CHAPTER

EIGHT

DATA TYPES

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, synchronized queues, and sets.

Python also provides some built-in data types, in particular, *dict*, *list*, *set* and *frozenset*, and *tuple*. The *str* class is used to hold Unicode strings, and the *bytes* class is used to hold binary data.

The following modules are documented in this chapter:

8.1 datetime — Basic date and time types

Source code: Lib/datetime.py

The *datetime* module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation. For related functionality, see also the *time* and *calendar* modules.

There are two kinds of date and time objects: "naive" and "aware".

An aware object has sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, to locate itself relative to other aware objects. An aware object is used to represent a specific moment in time that is not open to interpretation[1].

A naive object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, *datetime* and *time* objects have an optional time zone information attribute, `tzinfo`, that can be set to an instance of a subclass of the abstract *tzinfo* class. These *tzinfo* objects capture information about the offset from UTC time, the time zone name, and whether Daylight Saving Time is in effect. Note that only one concrete *tzinfo* class, the *timezone* class, is supplied by the *datetime* module. The *timezone* class can represent simple timezones with fixed offset from UTC, such as UTC itself or North American EST and EDT timezones. Supporting timezones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

The *datetime* module exports the following constants:

`datetime.MINYEAR`
: The smallest year number allowed in a *date* or *datetime* object. *MINYEAR* is `1`.

[1] If, that is, we ignore the effects of Relativity

datetime.MAXYEAR
The largest year number allowed in a *date* or *datetime* object. *MAXYEAR* is 9999.

See also:

Module *calendar* General calendar related functions.

Module *time* Time access and conversions.

8.1.1 Available Types

class datetime.date
An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: *year*, *month*, and *day*.

class datetime.time
An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of "leap seconds" here). Attributes: *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

class datetime.datetime
A combination of a date and a time. Attributes: *year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

class datetime.timedelta
A duration expressing the difference between two *date*, *time*, or *datetime* instances to microsecond resolution.

class datetime.tzinfo
An abstract base class for time zone information objects. These are used by the *datetime* and *time* classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

class datetime.timezone
A class that implements the *tzinfo* abstract base class as a fixed offset from the UTC.

New in version 3.2.

Objects of these types are immutable.

Objects of the *date* type are always naive.

An object of type *time* or *datetime* may be naive or aware. A *datetime* object *d* is aware if `d.tzinfo` is not `None` and `d.tzinfo.utcoffset(d)` does not return `None`. If `d.tzinfo` is `None`, or if `d.tzinfo` is not `None` but `d.tzinfo.utcoffset(d)` returns `None`, *d* is naive. A *time* object *t* is aware if `t.tzinfo` is not `None` and `t.tzinfo.utcoffset(None)` does not return `None`. Otherwise, *t* is naive.

The distinction between naive and aware doesn't apply to *timedelta* objects.

Subclass relationships:

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

8.1.2 timedelta Objects

A *timedelta* object represents a duration, the difference between two dates or times.

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

All arguments are optional and default to `0`. Arguments may be integers or floats, and may be positive or negative.

Only *days*, *seconds* and *microseconds* are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (the number of seconds in one day)
- `-999999999 <= days <= 999999999`

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond using round-half-to-even tiebreaker. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, *OverflowError* is raised.

Note that normalization of negative values may be surprising at first. For example,

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Class attributes are:

timedelta.min
The most negative *timedelta* object, `timedelta(-999999999)`.

timedelta.max
The most positive *timedelta* object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

timedelta.resolution
The smallest possible difference between non-equal *timedelta* objects, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max` > `-timedelta.min`. `-timedelta.max` is not representable as a *timedelta* object.

Instance attributes (read-only):

Attribute	Value
`days`	Between -999999999 and 999999999 inclusive
`seconds`	Between 0 and 86399 inclusive
`microseconds`	Between 0 and 999999 inclusive

Supported operations:

Operation	Result
`t1 = t2 + t3`	Sum of *t2* and *t3*. Afterwards *t1-t2* == *t3* and *t1-t3* == *t2* are true. (1)
`t1 = t2 - t3`	Difference of *t2* and *t3*. Afterwards *t1* == *t2* - *t3* and *t2* == *t1* + *t3* are true. (1)(6)
`t1 = t2 * i or t1 = i * t2`	Delta multiplied by an integer. Afterwards *t1* // i == *t2* is true, provided `i != 0`.
	In general, *t1* * i == *t1* * (i-1) + *t1* is true. (1)
`t1 = t2 * f or t1 = f * t2`	Delta multiplied by a float. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
`f = t2 / t3`	Division (3) of *t2* by *t3*. Returns a *float* object.
`t1 = t2 / f or t1 = t2 / i`	Delta divided by a float or an int. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
`t1 = t2 // i` or `t1 = t2 // t3`	The floor is computed and the remainder (if any) is thrown away. In the second case, an integer is returned. (3)
`t1 = t2 % t3`	The remainder is computed as a *timedelta* object. (3)
`q, r = divmod(t1, t2)`	Computes the quotient and the remainder: `q = t1 // t2` (3) and `r = t1 % t2`. q is an integer and r is a *timedelta* object.
`+t1`	Returns a *timedelta* object with the same value. (2)
`-t1`	equivalent to *timedelta*(*-t1.days*, *-t1.seconds*, *-t1.microseconds*), and to *t1** -1. (1)(4)
`abs(t)`	equivalent to +*t* when `t.days >= 0`, and to -*t* when `t.days < 0`. (2)
`str(t)`	Returns a string in the form `[D day[s], ][H]H:MM:SS[.UUUUUU]`, where D is negative for negative `t`. (5)
`repr(t)`	Returns a string representation of the *timedelta* object as a constructor call with canonical attribute values.

Notes:

(1) This is exact, but may overflow.

(2) This is exact, and cannot overflow.

(3) Division by 0 raises *ZeroDivisionError*.

(4) *-timedelta.max* is not representable as a *timedelta* object.

(5) String representations of *timedelta* objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

(6) The expression `t2 - t3` will always be equal to the expression `t2 + (-t3)` except when t3 is equal to `timedelta.max`; in that case the former will produce a result while the latter will overflow.

In addition to the operations listed above *timedelta* objects support certain additions and subtractions with *date* and *datetime* objects (see below).

Changed in version 3.2: Floor division and true division of a *timedelta* object by another *timedelta* object are now supported, as are remainder operations and the *divmod()* function. True division and multiplication of a *timedelta* object by a *float* object are now supported.

Comparisons of *timedelta* objects are supported with the *timedelta* object representing the smaller duration considered to be the smaller timedelta. In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a *timedelta* object is compared to an object of a different type, *TypeError* is raised unless the comparison is `==` or `!=`. The latter cases return *False* or *True*, respectively.

timedelta objects are *hashable* (usable as dictionary keys), support efficient pickling, and in Boolean contexts, a *timedelta* object is considered to be true if and only if it isn't equal to `timedelta(0)`.

Instance methods:

timedelta.total_seconds()
Return the total number of seconds contained in the duration. Equivalent to `td / timedelta(seconds=1)`.

Note that for very large time intervals (greater than 270 years on most platforms) this method will lose microsecond accuracy.

New in version 3.2.

Example usage:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)  # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(days=3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(days=3285), 9)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```

8.1.3 `date` Objects

A *date* object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book Calendrical Calculations, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

class datetime.date(*year*, *month*, *day*)
All arguments are required. Arguments may be integers, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= number of days in the given month and year`

If an argument outside those ranges is given, *ValueError* is raised.

Other constructors, all class methods:

classmethod date.today()
Return the current local date. This is equivalent to `date.fromtimestamp(time.time())`.

classmethod `date.fromtimestamp`(*timestamp*)
Return the local date corresponding to the POSIX timestamp, such as is returned by *time.time()*. This may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `localtime()` function, and *OSError* on `localtime()` failure. It's common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*.

