscheduler.jobstores.default': {
        'type': 'sqlalchemy',
        'url': 'sqlite:///jobs.sqlite'
    },
    'apscheduler.executors.default': {
        'class': 'apscheduler.executors.pool:ThreadPoolExecutor',
        'max_workers': '20'
    },
    'apscheduler.executors.processpool': {
        'type': 'processpool',
        'max_workers': '5'
    },
    'apscheduler.job_defaults.coalesce': 'false',
    'apscheduler.job_defaults.max_instances': '3',
    'apscheduler.timezone': 'UTC',
})
```

Method 3:

```
from pytz import utc

from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore
from apscheduler.executors.pool import ProcessPoolExecutor


jobstores = {
    'mongo': {'type': 'mongodb'},
    'default': SQLAlchemyJobStore(url='sqlite:///jobs.sqlite')
}
executors = {
    'default': {'type': 'threadpool', 'max_workers': 20},
    'processpool': ProcessPoolExecutor(max_workers=5)
}
job_defaults = {
    'coalesce': False,
    'max_instances': 3
}
scheduler = BackgroundScheduler()

# .. do something else here, maybe add jobs etc.
```

```
scheduler.configure(jobstores=jobstores, executors=executors, job_defaults=job_
↪defaults, timezone=utc)
```

Starting the scheduler

Starting the scheduler is done by simply calling `start()` on the scheduler. For schedulers other than *~apscheduler.schedulers.blocking.BlockingScheduler*, this call will return immediately and you can continue the initialization process of your application, possibly adding jobs to the scheduler.

For BlockingScheduler, you will only want to call `start()` after you're done with any initialization steps.

Note: After the scheduler has been started, you can no longer alter its settings.

Adding jobs

There are two ways to add jobs to a scheduler:

1. by calling `add_job()`
2. by decorating a function with `scheduled_job()`

The first way is the most common way to do it. The second way is mostly a convenience to declare jobs that don't change during the application's run time. The `add_job()` method returns a `apscheduler.job.Job` instance that you can use to modify or remove the job later.

You can schedule jobs on the scheduler **at any time**. If the scheduler is not yet 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.

It is important to note that if you use an executor or job store that serializes the job, it will add a couple requirements on your job:

1. The target callable must be globally accessible
2. Any arguments to the callable must be serializable

Of the builtin job stores, only MemoryJobStore doesn't serialize jobs. Of the builtin executors, only ProcessPoolExecutor will serialize jobs.

Important: If you schedule jobs in a persistent job store during your application's initialization, you **MUST** define an explicit ID for the job and use `replace_existing=True` or you will get a new copy of the job every time your application restarts!

Tip: To run a job immediately, omit `trigger` argument when adding the job.

Removing jobs

When you remove a job from the scheduler, it is removed from its associated job store and will not be executed anymore. There are two ways to make this happen:

1. by calling `remove_job()` with the job's ID and job store alias

2. by calling `remove()` on the Job instance you got from `add_job()`

The latter method is probably more convenient, but it requires that you store somewhere the `Job` instance you received when adding the job. For jobs scheduled via the `scheduled_job()`, the first way is the only way.

If the job's schedule ends (i.e. its trigger doesn't produce any further run times), it is automatically removed.

Example:

```
job = scheduler.add_job(myfunc, 'interval', minutes=2)
job.remove()
```

Same, using an explicit job ID:

```
scheduler.add_job(myfunc, 'interval', minutes=2, id='my_job_id')
scheduler.remove_job('my_job_id')
```

Pausing and resuming jobs

You can easily pause and resume jobs through either the `Job` instance or the scheduler itself. When a job is paused, its next run time is cleared and no further run times will be calculated for it until the job is resumed. To pause a job, use either method:

- `apscheduler.job.Job.pause()`
- `apscheduler.schedulers.base.BaseScheduler.pause_job()`

To resume:

- `apscheduler.job.Job.resume()`
- `apscheduler.schedulers.base.BaseScheduler.resume_job()`

Getting a list of scheduled jobs

To get a machine processable list of the scheduled jobs, you can use the `get_jobs()` method. It will return a list of `Job` instances. If you're only interested in the jobs contained in a particular job store, then give a job store alias as the second argument.

As a convenience, you can use the `print_jobs()` method which will print out a formatted list of jobs, their triggers and next run times.

Modifying jobs

You can modify any job attributes by calling either `apscheduler.job.Job.modify()` or `modify_job()`. You can modify any Job attributes except for `id`.

Example:

