12. Time Operations - Python in a Nutshell, 3rd Edition

Chapter 12. Time Operations

A Python program can handle time in several ways. Time intervals are floating-point numbers in units of seconds (a fraction of a second is the fractional part of the interval): all standard library functions accepting an argument that expresses a time interval in seconds accept a float as the value of that argument. Instants in time are expressed in seconds since a reference instant, known as the *epoch*. (Midnight, UTC, of January 1, 1970, is a popular epoch used on both Unix and Windows platforms.) Time instants often also need to be expressed as a mixture of units of measurement (e.g., years, months, days, hours, minutes, and seconds), particularly for I/O purposes. I/O, of course, also requires the ability to format times and dates into human-readable strings, and parse them back from string formats.

This chapter covers the `time` module, which supplies Python's core time-handling functionality. The `time` module is somewhat dependent on the underlying system's C library. The chapter also presents the `datetime`, `sched`, and `calendar` modules from the standard Python library, and the third-party modules `dateutil` and `pytz`.

The time Module

The underlying C library determines the range of dates that the `time` module can handle. On Unix systems, years 1970 and 2038 are typical cut-off points, a limitation that `datetime` avoids. Time instants are normally specified in UTC (Coordinated Universal Time, once known as GMT, or Greenwich Mean Time). The `time` module also supports local time zones and daylight saving time (DST), but only to the extent the underlying C system library does.

As an alternative to seconds since the epoch, a time instant can be represented by a tuple of nine integers, called a *timetuple*. (Timetuples are covered in Table 12-1.) All items are integers: timetuples don't keep track of fractions of a second. A timetuple is an instance of `struct_time`. You can use it as a tuple, and you can also access the items as the read-only attributes `x.tm_year`, `x.tm_mon`, and so on, with the attribute names listed in Table 12-1. Wherever a function requires a timetuple argument, you can pass an instance of `struct_time` or any other sequence whose items are nine integers in the right ranges (all ranges in the table include both lower and upper bounds).

Table 12-1. Tuple form of time representation

Item	Meaning	Field name	Range	Notes
0	Year	`tm_year`	1970–2038	Wider on some platforms.
1	Month	`tm_mon`	1–12	`1` is January; `12` is December.
2	Day	`tm_mday`	1–31	
3	Hour	`tm_hour`	0–23	`0` is midnight; `12` is noon.
4	Minute	`tm_min`	0–59	
5	Second	`tm_sec`	0–61	`60` and `61` for leap seconds.
6	Weekday	`tm_wday`	0–6	`0` is Monday; `6` is Sunday.

Item	Meaning	Field name	Range	Notes
7	Year day	`tm_yday`	1–366	Day number within the year.
8	DST flag	`tm_isdst`	–1 to 1	`-1` means library determines DST.

To translate a time instant from a “seconds since the epoch” floating-point value into a timetuple, pass the floating-point value to a function (e.g., `localtime`) that returns a timetuple with all nine items valid. When you convert in the other direction, `mktime` ignores redundant items six (`tm_wday`) and seven (`tm_yday`) of the tuple. In this case, you normally set item eight (`tm_isdst`) to `-1` so that `mktime` itself determines whether to apply DST.

`time` supplies the functions and attributes listed in Table 12-2.

Table 12-2.

