# schedule Documentation

*Release 1.2.0*

**Daniel Bader**

**Apr 10, 2023**

# Contents

Python job scheduling for humans. Run Python functions (or any other callable) periodically using a friendly syntax.

- A simple to use API for scheduling jobs, made for humans.
- In-process scheduler for periodic jobs. No extra processes needed!
- Very lightweight and no external dependencies.
- Excellent test coverage.
- Tested on Python 3.7, 3.8, 3.9, 3.10 and 3.11

# CHAPTER 1

# Example

```
$ 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().day.at("12:42", "Europe/Amsterdam").do(job)
schedule.every().minute.at(":17").do(job)

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

More *Examples*

CHAPTER 2

# When **not** to use Schedule

Let's be honest, Schedule is not a 'one size fits all' scheduling library. This library is designed to be a simple solution for simple scheduling problems. You should probably look somewhere else if you need:

- Job persistence (remember schedule between restarts)
- Exact timing (sub-second precision execution)
- Concurrent execution (multiple threads)
- Localization (workdays or holidays)

**Schedule does not account for the time it takes for 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 *Parallel execution* for a sample implementation.

CHAPTER 3

# Read More

## 3.1 Installation

### 3.1.1 Python version support

We recommend using the latest version of Python. Schedule is tested on Python 3.7, 3.8, 3.9, 3.10 and 3.11.

Want to use Schedule on earlier Python versions? See the History.

### 3.1.2 Dependencies

Schedule has no dependencies. None. Zero. Nada. Nopes. We plan to keep it that way.

### 3.1.3 Installation instructions

Problems? Check out *Frequently Asked Questions*.

#### PIP (preferred)

The recommended way to install this package is to use pip. Use the following command to install it:

```
$ pip install schedule
```

Schedule is now installed. Check out the *examples* or go to the *the documentation overview*.

#### Using another package manager

Schedule is available through some linux package managers. These packages are not maintained by the maintainers of this project. It cannot be guarantee that these packages are up-to-date (and will stay up-to-date) with the latest released version. If you don't mind having an old version, you can use it.

### Ubuntu

**OUTDATED!** At the time of writing, the packages for 20.04LTS and below use version 0.3.2 (2015).

```
$ apt-get install python3-schedule
```

See package page.

### Debian

**OUTDATED!** At the time of writing, the packages for buster and below use version 0.3.2 (2015).

```
$ apt-get install python3 schedule
```

See package page.

### Arch

On the Arch Linux User repository (AUR) the package is available using the name *python-schedule*. See the package page here. For yay users, run:

```
$ yay -S python-schedule
```

### Conda (Anaconda)

Schedule is published in conda (the Anaconda package manager).

For installation instructions, visit the conda-forge Schedule repo. The release of Schedule on conda is maintained by the conda-forge project.

### Install manually

If you don't have access to a package manager or need more control, you can manually copy the library into your project. This is easy as the schedule library consists of a single sourcefile MIT licenced. However, this method is highly discouraged as you won't receive automatic updates.

1. Go to the Github repo.
2. Open file *schedule/__init__.py* and copy the code.
3. In your project, create a packaged named *schedule* and paste the code in a file named *__init__.py*.

## 3.2 Examples

Eager to get started? This page gives a good introduction to Schedule. It assumes you already have Schedule installed. If you do not, head over to *Installation*.

### 3.2.1 Run a job every x minute

```
import schedule
import time

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

# Run job every 3 second/minute/hour/day/week,
# Starting 3 second/minute/hour/day/week from now
schedule.every(3).seconds.do(job)
schedule.every(3).minutes.do(job)
schedule.every(3).hours.do(job)
schedule.every(3).days.do(job)
schedule.every(3).weeks.do(job)