```
job.modify(max_instances=6, name='Alternate name')
```

If you want to reschedule the job – that is, change its trigger, you can use either `apscheduler.job.Job.reschedule()` or `reschedule_job()`. These methods construct a new trigger for the job and recalculate its next run time based on the new trigger.

Example:

```
scheduler.reschedule_job('my_job_id', trigger='cron', minute='*/5')
```

Shutting down the scheduler

To shut down the scheduler:

```
scheduler.shutdown()
```

By default, the scheduler shuts down its job stores and executors and waits until all currently executing jobs are finished. If you don't want to wait, you can do:

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

This will still shut down the job stores and executors but does not wait for any running tasks to complete.

Pausing/resuming job processing

It is possible to pause the processing of scheduled jobs:

```
scheduler.pause()
```

This will cause the scheduler to not wake up until processing is resumed:

```
scheduler.resume()
```

It is also possible to start the scheduler in paused state, that is, without the first wakeup call:

```
scheduler.start(paused=True)
```

This is useful when you need to prune unwanted jobs before they have a chance to run.

Limiting the number of concurrently executing instances of a job

By default, only one instance of each job is allowed to be run at the same time. 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.

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. No misfire events will be sent for the "bypassed" runs.

Note: If the execution of a job is delayed due to no threads or processes being available in the pool, the executor may skip it due to it being run too late (compared to its originally designated run time). If this is likely to happen in your application, you may want to either increase the number of threads/processes in the executor, or adjust the `misfire_grace_time` setting to a higher value.

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.

See the documentation for the `events` module for specifics on the available events and their 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)
```

Reporting bugs

A bug tracker is provided by Github.

Getting help

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

- Ask on the `#apscheduler` channel on Freenode IRC
- Ask on the APScheduler Google group, or
- Ask on StackOverflow and tag your question with the `apscheduler` tag

Version history

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

3.3.1

- Fixed Python 2.7 compatibility in `TornadoExecutor`

3.3.0

- The asyncio and Tornado schedulers can now run jobs targeting coroutine functions (requires Python 3.5; only native coroutines (`async def`) are supported)
- The Tornado scheduler now uses TornadoExecutor as its default executor (see above as for why)
- Added ZooKeeper job store (thanks to Jose Ignacio Villar for the patch)
- Fixed job store failure (`get_due_jobs()`) causing the scheduler main loop to exit (it now waits a configurable number of seconds before retrying)
- Fixed `@scheduled_job` not working when serialization is required (persistent job stores and `ProcessPoolScheduler`)
- Improved import logic in `ref_to_obj()` to avoid errors in cases where traversing the path with `getattr()` would not work (thanks to Jarek Glowacki for the patch)
- Fixed CronTrigger's weekday position expressions failing on Python 3
- Fixed CronTrigger's range expressions sometimes allowing values outside the given range

3.2.0

- Added the ability to pause and unpause the scheduler
- Fixed pickling problems with persistent jobs when upgrading from 3.0.x
- Fixed AttributeError when importing apscheduler with setuptools < 11.0
- Fixed some events missing from `apscheduler.events.__all__` and `apscheduler.events.EVENTS_ALL`
- Fixed wrong run time being set for date trigger when the timezone isn't the same as the local one
- Fixed builtin `id()` erroneously used in MongoDBJobStore's `JobLookupError()`
- **Fixed endless loop with CronTrigger that may occur when the computer's clock resolution is too** low (thanks to Jinping Bai for the patch)

3.1.0

- Added RethinkDB job store (contributed by Allen Sanabria)
- **Added method chaining to the `modify_job()`, `reschedule_job()`, `pause_job()` and** `resume_job()` methods in `BaseScheduler` and the corresponding methods in the `Job` class
- Added the EVENT_JOB_SUBMITTED event that indicates a job has been submitted to its executor.
- Added the EVENT_JOB_MAX_INSTANCES event that indicates a job's execution was skipped due to its maximum number of concurrently running instances being reached
- Added the time zone to the repr() output of `CronTrigger` and `IntervalTrigger`
- Fixed rare race condition on scheduler `shutdown()`
- Dropped official support for CPython 2.6 and 3.2 and PyPy3
- Moved the connection logic in database backed job stores to the `start()` method
- Migrated to setuptools_scm for versioning