asctime	`asctime([tupletime])` Accepts a timetuple and returns a readable 24-character string such as `'Sun Jan 8 14:41:06 2017'`. `asctime()` without arguments is like `asctime(localtime(time()))` (formats current time in local time).
clock	`clock()` Returns the current CPU time as a floating-point number of seconds, but is platform dependent. Deprecated in v3. To measure computational costs of different approaches, use the standard library module `timeit`, covered in “The timeit module” instead. To implement a timeout or schedule events, in v3, use `perf_counter()` or `process_time()` instead. See also the `sched` module for multithreading safe event scheduling.
ctime	`ctime([secs])` Like `asctime(localtime(secs))`, accepts an instant expressed in seconds since the epoch and returns a readable 24-character string form of that instant, in local time. `ctime()` without arguments is like `asctime()` (formats the current time instant in local time).
gmtime	`gmtime([secs])` Accepts an instant expressed in seconds since the epoch and returns a timetuple `t` with the UTC time (`t.tm_isdst` is always `0`). `gmtime()` without arguments is like `gmtime(time())` (returns the timetuple for the current time instant).
localtime	`localtime([secs])` Accepts an instant expressed in seconds since the epoch and returns a timetuple `t` with the local time (`t.tm_isdst` is `0` or `1`, depending on whether DST applies to instant `secs` by local rules). `localtime()` without arguments is like `localtime(time())` (returns the timetuple for the current time instant).
mktime	`mktime(tupletime)` Accepts an instant expressed as a timetuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch. DST, the last item in `tupletime`, is meaningful: set it to `0` to get solar time, to `1` to get DST, or to `-1` to let `mktime` compute whether DST is in effect at the given instant.

monotonic	`monotonic()` v3 only. Like `time()`, returns the current time instant, a `float` with seconds since the epoch. Guaranteed to never go backward between calls, even when the system clock is adjusted (e.g., due to leap seconds).
perf_counter	`perf_counter()` v3 only. Returns the value in fractional seconds using the highest-resolution clock available to get accuracy for short durations. It is system-wide and *includes* time elapsed during `sleep`. Use only the difference between successive calls, as there is no defined reference point.
process_time	`process_time()` v3 only. Returns the value in fractional seconds using the highest-resolution clock available to get accuracy for short durations. It is process-wide and *doesn't* include time elapsed during `sleep`. Use only the difference between successive calls, as there is no defined reference point.
sleep	`sleep(secs)` Suspends the calling thread for `secs` seconds. The calling thread may start executing again before `secs` seconds (when it's the main thread and some signal wakes it up) or after a longer suspension (depending on system scheduling of processes and threads). You can call sleep with `secs=0` to offer other threads a chance to run, incurring no significant delay if the current thread is the only one ready to run.
strftime	`strftime(fmt[,tupletime])` Accepts an instant expressed as a timetuple in local time and returns a string representing the instant as specified by string `fmt`. If you omit `tupletime`, `strftime` uses `localtime(time())` (formats the current time instant). The syntax of string `format` is similar to the one covered in "Legacy String Formatting with `%`". Conversion characters are different, as shown in Table 12-3. Refer to the time instant specified by `tupletime`; the format can't specify width and precision.

Table 12-3. Conversion characters for strftime

Type char	Meaning	Special notes
`a`	Weekday name, abbreviated	Depends on locale
`A`	Weekday name, full	Depends on locale
`b`	Month name, abbreviated	Depends on locale
`B`	Month name, full	Depends on locale
`c`	Complete date and time representation	Depends on locale
`d`	Day of the month	Between 1 and 31
`G`	***New in 3.6***: ISO 8601:2000 standard week-based year number	
`H`	Hour (24-hour clock)	Between 0 and 23
`I`	Hour (12-hour clock)	Between 1 and 12

Type char	Meaning	Special notes
`j`	Day of the year	Between 1 and 366
`m`	Month number	Between 1 and 12
`M`	Minute number	Between 0 and 59
`p`	A.M. or P.M. equivalent	Depends on locale
`S`	Second number	Between 0 and 61
`u`	***New in 3.6***: day of week	Numbered from Monday == 1
`U`	Week number (Sunday first weekday)	Between 0 and 53
`V`	***New in 3.6***: ISO 8601:2000 standard week-based week number	
`w`	Weekday number	0 is Sunday, up to 6
`W`	Week number (Monday first weekday)	Between 0 and 53
`x`	Complete date representation	Depends on locale
`X`	Complete time representation	Depends on locale
`y`	Year number within century	Between 0 and 99
`Y`	Year number	1970 to 2038, or wider
`Z`	Name of time zone	Empty if no time zone exists
`%`	A literal `%` character	Encoded as `%%`

For example, you can obtain dates just as formatted by `asctime` (e.g., `'Tue Dec 10 18:07:14 2002'`) with the format string:

```
          %H:%M:%S
'%a %b %d%Y'
```