Changed in version 3.3: Raise *OverflowError* instead of *ValueError* if the timestamp is out of the range of values supported by the platform C `localtime()` function. Raise *OSError* instead of *ValueError* on `localtime()` failure.

classmethod `date.fromordinal`(*ordinal*)
Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. *ValueError* is raised unless `1 <= ordinal <= date.max.toordinal()`. For any date *d*, `date.fromordinal(d.toordinal()) == d`.

classmethod `date.fromisoformat`(*date_string*)
Return a *date* corresponding to a *date_string* in the format emitted by *date.isoformat()*. Specifically, this function supports strings in the format(s) `YYYY-MM-DD`.

> **Caution:** This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of *date.isoformat()*.

New in version 3.7.

Class attributes:

date.min
The earliest representable date, `date(MINYEAR, 1, 1)`.

date.max
The latest representable date, `date(MAXYEAR, 12, 31)`.

date.resolution
The smallest possible difference between non-equal date objects, `timedelta(days=1)`.

Instance attributes (read-only):

date.year
Between *MINYEAR* and *MAXYEAR* inclusive.

date.month
Between 1 and 12 inclusive.

date.day
Between 1 and the number of days in the given month of the given year.

Supported operations:

Operation	Result
`date2 = date1 + timedelta`	*date2* is `timedelta.days` days removed from *date1*. (1)
`date2 = date1 - timedelta`	Computes *date2* such that `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 < date2`	*date1* is considered less than *date2* when *date1* precedes *date2* in time. (4)

Notes:

(1) *date2* is moved forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. Afterward `date2 - date1 == timedelta.days`. `timedelta.seconds` and `timedelta.microseconds` are ignored. *OverflowError* is raised if `date2.year` would be smaller than *MINYEAR* or larger than *MAXYEAR*.

(2) `timedelta.seconds` and `timedelta.microseconds` are ignored.

(3) This is exact, and cannot overflow. timedelta.seconds and timedelta.microseconds are 0, and date2 + timedelta == date1 after.

(4) In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`. Date comparison raises *TypeError* if the other comparand isn't also a *date* object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a *date* object is compared to an object of a different type, *TypeError* is raised unless the comparison is `==` or `!=`. The latter cases return *False* or *True*, respectively.

Dates can be used as dictionary keys. In Boolean contexts, all *date* objects are considered to be true.

Instance methods:

date.replace(*year=self.year*, *month=self.month*, *day=self.day*)
Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified. For example, if `d == date(2002, 12, 31)`, then `d.replace(day=26) == date(2002, 12, 26)`.

date.timetuple()
Return a *time.struct_time* such as returned by *time.localtime()*. The hours, minutes and seconds are 0, and the DST flag is -1. `d.timetuple()` is equivalent to `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))`, where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with `1` for January 1st.

date.toordinal()
Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any *date* object *d*, `date.fromordinal(d.toordinal()) == d`.

date.weekday()
Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, `date(2002, 12, 4).weekday() == 2`, a Wednesday. See also *isoweekday()*.

date.isoweekday()
Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, `date(2002, 12, 4).isoweekday() == 3`, a Wednesday. See also *weekday()*, *isocalendar()*.

date.isocalendar()
Return a 3-tuple, (ISO year, ISO week number, ISO weekday).

The ISO calendar is a widely used variant of the Gregorian calendar. See https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm for a good explanation.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004, so that `date(2003, 12, 29).isocalendar() == (2004, 1, 1)` and `date(2004, 1, 4).isocalendar() == (2004, 1, 7)`.

date.isoformat()
Return a string representing the date in ISO 8601 format, 'YYYY-MM-DD'. For example, `date(2002, 12, 4).isoformat() == '2002-12-04'`.

`date.__str__()`
: For a date *d*, `str(d)` is equivalent to `d.isoformat()`.

`date.ctime()`
: Return a string representing the date, for example `date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which *time.ctime()* invokes, but which *date.ctime()* does not invoke) conforms to the C standard.

`date.strftime(`*format*`)`
: Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. For a complete list of formatting directives, see *strftime() and strptime() Behavior*.

`date.__format__(`*format*`)`
: Same as *date.strftime()*. This makes it possible to specify a format string for a *date* object in formatted string literals and when using *str.format()*. For a complete list of formatting directives, see *strftime() and strptime() Behavior*.

Example of counting days to an event:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

Example of working with *date*:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
```

(continues on next page)

(continued from previous page)