- Deprecated the various version related variables in the `apscheduler` module (`apscheduler.version_info`, `apscheduler.version`, `apscheduler.release`, `apscheduler.__version__`)

3.0.6

- Fixed bug in the cron trigger that produced off-by-1-hour datetimes when crossing the daylight saving threshold (thanks to Tim Strazny for reporting)

3.0.5

- Fixed cron trigger always coalescing missed run times into a single run time (contributed by Chao Liu)
- Fixed infinite loop in the cron trigger when an out-of-bounds value was given in an expression
- Fixed debug logging displaying the next wakeup time in the UTC timezone instead of the scheduler's configured timezone
- Allowed unicode function references in Python 2

3.0.4

- Fixed memory leak in the base executor class (contributed by Stefan Nordhausen)

3.0.3

- Fixed compatibility with pymongo 3.0

3.0.2

- Fixed ValueError when the target callable has a default keyword argument that wasn't overridden
- Fixed wrong job sort order in some job stores
- Fixed exception when loading all jobs from the redis job store when there are paused jobs in it
- Fixed AttributeError when printing a job list when there were pending jobs
- Added setuptools as an explicit requirement in install requirements

3.0.1

- A wider variety of target callables can now be scheduled so that the jobs are still serializable (static methods on Python 3.3+, unbound methods on all except Python 3.2)
- Attempting to serialize a non-serializable Job now raises a helpful exception during serialization. Thanks to Jeremy Morgan for pointing this out.
- Fixed table creation with SQLAlchemyJobStore on MySQL/InnoDB
- Fixed start date getting set too far in the future with a timezone different from the local one
- Fixed _run_job_error() being called with the incorrect number of arguments in most executors

3.0.0

- Added support for timezones (special thanks to Curtis Vogt for help with this one)
- Split the old Scheduler class into BlockingScheduler and BackgroundScheduler and added integration for asyncio (PEP 3156), Gevent, Tornado, Twisted and Qt event loops
- Overhauled the job store system for much better scalability
- Added the ability to modify, reschedule, pause and resume jobs
- Dropped the Shelve job store because it could not work with the new job store system
- Dropped the max_runs option and run counting of jobs since it could not be implemented reliably
- Adding jobs is now done exclusively through `add_job()` – the shortcuts to triggers were removed
- Added the `end_date` parameter to cron and interval triggers
- It is now possible to add a job directly to an executor without scheduling, by omitting the trigger argument
- Replaced the thread pool with a pluggable executor system
- Added support for running jobs in subprocesses (via the `processpool` executor)
- Switched from nose to py.test for running unit tests

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

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

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

2.0.1

- Fixed cron style jobs getting wrong default values

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

1.3.1

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

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**.

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

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

1.01

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

Migrating from previous versions of APScheduler

From v3.0 to v3.2

Prior to v3.1, the scheduler inadvertently exposed the ability to fetch and manipulate jobs before the scheduler had been started. The scheduler now requires you to call `scheduler.start()` before attempting to access any of the jobs in the job stores. To ensure that no old jobs are mistakenly executed, you can start the scheduler in paused mode (`scheduler.start(paused=True)`) (introduced in v3.2) to avoid any premature job processing.

From v2.x to v3.0

The 3.0 series is API incompatible with previous releases due to a design overhaul.

Scheduler changes

- The concept of "standalone mode" is gone. For `standalone=True`, use `BlockingScheduler` instead, and for `standalone=False`, use `BackgroundScheduler`. BackgroundScheduler matches the old default semantics.
- Job defaults (like `misfire_grace_time` and `coalesce`) must now be passed in a dictionary as the `job_defaults` option to `configure()`. When supplying an ini-style configuration as the first argument, they will need a corresponding `job_defaults.` prefix.
- The configuration key prefix for job stores was changed from `jobstore.` to `jobstores.` to match the dict-style configuration better.
- The `max_runs` option has been dropped since the run counter could not be reliably preserved when replacing a job with another one with the same ID. To make up for this, the `end_date` option was added to cron and interval triggers.