You can obtain dates compliant with RFC 822 (e.g., `'Tue, 10 Dec 2002 18:07:14 EST'`) with the format string:

```
        %b %Y %H:%M:%S
'%a, %d%Z'
```

strptime	`='%a %b %d %H:%M:%S` `strptime(str,[fmt%Y'])` Parses `str` according to format string `fmt` and returns the instant as a timetuple. The format string's syntax is as covered in `strftime` earlier.
time	`time()` Returns the current time instant, a `float` with seconds since the epoch. On some (mostly, older) platforms, the precision of this time is as low as one second. May return a lower value in a subsequent call if the system clock is adjusted backward between calls (e.g., due to leap seconds).
timezone	`timezone` The offset in seconds of the local time zone (without DST) from UTC (`>0` in the Americas; `<=0` in most of Europe, Asia, and Africa).
tzname	`tzname` A pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

The datetime Module

`datetime` provides classes for modeling date and time objects, which can be either *aware* of time zones or *naive* (the default). The class `tzinfo`, whose instances model a time zone, is abstract: the module `datetime` supplies no implementation (for all the gory details, see the online docs). See the module `pytz`, in "The pytz Module", for a good, simple implementation of `tzinfo`, which lets you easily create time zone-aware `datetime` objects. All types in `datetime` have immutable instances: attributes are read-only, and instances can be keys in a `dict` or items in a `set`.

The date Class

Instances of the `date` class represent a date (no time of day in particular within that date), are always naive, and assume the Gregorian calendar was always in effect. `date` instances have three read-only integer attributes: `year`, `month`, and `day`:

date `date(year,month,day)`

The `date` class supplies class methods usable as alternative constructors:

fromordinal	`date.fromordinal(ordinal)` Returns a `date` object corresponding to the proleptic Gregorian ordinal `ordinal`, where a value of `1` corresponds to the first day of year 1 CE.
fromtimestamp	`date.fromtimestamp(timestamp)` Returns a `date` object corresponding to the instant `timestamp` expressed in seconds since the epoch.
today	`date.today()` Returns a `date` object representing today’s date.

Instances of the `date` class support some arithmetic: the difference between `date` instances is a `timedelta` instance; you can add or subtract a `timedelta` to/from a `date` instance to make another `date` instance. You can compare any two instances of the `date` class (the later one is greater).

An instance `d` of the class `date` supplies the following methods:

ctime	`d.ctime()` Returns a string representing the date `d` in the same 24-character format as `time.ctime` (with the time of day set to 00:00:00, midnight).
isocalendar	`d.isocalendar()` Returns a tuple with three integers (ISO year, ISO week number, and ISO weekday). See the ISO 8601 standard for more details about the ISO (International Standards Organization) calendar.
isoformat	`d.isoformat()` Returns a string representing date `d` in the format `'YYYY-MM-DD'`; same as `str(d)`.
isoweekday	`d.isoweekday()` Returns the day of the week of date `d` as an integer, `1` for Monday through `7` for Sunday; like `d.weekday() + 1`.

replace	`d.replace(year=None,month=None,day=None)`
	
	Returns a new `date` object, like `d` except for those attributes explicitly specified as arguments, which get replaced. For example:
	
	`date(x,y,z).replace(month=m) == date(x,m,z)`

strftime `d.strftime()`

Returns a string representing date `d` as specified by string `fmt`, like:

```
time.strftime(fmt, d.timetuple())
```

timetuple `d.timetuple()`

Returns a time tuple corresponding to date `d` at time 00:00:00 (midnight).

toordinal `d.toordinal()`

Returns the proleptic Gregorian ordinal for date `d`. For example:

```
date(1,1,1).toordinal() == 1
```

weekday `d.weekday()`

Returns the day of the week of date `d` as an integer, 0 for Monday through 6 for Sunday; like `d.isoweekday() - 1`.