```
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'
```

8.1.4 `datetime` Objects

A *datetime* object is a single object containing all the information from a *date* object and a *time* object. Like a *date* object, *datetime* assumes the current Gregorian calendar extended in both directions; like a time object, *datetime* assumes there are exactly 3600*24 seconds in every day.

Constructor:

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

The year, month and day arguments are required. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments may be integers, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- `1 <= day <= number of days in the given month and year`,
- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

If an argument outside those ranges is given, *ValueError* is raised.

New in version 3.6: Added the `fold` argument.

Other constructors, all class methods:

classmethod datetime.**today**()

Return the current local datetime, with *tzinfo* `None`. This is equivalent to `datetime.fromtimestamp(time.time())`. See also *now()*, *fromtimestamp()*.

classmethod datetime.**now**(*tz=None*)

Return the current local date and time. If optional argument *tz* is `None` or not specified, this is like *today()*, but, if possible, supplies more precision than can be gotten from going through a *time.time()* timestamp (for example, this may be possible on platforms supplying the C `gettimeofday()` function).

If *tz* is not `None`, it must be an instance of a *tzinfo* subclass, and the current date and time are converted to *tz*'s time zone. In this case the result is equivalent to `tz.fromutc(datetime.utcnow().replace(tzinfo=tz))`. See also *today()*, *utcnow()*.

classmethod `datetime.utcnow()`
Return the current UTC date and time, with *tzinfo* `None`. This is like *now()*, but returns the current UTC date and time, as a naive *datetime* object. An aware current UTC datetime can be obtained by calling `datetime.now(timezone.utc)`. See also *now()*.

classmethod `datetime.fromtimestamp`(*timestamp*, *tz=None*)
Return the local date and time corresponding to the POSIX timestamp, such as is returned by *time.time()*. If optional argument *tz* is `None` or not specified, the timestamp is converted to the platform's local date and time, and the returned *datetime* object is naive.

If *tz* is not `None`, it must be an instance of a *tzinfo* subclass, and the timestamp is converted to *tz*'s time zone. In this case the result is equivalent to `tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz))`.

fromtimestamp() may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `localtime()` or `gmtime()` functions, and *OSError* on `localtime()` or `gmtime()` failure. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*, and then it's possible to have two timestamps differing by a second that yield identical *datetime* objects. See also *utcfromtimestamp()*.

Changed in version 3.3: Raise *OverflowError* instead of *ValueError* if the timestamp is out of the range of values supported by the platform C `localtime()` or `gmtime()` functions. Raise *OSError* instead of *ValueError* on `localtime()` or `gmtime()` failure.

Changed in version 3.6: *fromtimestamp()* may return instances with *fold* set to 1.

classmethod `datetime.utcfromtimestamp`(*timestamp*)
Return the UTC *datetime* corresponding to the POSIX timestamp, with *tzinfo* `None`. This may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `gmtime()` function, and *OSError* on `gmtime()` failure. It's common for this to be restricted to years in 1970 through 2038.

To get an aware *datetime* object, call *fromtimestamp()*:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

On the POSIX compliant platforms, it is equivalent to the following expression:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

except the latter formula always supports the full years range: between *MINYEAR* and *MAXYEAR* inclusive.

Changed in version 3.3: Raise *OverflowError* instead of *ValueError* if the timestamp is out of the range of values supported by the platform C `gmtime()` function. Raise *OSError* instead of *ValueError* on `gmtime()` failure.

classmethod `datetime.fromordinal`(*ordinal*)
Return the *datetime* corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. *ValueError* is raised unless `1 <= ordinal <= datetime.max.toordinal()`. The hour, minute, second and microsecond of the result are all 0, and *tzinfo* is `None`.

classmethod `datetime.combine`(*date*, *time*, *tzinfo=self.tzinfo*)
Return a new *datetime* object whose date components are equal to the given *date* object's, and whose time components are equal to the given *time* object's. If the *tzinfo* argument is provided, its value is used to set the *tzinfo* attribute of the result, otherwise the *tzinfo* attribute of the *time* argument is used.

For any *datetime* object *d*, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`. If date is a *datetime* object, its time components and *tzinfo* attributes are ignored.

Changed in version 3.6: Added the *tzinfo* argument.

classmethod `datetime.fromisoformat`(*date_string*)

Return a *datetime* corresponding to a *date_string* in one of the formats emitted by *date.isoformat()* and *datetime.isoformat()*. Specifically, this function supports strings in the format(s) `YYYY-MM-DD[*HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]]`, where `*` can match any single character.

> **Caution:** This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of *datetime.isoformat()*.

New in version 3.7.

classmethod `datetime.strptime`(*date_string*, *format*)

Return a *datetime* corresponding to *date_string*, parsed according to *format*. This is equivalent to `datetime(*(time.strptime(date_string, format)[0:6]))`. *ValueError* is raised if the date_string and format can't be parsed by *time.strptime()* or if it returns a value which isn't a time tuple. For a complete list of formatting directives, see *strftime() and strptime() Behavior*.

Class attributes:

datetime.min

The earliest representable *datetime*, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

datetime.max

The latest representable *datetime*, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

datetime.resolution

The smallest possible difference between non-equal *datetime* objects, `timedelta(microseconds=1)`.

Instance attributes (read-only):

datetime.year

Between *MINYEAR* and *MAXYEAR* inclusive.

datetime.month

Between 1 and 12 inclusive.

datetime.day

Between 1 and the number of days in the given month of the given year.

datetime.hour

In `range(24)`.

datetime.minute

In `range(60)`.

datetime.second

In `range(60)`.

datetime.microsecond

In `range(1000000)`.

datetime.tzinfo

The object passed as the *tzinfo* argument to the *datetime* constructor, or `None` if none was passed.

datetime.fold

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current

zone is decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

New in version 3.6.

Supported operations:

Operation	Result
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 < datetime2`	Compares *datetime* to *datetime*. (4)

(1) datetime2 is a duration of timedelta removed from datetime1, moving forward in time if `timedelta.days` > 0, or backward if `timedelta.days` < 0. The result has the same *tzinfo* attribute as the input datetime, and datetime2 - datetime1 == timedelta after. *OverflowError* is raised if datetime2.year would be smaller than *MINYEAR* or larger than *MAXYEAR*. Note that no time zone adjustments are done even if the input is an aware object.

(2) Computes the datetime2 such that datetime2 + timedelta == datetime1. As for addition, the result has the same *tzinfo* attribute as the input datetime, and no time zone adjustments are done even if the input is aware.

(3) Subtraction of a *datetime* from a *datetime* is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, *TypeError* is raised.

If both are naive, or both are aware and have the same *tzinfo* attribute, the *tzinfo* attributes are ignored, and the result is a *timedelta* object t such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

If both are aware and have different *tzinfo* attributes, `a-b` acts as if a and b were first converted to naive UTC datetimes first. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

(4) *datetime1* is considered less than *datetime2* when *datetime1* precedes *datetime2* in time.

If one comparand is naive and the other is aware, *TypeError* is raised if an order comparison is attempted. For equality comparisons, naive instances are never equal to aware instances.

If both comparands are aware, and have the same *tzinfo* attribute, the common *tzinfo* attribute is ignored and the base datetimes are compared. If both comparands are aware and have different *tzinfo* attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

Changed in version 3.3: Equality comparisons between naive and aware *datetime* instances don't raise *TypeError*.

Note: In order to stop comparison from falling back to the default scheme of comparing object addresses, datetime comparison normally raises *TypeError* if the other comparand isn't also a *datetime* object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a *datetime* object is compared to an object of a different type, *TypeError* is raised unless the comparison is `==` or `!=`. The latter cases return *False* or *True*, respectively.

---

datetime objects can be used as dictionary keys. In Boolean contexts, all *datetime* objects are considered to be true.

Instance methods:

`datetime.date()`
Return *date* object with same year, month and day.

`datetime.time()`
Return *time* object with same hour, minute, second, microsecond and fold. *tzinfo* is `None`. See also method *timetz()*.

Changed in version 3.6: The fold value is copied to the returned *time* object.

`datetime.timetz()`
Return *time* object with same hour, minute, second, microsecond, fold, and tzinfo attributes. See also method *time()*.

Changed in version 3.6: The fold value is copied to the returned *time* object.

`datetime.replace(`*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ** fold=0*`)`
Return a datetime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive datetime from an aware datetime with no conversion of date and time data.

New in version 3.6: Added the `fold` argument.

`datetime.astimezone(`*tz=None*`)`
Return a *datetime* object with new *tzinfo* attribute *tz*, adjusting the date and time data so the result is the same UTC time as *self*, but in *tz*'s local time.

If provided, *tz* must be an instance of a *tzinfo* subclass, and its *utcoffset()* and *dst()* methods must not return `None`. If *self* is naive, it is presumed to represent time in the system timezone.

If called without arguments (or with `tz=None`) the system local timezone is assumed for the target timezone. The `.tzinfo` attribute of the converted datetime instance will be set to an instance of *timezone* with the zone name and offset obtained from the OS.

If `self.tzinfo` is *tz*, `self.astimezone(tz)` is equal to *self*: no adjustment of date or time data is performed. Else the result is local time in the timezone *tz*, representing the same UTC time as *self*: after `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` will have the same date and time data as `dt - dt.utcoffset()`.

If you merely want to attach a time zone object *tz* to a datetime *dt* without adjustment of date and time data, use `dt.replace(tzinfo=tz)`. If you merely want to remove the time zone object from an aware datetime *dt* without conversion of date and time data, use `dt.replace(tzinfo=None)`.

Note that the default *tzinfo.fromutc()* method can be overridden in a *tzinfo* subclass to affect the result returned by *astimezone()*. Ignoring error cases, *astimezone()* acts like:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

Changed in version 3.3: *tz* now can be omitted.

Changed in version 3.6: The *astimezone()* method can now be called on naive instances that are presumed to represent system local time.

`datetime.utcoffset()`
If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.utcoffset(self)`, and raises an exception if the latter doesn't return `None` or a *timedelta* object with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

datetime.dst()
If *tzinfo* is None, returns None, else returns self.tzinfo.dst(self), and raises an exception if the latter doesn't return None or a *timedelta* object with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

datetime.tzname()
If *tzinfo* is None, returns None, else returns self.tzinfo.tzname(self), raises an exception if the latter doesn't return None or a string object,

datetime.timetuple()
Return a *time.struct_time* such as returned by *time.localtime()*. d.timetuple() is equivalent to time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst)), where yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 is the day number within the current year starting with 1 for January 1st. The tm_isdst flag of the result is set according to the *dst()* method: *tzinfo* is None or *dst()* returns None, tm_isdst is set to -1; else if *dst()* returns a non-zero value, tm_isdst is set to 1; else tm_isdst is set to 0.

datetime.utctimetuple()
If *datetime* instance *d* is naive, this is the same as d.timetuple() except that tm_isdst is forced to 0 regardless of what d.dst() returns. DST is never in effect for a UTC time.

If *d* is aware, *d* is normalized to UTC time, by subtracting d.utcoffset(), and a *time.struct_time* for the normalized time is returned. tm_isdst is forced to 0. Note that an *OverflowError* may be raised if *d*.year was MINYEAR or MAXYEAR and UTC adjustment spills over a year boundary.

datetime.toordinal()
Return the proleptic Gregorian ordinal of the date. The same as self.date().toordinal().

datetime.timestamp()
Return POSIX timestamp corresponding to the *datetime* instance. The return value is a *float* similar to that returned by *time.time()*.

Naive *datetime* instances are assumed to represent local time and this method relies on the platform C mktime() function to perform the conversion. Since *datetime* supports wider range of values than mktime() on many platforms, this method may raise *OverflowError* for times far in the past or far in the future.

For aware *datetime* instances, the return value is computed as:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

New in version 3.3.

Changed in version 3.6: The *timestamp()* method uses the *fold* attribute to disambiguate the times during a repeated interval.

Note: There is no method to obtain the POSIX timestamp directly from a naive *datetime* instance representing UTC time. If your application uses this convention and your system timezone is not set to UTC, you can obtain the POSIX timestamp by supplying tzinfo=timezone.utc:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

or by calculating the timestamp directly:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

`datetime.weekday()`
Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as `self.date().weekday()`. See also *isoweekday()*.

`datetime.isoweekday()`
Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as `self.date().isoweekday()`. See also *weekday()*, *isocalendar()*.

`datetime.isocalendar()`
Return a 3-tuple, (ISO year, ISO week number, ISO weekday). The same as `self.date().isocalendar()`.

`datetime.isoformat(`*sep='T', timespec='auto'*`)`
Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.ffffff or, if *microsecond* is 0, YYYY-MM-DDTHH:MM:SS

If *utcoffset()* does not return `None`, a string is appended, giving the UTC offset: YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]] or, if *microsecond* is 0 YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]].

The optional argument *sep* (default `'T'`) is a one-character separator, placed between the date and time portions of the result. For example,

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if *microsecond* is 0, same as `'microseconds'` otherwise.
- `'hours'`: Include the *hour* in the two-digit HH format.
- `'minutes'`: Include *hour* and *minute* in HH:MM format.
- `'seconds'`: Include *hour*, *minute*, and *second* in HH:MM:SS format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. HH:MM:SS.sss format.
- `'microseconds'`: Include full time in HH:MM:SS.ffffff format.

Note: Excluded time components are truncated, not rounded.

ValueError will be raised on an invalid *timespec* argument.

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')    # doctest: +SKIP
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

New in version 3.6: Added the *timespec* argument.

`datetime.__str__()`
For a *datetime* instance *d*, `str(d)` is equivalent to `d.isoformat(' ')`.

`datetime.ctime()`
Return a string representing the date and time, for example `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which *time.ctime()* invokes, but which *datetime.ctime()* does not invoke) conforms to the C standard.

`datetime.strftime(`*format*`)`
Return a string representing the date and time, controlled by an explicit format string. For a complete list of formatting directives, see *strftime() and strptime() Behavior*.

`datetime.__format__(`*format*`)`
Same as *datetime.strftime()*. This makes it possible to specify a format string for a *datetime* object in formatted string literals and when using *str.format()*. For a complete list of formatting directives, see *strftime() and strptime() Behavior*.

Examples of working with datetime objects:

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday
>>> # Formatting datetime
```