# Run job every minute at the 23rd second
schedule.every().minute.at(":23").do(job)

# Run job every hour at the 42nd minute
schedule.every().hour.at(":42").do(job)

# Run jobs every 5th hour, 20 minutes and 30 seconds in.
# If current time is 02:00, first execution is at 06:20:30
schedule.every(5).hours.at("20:30").do(job)

# Run job every day at specific HH:MM and next HH:MM:SS
schedule.every().day.at("10:30").do(job)
schedule.every().day.at("10:30:42").do(job)
schedule.every().day.at("12:42", "Europe/Amsterdam").do(job)

# Run job on a specific day of the week
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)
```

### 3.2.2 Use a decorator to schedule a job

Use the `@repeat` to schedule a function. Pass it an interval using the same syntax as above while omitting the `.do()`.

```
from schedule import every, repeat, run_pending
import time

@repeat(every(10).minutes)
def job():
    print("I am a scheduled job")

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

The `@repeat` decorator does not work on non-static class methods.

### 3.2.3 Pass arguments to a job

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

```
import schedule

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

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

from schedule import every, repeat

@repeat(every().second, "World")
@repeat(every().day, "Mars")
def hello(planet):
    print("Hello", planet)
```

### 3.2.4 Cancel a job

To remove a job from the scheduler, use the `schedule.cancel_job(job)` method

```
import schedule

def some_task():
    print('Hello world')

job = schedule.every().day.at('22:30').do(some_task)
schedule.cancel_job(job)
```

### 3.2.5 Run a job once

Return `schedule.CancelJob` from a job to remove it from the scheduler.

```
import schedule
import time

def job_that_executes_once():
    # Do some work that only needs to happen once...
    return schedule.CancelJob

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

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

### 3.2.6 Get all jobs

To retrieve all jobs from the scheduler, use `schedule.get_jobs()`

```
import schedule

def hello():
    print('Hello world')

schedule.every().second.do(hello)

all_jobs = schedule.get_jobs()
```

### 3.2.7 Cancel all jobs

To remove all jobs from the scheduler, use `schedule.clear()`

```
import schedule

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

schedule.every().second.do(greet)

schedule.clear()
```

### 3.2.8 Get several jobs, filtered by tags

You can retrieve a group of jobs from the scheduler, selecting them by a unique identifier.

```
import schedule

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')

friends = schedule.get_jobs('friend')
```

Will return a list of every job tagged as `friend`.

### 3.2.9 Cancel several jobs, filtered by tags

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

```
import schedule

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')
```

(continues on next page)

(continued from previous page)

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

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

### 3.2.10 Run a job at random intervals

```
def my_job():
    print('Foo')

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

`every(A).to(B).seconds` executes the job function every N seconds such that A <= N <= B.

### 3.2.11 Run a job until a certain time

```
import schedule
from datetime import datetime, timedelta, time

def job():
    print('Boo')

# run job until a 18:30 today
schedule.every(1).hours.until("18:30").do(job)

# run job until a 2030-01-01 18:33 today
schedule.every(1).hours.until("2030-01-01 18:33").do(job)

# Schedule a job to run for the next 8 hours
schedule.every(1).hours.until(timedelta(hours=8)).do(job)

# Run my_job until today 11:33:42
schedule.every(1).hours.until(time(11, 33, 42)).do(job)

# run job until a specific datetime
schedule.every(1).hours.until(datetime(2020, 5, 17, 11, 36, 20)).do(job)
```

The `until` method sets the jobs deadline. The job will not run after the deadline.