The time Class

Instances of the `time` class represent a time of day (of no particular date), may be naive or aware regarding time zones, and always ignore leap seconds. They have five attributes: four read-only integers (`hour`, `minute`, `second`, and `microsecond`) and an optional `tzinfo` (`None` for naive instances).

time `time(hour=0,minute=0,second=0,microsecond=0,tzinfo=None)`

Instances of the class `time` do not support arithmetic. You can compare two instances of `time` (the one that's later in the day is greater), but only if they are either both aware or both naive.

An instance `t` of the class `time` supplies the following methods:

isoformat `t.isoformat()`

Returns a string representing time `t` in the format `'HH:MM:SS'`; same as `str(t)`. If `t.microsecond!=0`, the resulting string is longer: `'HH:MM:SS.mmmmmm'`. If `t` is aware, six more characters, `'+HH:MM'`, are added at the end to represent the time zone's offset from UTC. In other words, this formatting operation follows the ISO 8601 standard.

replace `t.replace(hour=None,minute=None,second=None,microsecond=None[, tzinfo])`

Returns a new `time` object, like `t` except for those attributes explicitly specified as arguments, which get replaced. For example:

```
time(x,y,z).replace(minute=m) == time(x,m,z)
```

strftime `t.strftime()`

Returns a string representing time `t` as specified by the string `fmt`.

An instance `t` of the class `time` also supplies methods `dst`, `tzname`, and `utcoffset`, which accept no arguments and delegate to `t.tzinfo`, returning `None` when `t.tzinfo` is `None`.

The datetime Class

Instances of the `datetime` class represent an instant (a date, with a specific time of day within that date), may be naive or aware of time zones, and always ignore leap seconds. `datetime` extends `date` and adds `time`’s attributes; its instances have read-only integers `year`, `month`, `day`, `hour`, `minute`, `second`, and `microsecond`, and an optional `tzinfo` (`None` for naive instances).

Instances of `datetime` support some arithmetic: the difference between `datetime` instances (both aware, or both naive) is a `timedelta` instance, and you can add or subtract a `timedelta` instance to/from a `datetime` instance to construct another `datetime` instance. You can compare two instances of the `datetime` class (the later one is greater) as long as they’re both aware or both naive.

datetime `datetime(year,month,day,hour=0,minute=0,second=0,` `microsecond=0,tzinfo=None)`

The class `datetime` also supplies some class methods usable as alternative constructors.

combine `datetime.combine(date,time)`

Returns a `datetime` object with the date attributes taken from `date` and the time attributes (including `tzinfo`) taken from `time`. `datetime.combine(d,t)` is like:

```
datetime(d.year, d.month, d.day,
         t.hour, t.minute, t.second,
         t.microsecond, t.tzinfo)
```

fromordinal `datetime.fromordinal(ordinal)`

Returns a `datetime` object for the date given proleptic Gregorian ordinal `ordinal`, where a value of `1` means the first day of year `1` CE, at midnight.

fromtimestamp `datetime.fromtimestamp(timestamp,tz=None)`

Returns a `datetime` object corresponding to the instant `timestamp` expressed in seconds since the epoch, in local time. When `tz` is not `None`, returns an aware `datetime` object with the given `tzinfo` instance `tz`.

now `datetime.now(tz=None)`

Returns a `datetime` object for the current local date and time. When `tz` is not `None`, returns an aware `datetime` object with the given `tzinfo` instance `tz`.

strptime	`datetime.strptime(`*`str,fmt=`*`'`*`%a %b %d %H:%M:%S %Y %z`*`')` Returns a `datetime` representing `str` as specified by string `fmt`. In v3 only, when `%z` is specified, the resulting `datetime` object is time zone-aware.
today	`datetime.today()` Returns a naive `datetime` object representing the current local date and time, same as the `now` class method (but not accepting optional argument `tz`).
utcfromtimestamp	`datetime.utcfromtimestamp(timestamp)` Returns a naive `datetime` object corresponding to the instant `timestamp` expressed in seconds since the epoch, in UTC.
utcnow	`datetime.utcnow()` Returns a naive `datetime` object representing the current date and time, in UTC.