(continues on next page)

(continued from previous page)

```
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day",
↪"month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

Using datetime with tzinfo:

```
>>> from datetime import timedelta, datetime, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         # DST starts last Sunday in March
...         d = datetime(dt.year, 4, 1)   # ends last Sunday in October
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...          return "GMT +1"
...
>>> class GMT2(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=2) + self.dst(dt)
...     def dst(self, dt):
...         d = datetime(dt.year, 4, 1)
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...         return "GMT +2"
...
>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(seconds=3600)
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(seconds=3600)
>>> dt2.utcoffset()
datetime.timedelta(seconds=7200)
```

(continues on next page)

(continued from previous page)

```
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())
>>> dt3     # doctest: +ELLIPSIS
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>)
>>> dt2     # doctest: +ELLIPSIS
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x...>)
>>> dt2.utctimetuple() == dt3.utctimetuple()
True
```

8.1.5 `time` Objects

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a *tzinfo* object.

class `datetime.time`(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

All arguments are optional. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments may be integers, in the following ranges:

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

If an argument outside those ranges is given, *ValueError* is raised. All default to `0` except *tzinfo*, which defaults to *None*.

Class attributes:

time.min

The earliest representable *time*, `time(0, 0, 0, 0)`.

time.max

The latest representable *time*, `time(23, 59, 59, 999999)`.

time.resolution

The smallest possible difference between non-equal *time* objects, `timedelta(microseconds=1)`, although note that arithmetic on *time* objects is not supported.

Instance attributes (read-only):

time.hour

In `range(24)`.

time.minute

In `range(60)`.

time.second

In `range(60)`.

time.microsecond

In `range(1000000)`.

time.tzinfo

The object passed as the tzinfo argument to the *time* constructor, or `None` if none was passed.

`time.fold`
: In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

  New in version 3.6.

Supported operations:

- comparison of *time* to *time*, where *a* is considered less than *b* when *a* precedes *b* in time. If one comparand is naive and the other is aware, *TypeError* is raised if an order comparison is attempted. For equality comparisons, naive instances are never equal to aware instances.

  If both comparands are aware, and have the same *tzinfo* attribute, the common *tzinfo* attribute is ignored and the base times are compared. If both comparands are aware and have different *tzinfo* attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a *time* object is compared to an object of a different type, *TypeError* is raised unless the comparison is `==` or `!=`. The latter cases return *False* or *True*, respectively.

  Changed in version 3.3: Equality comparisons between naive and aware *time* instances don't raise *TypeError*.

- hash, use as dict key
- efficient pickling

In boolean contexts, a *time* object is always considered to be true.

Changed in version 3.5: Before Python 3.5, a *time* object was considered to be false if it represented midnight in UTC. This behavior was considered obscure and error-prone and has been removed in Python 3.5. See bpo-13936 for full details.

Other constructor:

classmethod `time.fromisoformat`(*time_string*)
: Return a *time* corresponding to a *time_string* in one of the formats emitted by *time.isoformat()*. Specifically, this function supports strings in the format(s) `HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]`.

  Caution: This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of *time.isoformat()*.

  New in version 3.7.

Instance methods:

`time.replace`(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ** fold=0*)
: Return a *time* with the same value, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive *time* from an aware *time*, without conversion of the time data.

  New in version 3.6: Added the `fold` argument.

`time.isoformat`(*timespec='auto'*)
: Return a string representing the time in ISO 8601 format, HH:MM:SS.ffffff or, if *microsecond* is 0, HH:MM:SS If *utcoffset()* does not return `None`, a string is appended, giving the UTC offset: HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]] or, if self.microsecond is 0, HH:MM:SS+HH:MM[:SS[.ffffff]].

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if *microsecond* is 0, same as `'microseconds'` otherwise.
- `'hours'`: Include the *hour* in the two-digit HH format.
- `'minutes'`: Include *hour* and *minute* in HH:MM format.
- `'seconds'`: Include *hour*, *minute*, and *second* in HH:MM:SS format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. HH:MM:SS.sss format.
- `'microseconds'`: Include full time in HH:MM:SS.ffffff format.

Note: Excluded time components are truncated, not rounded.

ValueError will be raised on an invalid *timespec* argument.

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec=
↪'minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

New in version 3.6: Added the *timespec* argument.

`time.__str__()`
For a time *t*, `str(t)` is equivalent to `t.isoformat()`.

`time.strftime(`*format*`)`
Return a string representing the time, controlled by an explicit format string. For a complete list of formatting directives, see *strftime() and strptime() Behavior*.

`time.__format__(`*format*`)`
Same as *time.strftime()*. This makes it possible to specify a format string for a *time* object in formatted string literals and when using *str.format()*. For a complete list of formatting directives, see *strftime() and strptime() Behavior*.

`time.utcoffset()`
If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.utcoffset(None)`, and raises an exception if the latter doesn't return `None` or a *timedelta* object with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

`time.dst()`
If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.dst(None)`, and raises an exception if the latter doesn't return `None`, or a *timedelta* object with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

`time.tzname()`
If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.tzname(None)`, or raises an exception if the latter doesn't return `None` or a string object.

Example:

```
>>> from datetime import time, tzinfo, timedelta
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t                                # doctest: +ELLIPSIS
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.6 `tzinfo` Objects

class datetime.tzinfo

This is an abstract base class, meaning that this class should not be instantiated directly. You need to derive a concrete subclass, and (at least) supply implementations of the standard *tzinfo* methods needed by the *datetime* methods you use. The *datetime* module supplies a simple concrete subclass of *tzinfo*, *timezone*, which can represent timezones with fixed offset from UTC such as UTC itself or North American EST and EDT.