### 3.2.12 Time until the next execution

Use `schedule.idle_seconds()` to get the number of seconds until the next job is scheduled to run. The returned value is negative if the next scheduled jobs was scheduled to run in the past. Returns `None` if no jobs are scheduled.

```
import schedule
import time

def job():
    print('Hello')

schedule.every(5).seconds.do(job)
```

(continues on next page)

(continued from previous page)

```
while 1:
    n = schedule.idle_seconds()
    if n is None:
        # no more jobs
        break
    elif n > 0:
        # sleep exactly the right amount of time
        time.sleep(n)
    schedule.run_pending()
```

### 3.2.13 Run all jobs now, regardless of their scheduling

To run all jobs regardless if they are scheduled to run or not, use `schedule.run_all()`. Jobs are re-scheduled after finishing, just like they would if they were executed using `run_pending()`.

```
import schedule

def job_1():
    print('Foo')

def job_2():
    print('Bar')

schedule.every().monday.at("12:40").do(job_1)
schedule.every().tuesday.at("16:40").do(job_2)

schedule.run_all()

# Add the delay_seconds argument to run the jobs with a number
# of seconds delay in between.
schedule.run_all(delay_seconds=10)
```

## 3.3 Run in the background

Out of the box it is not possible to run the schedule in the background. However, you can create a thread yourself and use it to run jobs without blocking the main thread. This is an example of how you could do this:

```
import threading
import time

import schedule


def run_continuously(interval=1):
    """Continuously run, while executing pending jobs at each
    elapsed time interval.
    @return cease_continuous_run: threading. Event which can
    be set to cease continuous run. Please note that it is
    *intended behavior that run_continuously() does not run
    missed jobs*. For example, if you've registered a job that
    should run every minute and you set a continuous run
```

(continues on next page)

(continued from previous page)

```
    interval of one hour then your job won't be run 60 times
    at each interval but only once.
    """
    cease_continuous_run = threading.Event()

    class ScheduleThread(threading.Thread):
        @classmethod
        def run(cls):
            while not cease_continuous_run.is_set():
                schedule.run_pending()
                time.sleep(interval)

    continuous_thread = ScheduleThread()
    continuous_thread.start()
    return cease_continuous_run


def background_job():
    print('Hello from the background thread')


schedule.every().second.do(background_job)

# Start the background thread
stop_run_continuously = run_continuously()

# Do some other things...
time.sleep(10)

# Stop the background thread
stop_run_continuously.set()
```

## 3.4 Parallel execution

*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 limitation 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)
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)


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 time
import threading
import schedule
import queue

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.

## 3.5 Timezone & Daylight Saving Time

### 3.5.1 Timezone in .at()

Schedule supports setting the job execution time in another timezone using the `.at` method.

**To work with timezones** pytz **must be installed!** Get it:

```
pip install pytz
```

Timezones are only available in the `.at` function, like so:

```
# Pass a timezone as a string
schedule.every().day.at("12:42", "Europe/Amsterdam").do(job)

# Pass an pytz timezone object
from pytz import timezone
schedule.every().friday.at("12:42", timezone("Africa/Lagos")).do(job)
```

Schedule uses the timezone to calculate the next runtime in local time. All datetimes inside the library are stored naive. This causes the `next_run` and `last_run` to always be in Pythons local timezone.

### 3.5.2 Daylight Saving Time

When scheduling jobs with relative time (that is when not using `.at()`), daylight saving time (DST) is **not** taken into account. A job that is set to run every 4 hours might execute after 3 realtime hours when DST goes into effect. This is because schedule is timezone-unaware for relative times.

However, when using `.at()`, DST **is** handed correctly: the job will always run at (or close after) the set timestamp. A job scheduled during a moment that is skipped, the job will execute after the clock is moved. For example, a job is scheduled `.at("02:30")`, clock moves from `02:00` to `03:00`, the job will run at `03:00`.

### 3.5.3 Example

Let's say we are in `Europe/Berlin` and local datetime is `2022 march 20, 10:00:00`. At the moment daylight saving time is not in effect in Berlin (UTC+1).