An instance `d` of `datetime` also supplies the following methods:

astimezone	`d.astimezone(tz)` Returns a new aware `datetime` object, like `d` (which must also be aware), except that the time zone is converted to the one in `tzinfo` object `tz`. Note that `d.astimezone(tz)` is quite different from `d.replace(tzinfo=tz)`: the latter does no time zone conversion, but rather just copies all of `d`'s attributes except for `d.tzinfo`.
ctime	`d.ctime()` Returns a string representing date and time `d` in the same 24-character format as `time.ctime`.
date	`d.date()` Returns a date object representing the same date as `d`.
isocalendar	`d.isocalendar()` Returns a tuple with three integers (ISO year, ISO week number, and ISO weekday) for `d`'s date.
isoformat	`d.isoformat(`*`sep`*`='T')` Returns a string representing `d` in the format `'YYYY-MM-DDxHH:MM:SS'`, where `x` is the value of argument `sep` (must be a string of length 1). If `d.microsecond!=0`, seven characters, `'.mmmmmm'`, are added after the `'SS'` part of the string. If `t` is aware, six more characters, `'+HH:MM'`, are added at the end to represent the time zone's offset from UTC. In other words, this formatting operation follows the ISO 8601 standard. `str(d)` is the same as `d.isoformat(' ')` .
isoweekday	`d.isoweekday()` Returns the day of the week of `d`'s date as an integer; 1 for Monday through 7 for Sunday.

replace	`d.replace(year=None,month=None,day=None,hour=None,minute=None, second=None,microsecond=None[,tzinfo])` Returns a new `datetime` object, like `d` except for those attributes specified as arguments, which get replaced. For example: `datetime(x,y,z).replace(month=m) == datetime(x,m,z)`
strftime	`d.strftime(`*`fmt`*`)` Returns a string representing `d` as specified by the format string `fmt`.
time	`d.time()` Returns a naive time object representing the same time of day as `d`.
timestamp	`d.timestamp()` Returns a float with the seconds since the epoch (v3 only). Naive instances are assumed to be in the local time zone.
timetz	`d.timetz()` Returns a `time` object representing the same time of day as `d`, with the same `tzinfo`.
timetuple	`d.timetuple()` Returns a timetuple corresponding to instant `d`.
toordinal	`d.toordinal()` Returns the proleptic Gregorian ordinal for `d`’s date. For example: `datetime(1,1,1).toordinal() == 1`
utctimetuple	`d.utctimetuple()` Returns a timetuple corresponding to instant `d`, normalized to UTC if `d` is aware.
weekday	`d.weekday()` Returns the day of the week of `d`’s date as an integer; `0` for Monday through `6` for Sunday. An instance `d` of the class `datetime` also supplies the methods `dst`, `tzname`, and `utcoffset`, which accept no arguments and delegate to `d.tzinfo`, returning `None` when `d.tzinfo` is `None`.

The timedelta Class

Instances of the `timedelta` class represent time intervals with three read-only integer attributes: `days`, `seconds`, and `microseconds`.

timedelta `timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)`

Converts all units with the obvious factors (a week is 7 days, an hour is 3,600 seconds, and so on) and normalizes everything to the three integer attributes, ensuring that `0<=seconds<3600*24` and `0<=microseconds<1000000`. For example:

```
                                                          timedelta(0,
print(repr(timedelta(minutes=0.5)))# prints: datetime.30)              print
                                                     timedelta(-1,
(repr(timedelta(minutes=-0.5)))# prints: datetime.86370)
```

Instances of `timedelta` support arithmetic (`+` and `-` between themselves and with instances of the classes `date` and `datetime`; `*` and `/` with integers) and comparisons between themselves. v3 also supports division between `timedelta` instances (floor division, true division, `divmod`, `%`). The instance method `total_seconds` returns the total seconds in a timedelta instance.

The pytz Module

The third-party `pytz` module offers the best, simplest ways to create `tzinfo` instances to make time zone–aware instances of the classes `time` and `datetime`. (`pytz` is based on the Olson library of time zones. `pytz`, like just about every third-party Python package, is available from PyPI: just `pip install pytz`.)