- The old thread pool is gone, replaced by `ThreadPoolExecutor`. This means that the old `threadpool` options are no longer valid. See *Configuring the scheduler* on how to configure executors.
- The trigger-specific scheduling methods have been removed entirely from the scheduler. Use the generic `add_job()` method or the `scheduled_job()` decorator instead. The signatures of these methods were changed significantly.
- The `shutdown_threadpool` and `close_jobstores` options have been removed from the `shutdown()` method. Executors and job stores are now always shut down on scheduler shutdown.
- `unschedule_job()` and `unschedule_func()` have been replaced by `remove_job()`. You can also unschedule a job by using the job handle returned from `add_job()`.

Job store changes

The job store system was completely overhauled for both efficiency and forwards compatibility. Unfortunately, this means that the old data is not compatible with the new job stores. If you need to migrate existing data from APScheduler 2.x to 3.x, contact the APScheduler author.

The Shelve job store had to be dropped because it could not support the new job store design. Use SQLAlchemyJobStore with SQLite instead.

Trigger changes

From 3.0 onwards, triggers now require a pytz timezone. This is normally provided by the scheduler, but if you were instantiating triggers manually before, then one must be supplied as the `timezone` argument.

The only other backwards incompatible change was that `get_next_fire_time()` takes two arguments now: the previous fire time and the current datetime.

From v1.x to 2.0

There have been some API changes since the 1.x series. This document explains the changes made to v2.0 that are incompatible with the v1.x API.

API changes

- The behavior of cron scheduling with regards to default values for omitted fields has been made more intuitive – omitted fields lower than the least significant explicitly defined field will default to their minimum values except for the week number and weekday fields
- SchedulerShutdownError has been removed – jobs are now added tentatively and scheduled for real when/if the scheduler is restarted
- Scheduler.is_job_active() has been removed – use `job in scheduler.get_jobs()` instead
- dump_jobs() is now print_jobs() and prints directly to the given file or sys.stdout if none is given
- The `repeat` parameter was removed from `add_interval_job()` and `interval_schedule()` in favor of the universal `max_runs` option
- `unschedule_func()` now raises a KeyError if the given function is not scheduled
- The semantics of `shutdown()` have changed – the method no longer accepts a numeric argument, but two booleans

Configuration changes

- The scheduler can no longer be reconfigured while it's running

Contributing to APScheduler

If you wish to add a feature or fix a bug in APScheduler, you need to follow certain procedures and rules to get your changes accepted. This is to maintain the high quality of the code base.

Contribution Process

1. Fork the project on Github
2. Clone the fork to your local machine
3. Make the changes to the project
4. Run the test suite with tox (if you changed any code)
5. Repeat steps 3-4 until the test suite passes
6. Commit if you haven't already
7. Push the changes to your Github fork
8. Make a pull request on Github

There is no need to update the change log – this will be done prior to the next release at the latest. Should the test suite fail even before your changes (which should be rare), make sure you're at least not adding to the failures.

Development Dependencies

To fully run the test suite, you will need at least:

- A MongoDB server
- A Redis server
- A Zookeeper server

For other dependencies, it's best to look in tox.ini and install what is appropriate for the Python version you're using.

Code Style

This project uses PEP 8 rules with a maximum column limit of 120 characters instead of the standard 79. This limit applies to all text files (source code, tests, documentation). In particular, remember to group the imports correctly (standard library imports first, third party libs second, project libraries third, conditional imports last). The PEP 8 checker does not check for this. If in doubt, just follow the surrounding code style as closely as possible.

Testing

Running the test suite is done using the tox utility. This will test the code base against all supported Python versions and checks for PEP 8 violations as well.

Since running the tests on every supported Python version can take quite a long time, it is recommended that during the development cycle py.test is used directly. Before finishing, tox should however be used to make sure the code works on all supported Python versions.

Any nontrivial code changes must be accompanied with the appropriate tests. The tests should not only maintain the coverage, but should test any new functionality or bug fixes reasonably well. If you're fixing a bug, first make sure you have a test which fails against the unpatched codebase and succeeds against the fixed version. Naturally, the test suite has to pass on every Python version. If setting up all the required Python interpreters seems like too much trouble, make sure that it at least passes on the lowest supported versions of both Python 2 and 3.

Extending APScheduler

This document is meant to explain how to develop your custom triggers, job stores, executors and schedulers.