We schedule a job to run every day at 10:30:00 in America/New_York. At this time, daylight saving time is in effect in New York (UTC-4).

```
s = every().day.at("10:30", "America/New_York").do(job)
```

Because of the 5 hour time difference between Berlin and New York the job should effectively run at `15:30:00`. So the next run in Berlin time is `2022 march 20, 15:30:00`:

```
print(s.next_run)
# 2022-03-20 15:30:00

print(repr(s))
# Every 1 day at 10:30:00 do job() (last run: [never], next run: 2022-03-20 15:30:00)
```

## 3.6 Exception Handling

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):
```

(continues on next page)

(continued from previous page)

```
            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.

## 3.7 Logging

Schedule logs messages to the Python logger named `schedule` at `DEBUG` level. To receive logs from Schedule, set the logging level to `DEBUG`.

```
import schedule
import logging

logging.basicConfig()
schedule_logger = logging.getLogger('schedule')
schedule_logger.setLevel(level=logging.DEBUG)

def job():
    print("Hello, Logs")

schedule.every().second.do(job)

schedule.run_all()

schedule.clear()
```

This will result in the following log messages:

```
DEBUG:schedule:Running *all* 1 jobs with 0s delay in between
DEBUG:schedule:Running job Job(interval=1, unit=seconds, do=job, args=(), kwargs={})
Hello, Logs
DEBUG:schedule:Deleting *all* jobs
```

### 3.7.1 Customize logging

The easiest way to add reusable logging to jobs is to implement a decorator that handles logging. As an example, below code adds the `print_elapsed_time` decorator:

```
import functools
import time
import schedule

# This decorator can be applied to any job function to log the elapsed time of each␣
↪job
def print_elapsed_time(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_timestamp = time.time()
        print('LOG: Running job "%s"' % func.__name__)
        result = func(*args, **kwargs)
        print('LOG: Job "%s" completed in %d seconds' % (func.__name__, time.time() -␣
↪start_timestamp))
        return result

    return wrapper


@print_elapsed_time
def job():
    print('Hello, Logs')
    time.sleep(5)

schedule.every().second.do(job)

schedule.run_all()
```

This outputs:

```
LOG: Running job "job"
Hello, Logs
LOG: Job "job" completed in 5 seconds
```

## 3.8 Multiple schedulers

You can run as many jobs from a single scheduler as you wish. However, for larger installations it might be desirable to have multiple schedulers. This is supported:

```
import time
import schedule

def fooJob():
    print("Foo")

def barJob():
    print("Bar")

# Create a new scheduler
scheduler1 = schedule.Scheduler()

# Add jobs to the created scheduler
scheduler1.every().hour.do(fooJob)
scheduler1.every().hour.do(barJob)
```

(continues on next page)

(continued from previous page)

```
# Create as many schedulers as you need
scheduler2 = schedule.Scheduler()
scheduler2.every().second.do(fooJob)
scheduler2.every().second.do(barJob)

while True:
    # run_pending needs to be called on every scheduler
    scheduler1.run_pending()
    scheduler2.run_pending()
    time.sleep(1)
```

## 3.9 Frequently Asked Questions

Frequently asked questions on the usage of schedule. Did you get here using an 'old' link and expected to see more questions?

### 3.9.1 AttributeError: 'module' object has no attribute 'every'

I'm getting

```
AttributeError: 'module' object has no attribute 'every'
```

when I try to use schedule.

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.

### 3.9.2 ModuleNotFoundError: No module named 'schedule'

It seems python can't find the schedule package. Let's check some common causes.

Did you install schedule? If not, follow *Installation*. Validate installation:

- Did you install using pip? Run `pip3 list | grep schedule`. This should return `schedule 0.6.0` (or a higher version number)
- Did you install using apt? Run `dpkg -l | grep python3-schedule`. This should return something along the lines of `python3-schedule 0.3.2-1.1 Job scheduling for humans (Python 3)` (or a higher version number)

Are you used python 3 to install Schedule, and are running the script using python 3? For example, if you installed schedule using a version of pip that uses Python 2, and your code runs in Python 3, the package won't be found. In this case the solution is to install Schedule using pip3: `pip3 install schedule`.