An instance of (a concrete subclass of) *tzinfo* can be passed to the constructors for *datetime* and *time* objects. The latter objects view their attributes as being in local time, and the *tzinfo* object supports methods revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

Special requirement for pickling: A *tzinfo* subclass must have an `__init__()` method that can be called with no arguments, else it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of *tzinfo* may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware *datetime* objects. If in doubt, simply implement all of them.

tzinfo.utcoffset(*dt*)

Return offset of local time from UTC, as a *timedelta* object that is positive east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a *tzinfo* object represents both time zone and DST adjustments, *utcoffset()* should return their sum. If the UTC offset isn't known, return `None`. Else the value returned must be a *timedelta* object strictly between `-timedelta(hours=24)` and `timedelta(hours=24)` (the magnitude of the offset must be less than one day). Most implementations of *utcoffset()* will probably look like one of these two:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

If *utcoffset()* does not return `None`, *dst()* should not return `None` either.

The default implementation of *utcoffset()* raises *NotImplementedError*.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

`tzinfo.dst(`*dt*`)`

Return the daylight saving time (DST) adjustment, as a *timedelta* object or `None` if DST information isn't known. Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a *timedelta* object (see *utcoffset()* for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by *utcoffset()*, so there's no need to consult *dst()* unless you're interested in obtaining DST info separately. For example, *datetime.timetuple()* calls its *tzinfo* attribute's *dst()* method to determine how the `tm_isdst` flag should be set, and *tzinfo.fromutc()* calls *dst()* to account for DST changes when crossing time zones.

An instance *tz* of a *tzinfo* subclass that models both standard and daylight times must be consistent in this sense:

`tz.utcoffset(dt) - tz.dst(dt)`

must return the same result for every *datetime* *dt* with `dt.tzinfo == tz` For sane *tzinfo* subclasses, this expression yields the time zone's "standard offset", which should not depend on the date or the time, but only on geographic location. The implementation of *datetime.astimezone()* relies on this, but cannot detect violations; it's the programmer's responsibility to ensure it. If a *tzinfo* subclass cannot guarantee this, it may be able to override the default implementation of *tzinfo.fromutc()* to work correctly with `astimezone()` regardless.

Most implementations of *dst()* will probably look like one of these two:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

or

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.  Then

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

The default implementation of *dst()* raises *NotImplementedError*.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

`tzinfo.tzname(`*dt*`)`

Return the time zone name corresponding to the *datetime* object *dt*, as a string. Nothing about string names is defined by the *datetime* module, and there's no requirement that it mean anything in particular. For example, "GMT", "UTC", "-500", "-5:00", "EDT", "US/Eastern", "America/New York" are all valid replies. Return `None` if a string name isn't known. Note that this is a method rather than a fixed string primarily because some *tzinfo* subclasses will wish to return different names

depending on the specific value of *dt* passed, especially if the *tzinfo* class is accounting for daylight time.

The default implementation of *tzname()* raises *NotImplementedError*.

These methods are called by a *datetime* or *time* object, in response to their methods of the same names. A *datetime* object passes itself as the argument, and a *time* object passes `None` as the argument. A *tzinfo* subclass's methods should therefore be prepared to accept a *dt* argument of `None`, or of class *datetime*.

When `None` is passed, it's up to the class designer to decide the best response. For example, returning `None` is appropriate if the class wishes to say that time objects don't participate in the *tzinfo* protocols. It may be more useful for `utcoffset(None)` to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a *datetime* object is passed in response to a *datetime* method, `dt.tzinfo` is the same object as *self*. *tzinfo* methods can rely on this, unless user code calls *tzinfo* methods directly. The intent is that the *tzinfo* methods interpret *dt* as being in local time, and not need worry about objects in other timezones.

There is one more *tzinfo* method that a subclass may wish to override:

`tzinfo.fromutc(`*dt*`)`

This is called from the default *datetime.astimezone()* implementation. When called from that, `dt.tzinfo` is *self*, and *dt*'s date and time data are to be viewed as expressing a UTC time. The purpose of *fromutc()* is to adjust the date and time data, returning an equivalent datetime in *self*'s local time.

Most *tzinfo* subclasses should be able to inherit the default *fromutc()* implementation without problems. It's strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default *fromutc()* implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of `astimezone()` and *fromutc()* may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default *fromutc()* implementation acts like:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

In the following `tzinfo_examples.py` file there are some examples of *tzinfo* classes:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)
```

(continues on next page)

(continued from previous page)

```
# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()
```

(continues on next page)

(continued from previous page)

```
# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end
```

(continues on next page)

(continued from previous page)

```
class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
```

(continues on next page)

(continued from previous page)