Dealing with time zones

The best way to program around the traps and pitfalls of time zones is to always use the UTC time zone internally, converting from other time zones on input, and to other time zones only for display purposes.

`pytz` supplies the attributes `common_timezones`, a list of over 400 strings that name the most common time zones you might want to use (mostly of the form `continent/city`, with some synonyms like `'UTC'` and `'US/Pacific'`) and `all_timezones`, a list of over 500 strings that also supply other synonyms for the time zones. For example, to specify the time zone of Lisbon, Portugal, by Olson library standards, the canonical way is `'Europe/Lisbon'`, and that is what you find in `common_timezones`; however, you may also use `'Portugal'`, which you find only in `all_timezones`. `pytz` also supplies the attributes `utc` and `UTC`, two names for the same object: a `tzinfo` instance representing Coordinated Universal Time (UTC).

`pytz` also supplies two functions:

country_timezones `country_timezones(code)`

Returns a list of time zone names corresponding to the country whose two-letter ISO code is `code`. For example, `pytz.country_timezones('US')` returns a list of 22 strings, from `'America/New_York'` to `'Pacific/Honolulu'`.

timezone `timezone(name)`

Returns an instance of `tzinfo` corresponding to the time zone named `name`.

For example, to print the Honolulu equivalent of midnight, December 31, 2005, in New York:

```
dt = datetime.datetime(2005,12,31,tzinfo=pytz.timezone('
America/New_York'))print(dt.astimezone(  pytz.timezone('
Pacific/Honolulu')))# prints: 2005-12-30 19:00:00-10:00
```

The dateutil Module

The third-party package `dateutil` (which you can install with `pip install python-dateutil`) offers modules to manipulate dates in many ways: time deltas, recurrence, time zones, Easter dates, and fuzzy parsing. (See the package's website for complete documentation of its rich functionality.) `dateutil`'s main modules are:

easter `easter.easter(year)`

Returns the `datetime.date` object for Easter of the given `year`. For example:

```
from dateutil import easter
print(easter.easter(2006))
```

prints `2006-04-16`.

parser `parser.parse(s)`

Returns the `datetime.datetime` object denoted by string `s`, with very permissive (AKA "fuzzy") parsing rules. For example:

```
from dateutil import parser
print(parser.parse('Saturday, January 28, 2006,
                         at 11:15pm'))
```

prints `2006-01-28 23:15:00`.

relativedelta `relativedelta.relativedelta(...)`

You can call `relativedelta` with two instances of `datetime.datetime`: the resulting `relativedelta` instance captures the relative difference between the two arguments. Alternatively, you can call `relativedelta` with a named argument representing absolute information (`year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`); relative information (`years`, `months`, `weeks`, `days`, `hours`, `minutes`, `seconds`, `microseconds`), which may have positive or negative values; or the special named argument `weekday`, which can be a number from 0 (Monday) to 6 (Sunday), or one of the module attributes `MO`, `TU`,..., `SU`, which can also be called with a numeric argument `n` to specify the `n`th weekday. In any case, the resulting `relativedelta` instance captures the information in the call. For example, after:

```
from dateutil import relativedelta
r = relativedelta.relativedelta(weekday=relativedelta.MO(1))
```

`r` means “next Monday.” You can add a `relativedelta` instance to a `datetime.datetime` instance to get a new `datetime.datetime` instance at the stated relative delta from the other:

```
print(datetime.datetime(2006,1,29)+r)
# prints: 2006-01-30
print(datetime.datetime(2006,1,30)+r)
# prints: 2006-01-30
print(datetime.datetime(2006,1,31)+r)
# prints: 2006-02-06
```

Note that “next Monday,” by `relativedelta`’s interpretation, is the very same date, if that day is already a Monday (so, a more detailed name might be “the first date, on or following the given date, which falls on a Monday”). `dateutil`’s site has detailed explanations of the rules defining the inevitably complicated behavior of `relativedelta` instances.