Are you using virtualenv? Check that you are running the script inside the same virtualenv where you installed schedule.

Is this problem occurring when running the program from inside and IDE like PyCharm or VSCode? Try to run your program from a commandline outside of the IDE. If it works there, the problem is with your IDE configuration. It might be that your IDE uses a different Python interpreter installation.

Still having problems? Use Google and StackOverflow before submitting an issue.

### 3.9.3 ModuleNotFoundError: ModuleNotFoundError: No module named 'pytz'

This error happens when you try to set a timezone in `.at()` without having the pytz package installed. Pytz is a required dependency when working with timezones. To resolve this issue, install the `pytz` module by running `pip install pytz`.

### 3.9.4 Does schedule support time zones?

Yes! See *Timezones*.

### 3.9.5 What if my task throws an exception?

See *Exception Handling*.

### 3.9.6 How can I run a job only once?

See *Examples*.

### 3.9.7 How can I cancel several jobs at once?

See *Examples*.

### 3.9.8 How to execute jobs in parallel?

See *Parallel Execution*.

### 3.9.9 How to continuously run the scheduler without blocking the main thread?

*Background Execution*.

### 3.9.10 Another question?

If you are left with an unanswered question, browse the issue tracker to see if your question has been asked before. Feel free to create a new issue if that's not the case. Thank you

## 3.10 Reference

This part of the documentation covers all the interfaces of schedule.

### 3.10.1 Main Interface

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

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

schedule.**every**(*interval: int = 1*) → schedule.Job
Calls *every* on the *default scheduler instance*.

schedule.**run_pending**() → None
Calls *run_pending* on the *default scheduler instance*.

schedule.**run_all**(*delay_seconds: int = 0*) → None
Calls *run_all* on the *default scheduler instance*.

schedule.**get_jobs**(*tag: Optional[collections.abc.Hashable] = None*) → List[schedule.Job]
Calls *get_jobs* on the *default scheduler instance*.

schedule.**clear**(*tag: Optional[collections.abc.Hashable] = None*) → None
Calls *clear* on the *default scheduler instance*.

schedule.**cancel_job**(*job: schedule.Job*) → None
Calls *cancel_job* on the *default scheduler instance*.

schedule.**next_run**(*tag: Optional[collections.abc.Hashable] = None*) → Optional[datetime.datetime]
Calls *next_run* on the *default scheduler instance*.

schedule.**idle_seconds**() → Optional[float]
Calls *idle_seconds* on the *default scheduler instance*.

### 3.10.2 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**() → None
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: int = 0*) → None
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

**get_jobs**(*tag: Optional[collections.abc.Hashable] = None*) → List[schedule.Job]
Gets 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 retrieve

**clear**(*tag: Optional[collections.abc.Hashable] = None*) → 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: schedule.Job*) → None
Delete a scheduled job.

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

**every**(*interval: int = 1*) → schedule.Job
Schedule a new periodic job.

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

**Returns** An unconfigured *Job*

**get_next_run**(*tag: Optional[collections.abc.Hashable] = None*) → Optional[datetime.datetime]
Datetime when the next job should run.

**Parameters** **tag** – Filter the next run for the given tag parameter

**Returns** A `datetime` object or None if no jobs scheduled

**next_run**
Datetime when the next job should run.

**Parameters** **tag** – Filter the next run for the given tag parameter

**Returns** A `datetime` object or None if no jobs scheduled

**idle_seconds**

**Returns** Number of seconds until *next_run* or None if no jobs are scheduled

**class** schedule.**Job**(*interval: int*, *scheduler: Optional[schedule.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 identifiers.

Tags must be hashable. Duplicate tags are discarded.