```
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

Note that there are unavoidable subtleties twice per year in a *tzinfo* subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the second Sunday in March, and ends the minute after 1:59 (EDT) on the first Sunday in November:

```
  UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

When DST starts (the "start" line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn't really make sense on that day, so `astimezone(Eastern)` won't deliver a result with `hour == 2` on the day DST begins. For example, at the Spring forward transition of 2016, we get

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

When DST ends (the "end" line), there's a potentially worse problem: there's an hour that can't be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that's times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. `astimezone()` mimics the local clock's behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern, but earlier times have the *fold* attribute set to 0 and the later times have it set to 1. For example, at the Fall back transition of 2016, we get

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Note that the *datetime* instances that differ only by the value of the *fold* attribute are considered equal in comparisons.

Applications that can't bear wall-time ambiguities should explicitly check the value of the *fold* attribute or avoid using hybrid *tzinfo* subclasses; there are no ambiguities when using *timezone*, or any other fixed-offset *tzinfo* subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

See also:

dateutil.tz The standard library has *timezone* class for handling arbitrary fixed offsets from UTC and *timezone.utc* as UTC timezone instance.

dateutil.tz library brings the *IANA timezone database* (also known as the Olson database) to Python and its usage is recommended.

IANA timezone database The Time Zone Database (often called tz, tzdata or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.

8.1.7 `timezone` Objects

The *timezone* class is a subclass of *tzinfo*, each instance of which represents a timezone defined by a fixed offset from UTC. Note that objects of this class cannot be used to represent timezone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

class datetime.timezone(*offset*, *name=None*)

The *offset* argument must be specified as a *timedelta* object representing the difference between the local time and UTC. It must be strictly between `-timedelta(hours=24)` and `timedelta(hours=24)`, otherwise *ValueError* is raised.

The *name* argument is optional. If specified it must be a string that will be used as the value returned by the *datetime.tzname()* method.

New in version 3.2.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

timezone.utcoffset(*dt*)

Return the fixed value specified when the *timezone* instance is constructed. The *dt* argument is ignored. The return value is a *timedelta* instance equal to the difference between the local time and UTC.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

timezone.tzname(*dt*)

Return the fixed value specified when the *timezone* instance is constructed. If *name* is not provided in

the constructor, the name returned by `tzname(dt)` is generated from the value of the `offset` as follows. If *offset* is `timedelta(0)`, the name is "UTC", otherwise it is a string 'UTC±HH:MM', where ± is the sign of `offset`, HH and MM are two digits of `offset.hours` and `offset.minutes` respectively.

Changed in version 3.6: Name generated from `offset=timedelta(0)` is now plain 'UTC', not 'UTC+00:00'.

`timezone.dst(`*dt*`)`
Always returns `None`.

`timezone.fromutc(`*dt*`)`
Return `dt + offset`. The *dt* argument must be an aware *datetime* instance, with `tzinfo` set to `self`.

Class attributes:

`timezone.utc`
The UTC timezone, `timezone(timedelta(0))`.

8.1.8 `strftime()` and `strptime()` Behavior

date, *datetime*, and *time* objects all support a `strftime(format)` method, to create a string representing the time under the control of an explicit format string. Broadly speaking, `d.strftime(fmt)` acts like the *time* module's `time.strftime(fmt, d.timetuple())` although not all objects support a `timetuple()` method.

Conversely, the *datetime.strptime()* class method creates a *datetime* object from a string representing a date and time and a corresponding format string. `datetime.strptime(date_string, format)` is equivalent to `datetime(*(time.strptime(date_string, format)[0:6]))`, except when the format includes sub-second components or timezone offset information, which are supported in `datetime.strptime` but are discarded by `time.strptime`.

For *time* objects, the format codes for year, month, and day should not be used, as time objects have no such values. If they're used anyway, `1900` is substituted for the year, and `1` for the month and day.

For *date* objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as *date* objects have no such values. If they're used anyway, `0` is substituted for them.

The full set of format codes supported varies across platforms, because Python calls the platform C library's `strftime()` function, and platform variations are common. To see the full set of format codes supported on your platform, consult the *strftime(3)* documentation.

The following is a list of all the format codes that the C standard (1989 version) requires, and these work on all platforms with a standard C implementation. Note that the 1999 version of the C standard added additional format codes.

Directive	Meaning	Example	Notes
%a	Weekday as locale's abbreviated name.	Sun, Mon, …, Sat (en_US); So, Mo, …, Sa (de_DE)	(1)
%A	Weekday as locale's full name.	Sunday, Monday, …, Saturday (en_US); Sonntag, Montag, …, Samstag (de_DE)	(1)
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, …, 6	
%d	Day of the month as a zero-padded decimal number.	01, 02, …, 31	
%b	Month as locale's abbreviated name.	Jan, Feb, …, Dec (en_US); Jan, Feb, …, Dez (de_DE)	(1)
%B	Month as locale's full name.	January, February, …, December (en_US); Januar, Februar, …, Dezember (de_DE)	(1)
%m	Month as a zero-padded decimal number.	01, 02, …, 12	
%y	Year without century as a zero-padded decimal number.	00, 01, …, 99	
%Y	Year with century as a decimal number.	0001, 0002, …, 2013, 2014, …, 9998, 9999	(2)
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, …, 23	
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, …, 12	
%p	Locale's equivalent of either AM or PM.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%M	Minute as a zero-padded decimal number.	00, 01, …, 59	
%S	Second as a zero-padded decimal number.	00, 01, …, 59	(4)
%f	Microsecond as a decimal number, zero-padded on the left.	000000, 000001, …, 999999	(5)
%z	UTC offset in the form	(empty), +0000, -	(6)

Several additional directives not required by the C89 standard are included for convenience. These parameters all correspond to ISO 8601 date values. These may not be available on all platforms when used with the `strftime()` method. The ISO 8601 year and ISO 8601 week directives are not interchangeable with the year and week number directives above. Calling `strptime()` with incomplete or ambiguous ISO 8601 directives will raise a *ValueError*.

Directive	Meaning	Example	Notes
`%G`	ISO 8601 year with century representing the year that contains the greater part of the ISO week (`%V`).	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(8)
`%u`	ISO 8601 weekday as a decimal number where 1 is Monday.	1, 2, ..., 7	
`%V`	ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4.	01, 02, ..., 53	(8)

New in version 3.6: `%G`, `%u` and `%V` were added.

Notes:

(1) Because the format depends on the current locale, care should be taken when making assumptions about the output value. Field orderings will vary (for example, "month/day/year" versus "day/month/year"), and the output may contain Unicode characters encoded using the locale's default encoding (for example, if the current locale is `ja_JP`, the default encoding could be any one of `eucJP`, `SJIS`, or `utf-8`; use *locale.getlocale()* to determine the current locale's encoding).

(2) The `strptime()` method can parse years in the full [1, 9999] range, but years < 1000 must be zero-filled to 4-digit width.

Changed in version 3.2: In previous versions, `strftime()` method was restricted to years >= 1900.

Changed in version 3.3: In version 3.2, `strftime()` method was restricted to years >= 1000.

(3) When used with the `strptime()` method, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.

(4) Unlike the *time* module, the *datetime* module does not support leap seconds.

(5) When used with the `strptime()` method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).

(6) For a naive object, the `%z` and `%Z` format codes are replaced by empty strings.

For an aware object:

`%z` `utcoffset()` is transformed into a string of the form ±HHMM[SS[.ffffff]], where HH is a 2-digit string giving the number of UTC offset hours, MM is a 2-digit string giving the number of UTC offset minutes, SS is a 2-digit string giving the number of UTC offset seconds and ffffff is a 6-digit string giving the number of UTC offset microseconds. The ffffff part is omitted when the offset is a whole number of seconds and both the ffffff and the SS part is omitted when the offset is a whole number of minutes. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

Changed in version 3.7: When the `%z` directive is provided to the `strptime()` method, the UTC offsets can have a colon as a separator between hours, minutes and seconds. For example, `'+01:00:00'` will be parsed as an offset of one hour. In addition, providing `'Z'` is identical to `'+00:00'`.

`%Z` If `tzname()` returns `None`, `%Z` is replaced by an empty string. Otherwise `%Z` is replaced by the returned value, which must be a string.

Changed in version 3.2: When the `%z` directive is provided to the `strptime()` method, an aware *datetime* object will be produced. The `tzinfo` of the result will be set to a *timezone* instance.

(7) When used with the `strptime()` method, `%U` and `%W` are only used in calculations when the day of the week and the calendar year (`%Y`) are specified.

(8) Similar to `%U` and `%W`, `%V` is only used in calculations when the day of the week and the ISO year (`%G`) are specified in a `strptime()` format string. Also note that `%G` and `%Y` are not interchangeable.

8.2 `calendar` — General calendar-related functions

Source code: Lib/calendar.py

This module allows you to output calendars like the Unix `cal` program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use *setfirstweekday()* to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers. For related functionality, see also the *datetime* and *time* modules.

The functions and classes defined in this module use an idealized calendar, the current Gregorian calendar extended indefinitely in both directions. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book "Calendrical Calculations", where it's the base calendar for all computations. Zero and negative years are interpreted as prescribed by the ISO 8601 standard. Year 0 is 1 BC, year -1 is 2 BC, and so on.

class calendar.Calendar(*firstweekday=0*)

Creates a *Calendar* object. *firstweekday* is an integer specifying the first day of the week. `0` is Monday (the default), `6` is Sunday.

A *Calendar* object provides several methods that can be used for preparing the calendar data for formatting. This class doesn't do any formatting itself. This is the job of subclasses.

Calendar instances have the following methods:

iterweekdays()

Return an iterator for the week day numbers that will be used for one week. The first value from the iterator will be the same as the value of the *firstweekday* property.

itermonthdates(*year*, *month*)

Return an iterator for the month *month* (1–12) in the year *year*. This iterator will return all days (as *datetime.date* objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

itermonthdays(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will simply be day of the month numbers. For the days outside of the specified month, the day number is `0`.

itermonthdays2(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a day of the month number and a week day number.

itermonthdays3(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a year, a month and a day of the month numbers.

New in version 3.7.

itermonthdays4(*year*, *month*)
Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a year, a month, a day of the month, and a day of the week numbers.

New in version 3.7.

monthdatescalendar(*year*, *month*)
Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven *datetime.date* objects.

monthdays2calendar(*year*, *month*)
Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven tuples of day numbers and weekday numbers.

monthdayscalendar(*year*, *month*)
Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven day numbers.

yeardatescalendar(*year*, *width=3*)
Return the data for the specified year ready for formatting. The return value is a list of month rows. Each month row contains up to *width* months (defaulting to 3). Each month contains between 4 and 6 weeks and each week contains 1–7 days. Days are *datetime.date* objects.

yeardays2calendar(*year*, *width=3*)
Return the data for the specified year ready for formatting (similar to *yeardatescalendar()*). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this month are zero.

yeardayscalendar(*year*, *width=3*)
Return the data for the specified year ready for formatting (similar to *yeardatescalendar()*). Entries in the week lists are day numbers. Day numbers outside this month are zero.

class calendar.TextCalendar(*firstweekday=0*)
This class can be used to generate plain text calendars.

TextCalendar instances have the following methods:

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)
Return a month's calendar in a multi-line string. If *w* is provided, it specifies the width of the date columns, which are centered. If *l* is given, it specifies the number of lines that each week will use. Depends on the first weekday as specified in the constructor or set by the *setfirstweekday()* method.

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)
Print a month's calendar as returned by *formatmonth()*.

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
Return a *m*-column calendar for an entire year as a multi-line string. Optional parameters *w*, *l*, and *c* are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as specified in the constructor or set by the *setfirstweekday()* method. The earliest year for which a calendar can be generated is platform-dependent.

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
Print the calendar for an entire year as returned by *formatyear()*.

class calendar.HTMLCalendar(*firstweekday=0*)
This class can be used to generate HTML calendars.

`HTMLCalendar` instances have the following methods:

formatmonth(*theyear*, *themonth*, *withyear=True*)
Return a month's calendar as an HTML table. If *withyear* is true the year will be included in the header, otherwise just the month name will be used.

formatyear(*theyear*, *width=3*)
Return a year's calendar as an HTML table. *width* (defaulting to 3) specifies the number of months per row.

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)
Return a year's calendar as a complete HTML page. *width* (defaulting to 3) specifies the number of months per row. *css* is the name for the cascading style sheet to be used. *None* can be passed if no style sheet should be used. *encoding* specifies the encoding to be used for the output (defaulting to the system default encoding).

`HTMLCalendar` has the following attributes you can override to customize the CSS classes used by the calendar:

cssclasses
A list of CSS classes used for each weekday. The default class list is:

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

more styles can be added for each day:

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red"]
```

Note that the length of this list must be seven items.

cssclass_noday
The CSS class for a weekday occurring in the previous or coming month.