rrule ,
`rrule.rrule(freq...)`

Module `rrule` implements RFC2445 (also known as the `iCalendar` RFC), in all the glory of its 140+ pages. `freq` must be one of the constant attributes of module `rrule`: `YEARLY`, `MONTHLY`, `WEEKLY`, `DAILY`, `HOURLY`, `MINUTELY`, or `SECONDLY`. After mandatory argument `freq` may optionally come some of many possible named arguments, such as `interval=2`, to specify that the recurrence is only on alternate occurrences of the specified frequency (for example, `rrule.rrule(rrule.YEARLY)` repeats every year, while `rrule.rrule(rrule.YEARLY, interval=7)` repeats only every seven years, as a typical academic sabbatical year would).

An instance `r` of the type `rrule.rrule` supplies several methods:

after `r.after(d, inc=False)`

Returns the earliest `datetime.datetime` instance that’s an occurrence of recurrence rule `r` and happens after date `d` (when `inc` is true, an occurrence happening on the date `d` itself is also acceptable).

before `r.before(d, inc=False)`

Returns the latest `datetime.datetime` instance that's an occurrence of recurrence rule `r` and happens before date `d` (when `inc` is true, an occurrence happening on date `d` itself is also acceptable).

between `r.between(start, finish, inc=False)`

Returns all `datetime.datetime` instances that are occurrences of recurrence rule `r` and happen between the dates `start` and `finish` (when `inc` is true, occurrences on the dates `start` and `finish` themselves are also acceptable).

For example, to say "once a week throughout January 2018," the snippet:

```
start=datetime.datetime(2018,1,1)
r=rrule.rrule(rrule.WEEKLY, dtstart=start)
for d in r.between(start,datetime.datetime(2018,2,1),True):
                              '
    print(d.date(),end='  )
```

prints: **2018-01-01 2018-01-08 2018-01-15 2018-01-22 2018-01-29**

count `r.count()`

Returns the number of occurrences of recurrence rule `r` (may loop forever when `r` has an unbounded number of occurrences).

The sched Module

The `sched` module implements an event scheduler, letting you easily deal, along a single thread of execution or in multithreaded environments, with events that may be scheduled in either a "real" or a "simulated" time scale. `sched` supplies a `scheduler` class:

scheduler `class scheduler([`*`timefunc`*`], [`*`delayfunc`*`])`

The arguments `timefunc` and `delayfunc` are mandatory in v2, and optional in v3 (where they default to `time.monotonic` and `time.sleep`, respectively).

`timefunc` must be callable without arguments to get the current time instant (in any unit of measure); for example, you can pass `time.time` (or, in v3 only, `time.monotonic`). `delayfunc` is callable with one argument (a time duration, in the same units as `timefunc`) to delay the current thread for that time; for example, you can pass `time.sleep`. `scheduler` calls `delayfunc(0)` after each event to give other threads a chance; this is compatible with `time.sleep`. By taking functions as arguments, `scheduler` lets you use whatever "simulated time" or "pseudotime" fits your application's needs (a great example of the dependency injection design pattern for purposes not necessarily related to testing).

If monotonic time (time cannot go backward, even if the system clock is adjusted backward between calls, e.g., due to leap seconds) is important to your application, use v3 `time.monotonic` for your scheduler. A `scheduler` instance `s` supplies the following methods:

cancel `s.cancel(event_token)`

Removes an event from `s`'s queue. `event_token` must be the result of a previous call to `s.enter` or `s.enterabs`, and the event must not yet have happened; otherwise, `cancel` raises `RuntimeError`.

---

empty `s.empty()`

Returns `True` when `s`'s queue is currently empty; otherwise, `False`.

---

enterabs `s.enterabs(when,priority,func,args=(),kwargs={})`

`args,`
Schedules a future event (a callback to `func(kwargs ))` at time `when`. This is the v3 signature; in v2, sequence *`args`* is mandatory, and mapping `kwargs` is not allowed.

`when` is in the units used by the time functions of `s`. Should several events be scheduled for the same time, `s` executes them in increasing order of `priority`. `enterabs` returns an event token `t`, which you may pass to `s.cancel` to cancel this event.