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

**Returns** The invoked job instance

**at**(*time_str: str*, *tz: Optional[str] = None*)
Specify a particular time that the job should be run at.

**Parameters**

- **time_str** – A string in one of the following formats:
  - For daily jobs -> *HH:MM:SS* or *HH:MM*
  - For hourly jobs -> *MM:SS* or *:MM*
  - For minute jobs -> *: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')*).
- **tz** – The timezone that this timestamp refers to. Can be a string that can be parsed by pytz.timezone(), or a pytz.BaseTzInfo object

**Returns** The invoked job instance

**to**(*latest: int*)
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

**until**(*until_time: Union[datetime.datetime, datetime.timedelta, datetime.time, str]*)
Schedule job to run until the specified moment.

The job is canceled whenever the next run is calculated and it turns out the next run is after the until_time. The job is also canceled right before it runs, if the current time is after until_time. This latter case can happen when the the job was scheduled to run before until_time, but runs after until_time.

If until_time is a moment in the past, ScheduleValueError is thrown.

**Parameters** **until_time** – A moment in the future representing the latest time a job can be run. If only a time is supplied, the date is set to today. The following formats are accepted:

- datetime.datetime
- datetime.timedelta
- datetime.time
- String in one of the following formats: "%Y-%m-%d %H:%M:%S", "%Y-%m-%d %H:%M", "%Y-%m-%d", "%H:%M:%S", "%H:%M" as defined by strptime() behaviour. If an invalid string format is passed, ScheduleValueError is thrown.

**Returns** The invoked job instance

**do**(*job_func: Callable*, **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. If the job's deadline is reached (configured using .until()), the job is not run and CancelJob is returned immediately. If the next scheduled run exceeds the job's deadline, CancelJob is returned after the execution. In this latter case CancelJob takes priority over any other returned value.

**Returns** The return value returned by the *job_func*, or CancelJob if the job's deadline is reached.

### 3.10.3 Exceptions

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

## 3.11 Development

These instructions are geared towards people who want to help develop this library.

### 3.11.1 Preparing for development

All required tooling and libraries can be installed using the `requirements-dev.txt` file:

```
pip install -r requirements-dev.txt
```

### 3.11.2 Running tests

`pytest` is used to run tests. Run all tests with coverage and formatting checks:

```
py.test test_schedule.py --flake8 schedule -v --cov schedule --cov-report term-missing
```

### 3.11.3 Formatting the code

This project uses black formatter. To format the code, run:

```
black .
```

Make sure you use version 20.8b1 of black.

### 3.11.4 Compiling documentation

The documentation is written in reStructuredText. It is processed using Sphinx using the alabaster theme. After installing the development requirements it is just a matter of running:

```
cd docs
make html
```

The resulting html can be found in `docs/_build/html`

### 3.11.5 Publish a new version

Update the `HISTORY.rst` and `AUTHORS.rst` files. Bump the version in `setup.py` and `docs/conf.py`. Merge these changes into master. Finally:

```
git tag X.Y.Z -m "Release X.Y.Z"
git push --tags

pip install --upgrade setuptools twine wheel
python setup.py sdist bdist_wheel --universal
twine upload --repository schedule dist/*
```

This project follows semantic versioning.‘

## 3.12 History

### 3.12.1 1.2.0 (2023-04-10)

- Dropped support for Python 3.6, add support for Python 3.10 and 3.11.
- Add timezone support for .at(). See #517. Thanks @chrimaho!
- Get next run by tag (#463) Thanks @jweijers!
- Add py.typed file. See #521. Thanks @Akuli!
- Fix the re pattern of the ‘days’. See #506 Thanks @sunpro108!
- Fix test_until_time failure when run early. See #563. Thanks @emollier!
- Fix crash repr on partially constructed job. See #569. Thanks @CPickens42!
- Code cleanup and modernization. See #567, #536. Thanks @masa-08 and @SergBobrovsky!
- Documentation improvements and fix typos. See #469, #479, #493, #519, #520. Thanks to @NaelsonDouglas, @chrimaho, @rudSarkar