New in version 3.7.

cssclasses_weekday_head
A list of CSS classes used for weekday names in the header row. The default is the same as *cssclasses*.

New in version 3.7.

cssclass_month_head
The month's head CSS class (used by `formatmonthname()`). The default value is `"month"`.

New in version 3.7.

cssclass_month
The CSS class for the whole month's table (used by *formatmonth()*). The default value is `"month"`.

New in version 3.7.

cssclass_year
The CSS class for the whole year's table of tables (used by *formatyear()*). The default value is `"year"`.

New in version 3.7.

cssclass_year_head
The CSS class for the table head for the whole year (used by *formatyear()*). The default value is `"year"`.

New in version 3.7.

Note that although the naming for the above described class attributes is singular (e.g. `cssclass_month cssclass_noday`), one can replace the single CSS class with a space separated list of CSS classes, for example:

```
"text-bold text-red"
```

Here is an example how `HTMLCalendar` can be customized:

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.**LocaleTextCalendar**(*firstweekday=0*, *locale=None*)

This subclass of *TextCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)

This subclass of *HTMLCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

Note: The `formatweekday()` and `formatmonthname()` methods of these two classes temporarily change the current locale to the given *locale*. Because the current locale is a process-wide setting, they are not thread-safe.

For simple text calendars this module provides the following functions.

calendar.**setfirstweekday**(*weekday*)

Sets the weekday (`0` is Monday, `6` is Sunday) to start each week. The values `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are provided for convenience. For example, to set the first weekday to Sunday:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()

Returns the current setting for the weekday to start each week.

calendar.**isleap**(*year*)

Returns *True* if *year* is a leap year, otherwise *False*.

calendar.**leapdays**(*y1*, *y2*)

Returns the number of leap years in the range from *y1* to *y2* (exclusive), where *y1* and *y2* are years.

This function works for ranges spanning a century change.

calendar.**weekday**(*year*, *month*, *day*)

Returns the day of the week (`0` is Monday) for *year* (`1970`–…), *month* (`1`–`12`), *day* (`1`–`31`).

calendar.**weekheader**(*n*)

Return a header containing abbreviated weekday names. *n* specifies the width in characters for one weekday.

`calendar.monthrange`(*year*, *month*)
Returns weekday of first day of the month and number of days in month, for the specified *year* and *month*.

`calendar.monthcalendar`(*year*, *month*)
Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month a represented by zeros. Each week begins with Monday unless set by *setfirstweekday()*.

`calendar.prmonth`(*theyear*, *themonth*, *w=0*, *l=0*)
Prints a month's calendar as returned by *month()*.

`calendar.month`(*theyear*, *themonth*, *w=0*, *l=0*)
Returns a month's calendar in a multi-line string using the `formatmonth()` of the *TextCalendar* class.

`calendar.prcal`(*year*, *w=0*, *l=0*, *c=6*, *m=3*)
Prints the calendar for an entire year as returned by *calendar()*.

`calendar.calendar`(*year*, *w=2*, *l=1*, *c=6*, *m=3*)
Returns a 3-column calendar for an entire year as a multi-line string using the `formatyear()` of the *TextCalendar* class.

`calendar.timegm`(*tuple*)
An unrelated but handy function that takes a time tuple such as returned by the *gmtime()* function in the *time* module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, *time.gmtime()* and *timegm()* are each others' inverse.

The *calendar* module exports the following data attributes:

`calendar.day_name`
An array that represents the days of the week in the current locale.

`calendar.day_abbr`
An array that represents the abbreviated days of the week in the current locale.

`calendar.month_name`
An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_name[0]` is the empty string.

`calendar.month_abbr`
An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_abbr[0]` is the empty string.

See also:

Module *datetime* Object-oriented interface to dates and times with similar functionality to the *time* module.

Module *time* Low-level time related functions.

8.3 `collections` — Container datatypes

Source code: Lib/collections/___init___.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, *dict*, *list*, *set*, and *tuple*.

namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

Changed in version 3.3: Moved *Collections Abstract Base Classes* to the *collections.abc* module. For backwards compatibility, they continue to be visible in this module through Python 3.7. Subsequently, they will be removed entirely.

8.3.1 ChainMap objects

New in version 3.3.

A *ChainMap* class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple *update()* calls.

The class can be used to simulate nested scopes and is useful in templating.

class collections.ChainMap(**maps*)

A *ChainMap* groups multiple dicts or other mappings together to create a single, updateable view. If no *maps* are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.

The underlying mappings are stored in a list. That list is public and can be accessed or updated using the *maps* attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A *ChainMap* incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in *ChainMap*.

All of the usual dictionary methods are supported. In addition, there is a *maps* attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

maps

A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

new_child(*m=None*)

Returns a new *ChainMap* containing a new map followed by all of the maps in the current instance. If `m` is specified, it becomes the new map at the front of the list of mappings; if not specified, an empty dict is used, so that a call to `d.new_child()` is equivalent to: `ChainMap({}, *d.maps)`. This method is used for creating subcontexts that can be updated without altering values in any of the parent mappings.

Changed in version 3.4: The optional `m` parameter was added.

parents

Property returning a new *ChainMap* containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search. Use cases are similar to those for the `nonlocal` keyword used in *nested scopes*. The use cases also parallel those for the built-in *super()* function. A reference to `d.parents` is equivalent to: `ChainMap(*d.maps[1:])`.

See also:

- The MultiContext class in the Enthought CodeTools package has options to support writing to any mapping in the chain.
- Django's Context class for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the *new_child()* method and the *parents* property.
- The Nested Contexts recipe has options to control whether writes and other mutations apply only to the first mapping or to any mapping in the chain.
- A greatly simplified read-only version of Chainmap.

ChainMap Examples and Recipes

This section shows various approaches to working with chained maps.

Example of simulating Python's internal lookup chain:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Example of letting user specified command-line arguments take precedence over environment variables which in turn take precedence over default values:

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k:v for k, v in vars(namespace).items() if v}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Example patterns for using the *ChainMap* class to simulate nested contexts:

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x']                  # Get first key in the chain of contexts
d['x'] = 1              # Set value in current context
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

The *ChainMap* class only makes updates (writes and deletions) to the first mapping in the chain while lookups will search the full chain. However, if deep writes and deletions are desired, it is easy to make a subclass that updates keys found deeper in the chain:

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.3.2 Counter objects

A counter tool is provided to support convenient and rapid tallies. For example:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.Counter([*iterable-or-mapping*])

A *Counter* is a *dict* subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The *Counter* class is similar to bags or multisets in other languages.

Elements are counted from an *iterable* or initialized from another *mapping* (or counter):

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a *KeyError*:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

Setting a count to zero does not remove an element from a counter. Use `del` to remove it entirely:

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

New in version 3.1.

Counter objects support three methods beyond those available for all dictionaries:

elements()
Return an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order. If an element's count is less than one, *elements()* will ignore it.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*]**)**
Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`, *most_common()* returns *all* elements in the counter. Elements with equal counts are ordered arbitrarily:

```
>>> Counter('abracadabra').most_common(3)  # doctest: +SKIP
[('a', 5), ('r', 2), ('b', 2)]
```

subtract([*iterable-or-mapping*]**)**
Elements are subtracted from an *iterable* or from another *mapping* (or counter). Like *dict.update()* but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

New in version 3.2.