Custom triggers

The built-in triggers cover the needs of the majority of all users. However, some users may need specialized scheduling logic. To that end, the trigger system was made pluggable.

To implement your scheduling logic, subclass `BaseTrigger`. Look at the interface documentation in that class. Then look at the existing trigger implementations. That should give you a good idea what is expected of a trigger implementation.

To use your trigger, you can use `add_job()` like this:

```
trigger = MyTrigger(arg1='foo')
scheduler.add_job(target, trigger)
```

You can also register it as a plugin so you can use can use the alternate form of `add_jobstore`:

```
scheduler.add_job(target, 'my_trigger', arg1='foo')
```

This is done by adding an entry point in your project's `setup.py`:

```
...
entry_points={
      'apscheduler.triggers': ['my_trigger = mytoppackage.subpackage:MyTrigger']
}
```

Custom job stores

If you want to store your jobs in a fancy new NoSQL database, or a totally custom datastore, you can implement your own job store by subclassing `BaseJobStore`.

A job store typically serializes the `Job` objects given to it, and constructs new Job objects from binary data when they are loaded from the backing store. It is important that the job store restores the `_scheduler` and `_jobstore_alias` attribute of any Job that it creates. Refer to existing implementations for examples.

It should be noted that `MemoryJobStore` is special in that it does not deserialize the jobs. This comes with its own problems, which it handles in its own way. If your job store does serialize jobs, you can of course use a serializer other than pickle. You should, however, use the `__getstate__` and `__setstate__` special methods to respectively get and set the Job state. Pickle uses them implicitly.

To use your job store, you can add it to the scheduler like this:

```
jobstore = MyJobStore()
scheduler.add_jobstore(jobstore, 'mystore')
```

You can also register it as a plugin so you can use can use the alternate form of `add_jobstore`:

```
scheduler.add_jobstore('my_jobstore', 'mystore')
```

This is done by adding an entry point in your project's `setup.py`:

```
...
entry_points={
      'apscheduler.jobstores': ['my_jobstore = mytoppackage.subpackage:MyJobStore']
}
```

Custom executors

If you need custom logic for executing your jobs, you can create your own executor classes. One scenario for this would be if you want to use distributed computing to run your jobs on other nodes.

Start by subclassing `BaseExecutor`. The responsibilities of an executor are as follows:

- Performing any initialization when `start()` is called
- Releasing any resources when `shutdown()` is called
- Keeping track of the number of instances of each job running on it, and refusing to run more than the maximum
- Notifying the scheduler of the results of the job

If your executor needs to serialize the jobs, make sure you either use pickle for it, or invoke the `__getstate__` and `__setstate__` special methods to respectively get and set the Job state. Pickle uses them implicitly.

To use your executor, you can add it to the scheduler like this:

```
executor = MyExecutor()
scheduler.add_executor(executor, 'myexecutor')
```

You can also register it as a plugin so you can use can use the alternate form of `add_executor`:

```
scheduler.add_executor('my_executor', 'myexecutor')
```

This is done by adding an entry point in your project's `setup.py`:

```
...
entry_points={
      'apscheduler.executors': ['my_executor = mytoppackage.subpackage:MyExecutor']
}
```

Custom schedulers

A typical situation where you would want to make your own scheduler subclass is when you want to integrate it with your application framework of choice.

Your custom scheduler should always be a subclass of `BaseScheduler`. But if you're not adapting to a framework that relies on callbacks, consider subclassing `BlockingScheduler` instead.

The most typical extension points for scheduler subclasses are:

- **start()** must be overridden to wake up the scheduler for the first time
- **shutdown()** must be overridden to release resources allocated during `start()`
- **wakeup()** must be overridden to manage the timernotify the scheduler of changes in the job store
- **_create_lock()** override if your framework uses some alternate locking implementation (like gevent)
- **_create_default_executor()** override if you need to use an alternative default executor

---

Important: Remember to call the superclass implementations of overridden methods, even abstract ones (unless they're empty).

The most important responsibility of the scheduler subclass is to manage the scheduler's sleeping based on the return values of `_process_jobs()`. This can be done in various ways, including setting timeouts in `wakeup()` or running a blocking loop in `start()`. Again, see the existing scheduler classes for examples.

CHAPTER 2

Indices and tables

- API reference

Index

P

Python Enhancement Proposals
    PEP 3156, 1