### 3.12.2 1.1.0 (2021-04-09)

- Added @repeat() decorator. See #148. Thanks @rhagenaars!
- Added execute .until(). See #195. Thanks @fredthomsen!
- Added job retrieval filtered by tags using get_jobs('tag'). See #419. Thanks @skenvy!
- Added type annotations. See #427. Thanks @martinthoma!
- Bugfix: str() of job when there is no __name__. See #430. Thanks @biggerfisch!
- Improved error messages. See #280, #439. Thanks @connorskees and @sosolidkk!
- Improved logging. See #193. Thanks @zcking!
- Documentation improvements and fix typos. See #424, #435, #436, #453, #437, #448. Thanks @ebllg!

### 3.12.3 1.0.0 (2021-01-20)

Depending on your configuration, the following bugfixes might change schedule's behaviour:

- Fix: idle_seconds crashes when no jobs are scheduled. See #401. Thanks @yoonghm!
- Fix: day.at('HH:MM:SS') where HMS=now+10s doesn't run today. See #331. Thanks @qmorek!
- Fix: hour.at('MM:SS'), the seconds are set to 00. See #290. Thanks @eladbi!
- Fix: Long-running jobs skip a day when they finish in the next day #404. Thanks @4379711!

Other changes:

- Dropped Python 2.7 and 3.5 support, added 3.8 and 3.9 support. See #409
- Fix RecursionError when the job is passed to the do function as an arg. See #190. Thanks @connorskees!
- Fix DeprecationWarning of 'collections'. See #296. Thanks @gaguirregabiria!
- Replaced Travis with Github Actions for automated testing
- Revamp and extend documentation. See #395
- Improved tests. Thanks @connorskees and @Jamim!
- Changed log messages to DEBUG level. Thanks @aisk!

### 3.12.4 0.6.0 (2019-01-20)

- Make at() accept timestamps with 1 second precision (#267). Thanks @NathanWailes!
- Introduce proper exception hierarchy (#271). Thanks @ConnorSkees!

### 3.12.5 0.5.0 (2017-11-16)

- Keep partially scheduled jobs from breaking the scheduler (#125)
- Add support for random intervals (Thanks @grampajoe and @gilbsgilbs)

### 3.12.6 0.4.3 (2017-06-10)

- Improve docs & clean up docstrings

### 3.12.7 0.4.2 (2016-11-29)

- Publish to PyPI as a universal (py2/py3) wheel

### 3.12.8 0.4.0 (2016-11-28)

- Add proper HTML (Sphinx) docs available at https://schedule.readthedocs.io/
- CI builds now run against Python 2.7 and 3.5 (3.3 and 3.4 should work fine but are untested)
- Fixed an issue with `run_all()` and having more than one job that deletes itself in the same iteration. Thanks @alaingilbert.
- Add ability to tag jobs and to cancel jobs by tag. Thanks @Zerrossetto.
- Improve schedule docs. Thanks @Zerrossetto.
- Additional docs fixes by @fkromer and @yetingsky.

### 3.12.9 0.3.2 (2015-07-02)

- Fixed issues where scheduling a job with a functools.partial as the job function fails. Thanks @dylwhich.
- Fixed an issue where scheduling a job to run every >= 2 days would cause the initial execution to happen one day early. Thanks @WoLfulus for identifying this and providing a fix.
- Added a FAQ item to describe how to schedule a job that runs only once.

### 3.12.10 0.3.1 (2014-09-03)

- Fixed an issue with unicode handling in setup.py that was causing trouble on Python 3 and Debian (https://github.com/dbader/schedule/issues/27). Thanks to @waghanza for reporting it.
- Added an FAQ item to describe how to deal with job functions that throw exceptions. Thanks @mplewis.