---

enter `s.enter(delay,priority,func,`*`args`*`=(),`*`kwargs`*`={})`

Like `enterabs`, except that `delay` is a relative time (a positive difference forward from the current instant), while `enterabs`'s argument `when` is an absolute time (a future instant). In v3, `args` is optional and `kwargs` is added.

To schedule an event for *repeated* execution, use a little wrapper function; for example:

```
def enter_repeat(s, first_delay, period, priority, func, args):
    def repeating_wrapper():
        s.enter(period, priority, repeating_wrapper, ())
        func(*args)
    s.enter(first_delay, priority, repeating_wrapper, ())
```

---

run `s.run(`*`blocking`*`=True)`

Runs scheduled events. In v2, or if `blocking` is true (v3 only), `s.run` loops until `s.empty()`, using the `delayfunc` passed on `s`'s initialization to wait for each scheduled event. If `blocking` is false (v3 only), executes any soon-to-expire events, then returns the next event's deadline (if any). When a callback `func` raises an exception, `s` propagates it, but `s` keeps its own state, removing the event from the schedule. If a callback `func` runs longer than the time available before the next scheduled event, `s` falls behind but keeps executing scheduled events in order, never dropping any. Call `s .cancel` to drop an event explicitly if that event is no longer of interest.

The calendar Module

The `calendar` module supplies calendar-related functions, including functions to print a text calendar for a given month or year. By default, `calendar` takes Monday as the first day of the week and Sunday as the last one. To change this, call `calendar.setfirstweekday`. `calendar` handles years in module `time`'s range, typically (at least) 1970 to 2038.

`python -m calendar` offers a useful command-line interface to the module's functionality: run `python -m calendar -h` to get a brief help message.

The `calendar` module supplies the following functions:

calendar	`calendar(year,w=2,l=1,c=6)` Returns a multiline string with a calendar for year `year` formatted into three columns separated by `c` spaces. `w` is the width in characters of each date; each line has length `21*w+18+2*c`. `l` is the number of lines for each week.
firstweekday	`firstweekday()` Returns the current setting for the weekday that starts each week. By default, when `calendar` is first imported, this is `0`, meaning Monday.
isleap	`isleap(year)` Returns `True` if `year` is a leap year; otherwise, `False`.
leapdays	`leapdays(y1,y2)` Returns the total number of leap days in the years within `range(y1,y2)` (remember, this means that `y2` is excluded).
month	`month(year,month,w=2,l=1)` Returns a multiline string with a calendar for month `month` of year `year`, one line per week plus two header lines. `w` is the width in characters of each date; each line has length `7*w+6`. `l` is the number of lines for each week.
monthcalendar	`monthcalendar(year,month)` Returns a list of lists of `int`s. Each sublist denotes a week. Days outside month `month` of year `year` are set to `0`; days within the month are set to their day-of-month, `1` and up.
monthrange	`monthrange(year,month)` Returns two integers. The first one is the code of the weekday for the first day of the month `month` in year `year`; the second one is the number of days in the month. Weekday codes are `0` (Monday) to `6` (Sunday); month numbers are `1` to `12`.
prcal	`prcal(year,w=2,l=1,c=6)` Like `print(calendar.calendar(year,w,l,c))`.
prmonth	`prmonth(year,month,w=2,l=1)` Like `print(calendar.month(year,month,w,l))`.
setfirstweekday	`setfirstweekday(weekday)` Sets the first day of each week to weekday code `weekday`. Weekday codes are `0` (Monday) to `6` (Sunday). `calendar` supplies the attributes `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY`, whose values are the integers `0` to `6`. Use these attributes when you mean weekdays (e.g., `calendar.FRIDAY` instead of `4`) to make your code clearer and more readable.

timegm	`timegm(tupletime)` The inverse of `time.gmtime`: accepts a time instant in timetuple form and returns that instant as a `float` num of seconds since the epoch.
weekday	`weekday(year,month,day)` Returns the weekday code for the given date. Weekday codes are `0` (Monday) to `6` (Sunday); month numbers are `1` (Jan) to `12` (Dec).