### 3.12.11 0.3.0 (2014-06-14)

- Added support for scheduling jobs on specific weekdays. Example: `schedule.every().tuesday.do(job)` or `schedule.every().wednesday.at("13:15").do(job)` (Thanks @abultman.)
- Run tests against Python 2.7 and 3.4. Python 3.3 should continue to work but we're not actively testing it on CI anymore.

### 3.12.12 0.2.1 (2013-11-20)

- Fixed history (no code changes).

### 3.12.13 0.2.0 (2013-11-09)

- This release introduces two new features in a backwards compatible way:
- Allow jobs to cancel repeated execution: Jobs can be cancelled by calling `schedule.cancel_job()` or by returning `schedule.CancelJob` from the job function. (Thanks to @cfrco and @matrixise.)

- Updated `at_time()` to allow running jobs at a particular time every hour. Example: `every().hour.at(':15').do(job)` will run `job` 15 minutes after every full hour. (Thanks @mattss.)
- Refactored unit tests to mock `datetime` in a cleaner way. (Thanks @matts.)

### 3.12.14 0.1.11 (2013-07-30)

- Fixed an issue with `next_run()` throwing a `ValueError` exception when the job queue is empty. Thanks to @dpagano for pointing this out and thanks to @mrhwick for quickly providing a fix.

### 3.12.15 0.1.10 (2013-06-07)

- Fixed issue with `at_time` jobs not running on the same day the job is created (Thanks to @mattss)

### 3.12.16 0.1.9 (2013-05-27)

- Added `schedule.next_run()`
- Added `schedule.idle_seconds()`
- Args passed into `do()` are forwarded to the job function at call time
- Increased test coverage to 100%

### 3.12.17 0.1.8 (2013-05-21)

- Changed default `delay_seconds` for `schedule.run_all()` to 0 (from 60)
- Increased test coverage

### 3.12.18 0.1.7 (2013-05-20)

- API change: renamed `schedule.run_all_jobs()` to `schedule.run_all()`
- API change: renamed `schedule.run_pending_jobs()` to `schedule.run_pending()`
- API change: renamed `schedule.clear_all_jobs()` to `schedule.clear()`
- Added `schedule.jobs`

### 3.12.19 0.1.6 (2013-05-20)

- Fix packaging
- README fixes

### 3.12.20 0.1.4 (2013-05-20)

- API change: renamed `schedule.tick()` to `schedule.run_pending_jobs()`
- Updated README and `setup.py` packaging

### 3.12.21 0.1.0 (2013-05-19)

- Initial release

CHAPTER 4

# Issues

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

CHAPTER 5

# About Schedule

Created by Daniel Bader - @dbader_org

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

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>
- qmorek <https://github.com/qmorek>

- aisk <https://github.com/aisk>
- MichaelCorleoneLi <https://github.com/MichaelCorleoneLi>
- sijmenhuizenga <https://github.com/SijmenHuizenga>
- eladbi <https://github.com/eladbi>
- chankeypathak <https://github.com/chankeypathak>
- vubon <https://github.com/vubon>
- gaguirregabiria <https://github.com/gaguirregabiria>
- rhagenaars <https://github.com/RHagenaars>
- Skenvy <https://github.com/skenvy>
- zcking <https://github.com/zcking>
- Martin Thoma <https://github.com/MartinThoma>
- ebllg <https://github.com/ebllg>
- fredthomsen <https://github.com/fredthomsen>
- biggerfisch <https://github.com/biggerfisch>
- sosolidkk <https://github.com/sosolidkk>
- rudSarkar <https://github.com/rudSarkar>
- chrimaho <https://github.com/chrimaho>
- jweijers <https://github.com/jweijers>
- Akuli <https://github.com/Akuli>
- NaelsonDouglas <https://github.com/NaelsonDouglas>
- SergBobrovsky <https://github.com/SergBobrovsky>
- CPickens42 <https://github.com/CPickens42>
- emollier <https://github.com/emollier>
- sunpro108 <https://github.com/sunpro108>

# Python Module Index

## S

# Index