The usual dictionary methods are available for *Counter* objects except for two which work differently for counters.

fromkeys(*iterable***)**
This class method is not implemented for *Counter* objects.

update([*iterable-or-mapping*]**)**
Elements are counted from an *iterable* or added-in from another *mapping* (or counter). Like

dict.update() but adds counts instead of replacing them. Also, the *iterable* is expected to be a sequence of elements, not a sequence of `(key, value)` pairs.

Common patterns for working with *Counter* objects:

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
+c                              # remove zero and negative counts
```

Several mathematical operations are provided for combining *Counter* objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x]) # doctest: +SKIP
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

Unary addition and subtraction are shortcuts for adding an empty counter or subtracting from an empty counter.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

New in version 3.3: Added support for unary plus, unary minus, and in-place multiset operations.

Note: Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The *Counter* class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you *could* store anything in the value field.
- The *most_common()* method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for *update()* and *subtract()* which allow negative and zero values for both inputs and outputs.
- The multiset methods are designed only for use cases with positive values. The inputs may be negative

or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.

- The *elements()* method requires integer counts. It ignores zero and negative counts.

See also:

- Bag class in Smalltalk.
- Wikipedia entry for Multisets.
- C++ multisets tutorial with examples.
- For mathematical operations on multisets and their use cases, see *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19.*
- To enumerate all distinct multisets of a given size over a given set of elements, see *itertools.combinations_with_replacement()*:

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

8.3.3 deque objects

class collections.deque([*iterable*[, *maxlen*]])

Returns a new deque object initialized left-to-right (using *append()*) with data from *iterable*. If *iterable* is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced "deck" and is short for "double-ended queue"). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Though *list* objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for `pop(0)` and `insert(0, v)` operations which change both the size and position of the underlying data representation.

If *maxlen* is not specified or is `None`, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the `tail` filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Deque objects support the following methods:

append(*x*)
Add *x* to the right side of the deque.

appendleft(*x*)
Add *x* to the left side of the deque.

clear()
Remove all elements from the deque leaving it with length 0.

copy()
Create a shallow copy of the deque.

New in version 3.5.

count(*x*)
Count the number of deque elements equal to *x*.

New in version 3.2.

extend(*iterable*)
Extend the right side of the deque by appending elements from the iterable argument.

extendleft(*iterable*)
Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

index(*x*[, *start*[, *stop*]])
Return the position of *x* in the deque (at or after index *start* and before index *stop*). Returns the first match or raises *ValueError* if not found.

New in version 3.5.

insert(*i*, *x*)
Insert *x* into the deque at position *i*.

If the insertion would cause a bounded deque to grow beyond *maxlen*, an *IndexError* is raised.

New in version 3.5.

pop()
Remove and return an element from the right side of the deque. If no elements are present, raises an *IndexError*.

popleft()
Remove and return an element from the left side of the deque. If no elements are present, raises an *IndexError*.

remove(*value*)
Remove the first occurrence of *value*. If not found, raises a *ValueError*.

reverse()
Reverse the elements of the deque in-place and then return `None`.

New in version 3.2.

rotate(*n=1*)
Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left.

When the deque is not empty, rotating one step to the right is equivalent to `d.appendleft(d.pop())`, and rotating one step to the left is equivalent to `d.append(d.popleft())`.

Deque objects also provide one read-only attribute:

maxlen
Maximum size of a deque or `None` if unbounded.

New in version 3.1.

In addition to the above, deques support iteration, pickling, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, membership testing with the `in` operator, and subscript references such as `d[-1]`. Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

Starting in version 3.5, deques support `__add__()`, `__mul__()`, and `__imul__()`.

Example:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I
```

(continues on next page)

(continued from previous page)

```
>>> d.append('j')                     # add a new entry to the right side
>>> d.appendleft('f')                 # add a new entry to the left side
>>> d                                 # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                           # return and remove the rightmost item
'j'
>>> d.popleft()                       # return and remove the leftmost item
'f'
>>> list(d)                           # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                              # peek at leftmost item
'g'
>>> d[-1]                             # peek at rightmost item
'i'

>>> list(reversed(d))                 # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                          # search the deque
True
>>> d.extend('jkl')                   # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                       # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                      # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))                # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                         # empty the deque
>>> d.pop()                           # cannot pop from an empty deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')               # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque Recipes

This section shows various approaches to working with deques.

Bounded length deques provide functionality similar to the `tail` filter in Unix:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
```

(continues on next page)

(continued from previous page)

```
    with open(filename) as f:
        return deque(f, n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

A round-robin scheduler can be implemented with input iterators stored in a *deque*. Values are yielded from the active iterator in position zero. If that iterator is exhausted, it can be removed with *popleft()*; otherwise, it can be cycled back to the end with the *rotate()* method:

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

The *rotate()* method provides a way to implement *deque* slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement *deque* slicing, use a similar approach applying *rotate()* to bring a target element to the left side of the deque. Remove old entries with *popleft()*, add new entries with *extend()*, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`.

8.3.4 `defaultdict` objects

class collections.**defaultdict**([*default_factory*[, ...]])

Returns a new dictionary-like object. *defaultdict* is a subclass of the built-in *dict* class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the *dict* class and is not documented here.

The first argument provides the initial value for the *default_factory* attribute; it defaults to `None`. All remaining arguments are treated the same as if they were passed to the *dict* constructor, including keyword arguments.

defaultdict objects support the following method in addition to the standard *dict* operations:

`__missing__`(*key*)
If the *default_factory* attribute is `None`, this raises a *KeyError* exception with the *key* as argument.

If *default_factory* is not `None`, it is called without arguments to provide a default value for the given *key*, this value is inserted in the dictionary for the *key*, and returned.

If calling *default_factory* raises an exception this exception is propagated unchanged.

This method is called by the `__getitem__()` method of the *dict* class when the requested key is not found; whatever it returns or raises is then returned or raised by `__getitem__()`.

Note that *__missing__()* is *not* called for any operations besides `__getitem__()`. This means that `get()` will, like normal dictionaries, return `None` as a default rather than using *default_factory*.

defaultdict objects support the following instance variable:

`default_factory`
This attribute is used by the *__missing__()* method; it is initialized from the first argument to the constructor, if present, or to `None`, if absent.

`defaultdict` Examples

Using *list* as the *default_factory*, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the *default_factory* function which returns an empty *list*. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than an equivalent technique using *dict.setdefault()*:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the *default_factory* to *int* makes the *defaultdict* useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

When a letter is first encountered, it is missing from the mapping, so the *default_factory* function calls *int()* to supply a default count of zero. The increment operation then builds up the count for each letter.

The function *int()* which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use a lambda function which can supply any constant value (not just zero):

```
>>> def constant_factory(value):
...     return lambda: value
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the *default_factory* to *set* makes the *defaultdict* useful for building a dictionary of sets:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.3.5 `namedtuple()` Factory Function for Tuples with Named Fields

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

`collections.namedtuple`(*typename*, *field_names*, ***, *rename=False*, *defaults=None*, *module=None*)

Returns a new tuple subclass named *typename*. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful `__repr__()` method which lists the tuple contents in a `name=value` format.

The *field_names* are a sequence of strings such as `['x', 'y']`. Alternatively, *field_names* can be a single string with each fieldname separated by whitespace and/or commas, for example `'x y'` or `'x, y'`.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a *keyword* such as *class*, *for*, *return*, *global*, *pass*, or *raise*.

If *rename* is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '_1', 'ghi', '_3']`, eliminating the keyword `def` and the duplicate fieldname `abc`.

defaults can be `None` or an *iterable* of default values. Since fields with a default value must come after any fields without a default, the *defaults* are applied to the rightmost parameters. For example, if the fieldnames are `['x', 'y', 'z']` and the defaults are `(1, 2)`, then `x` will be a required argument, `y` will default to `1`, and `z` will default to `2`.

If *module* is defined, the `__module__` attribute of the named tuple is set to that value.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

Changed in version 3.1: Added support for *rename*.

Changed in version 3.6: The *verbose* and *rename* parameters became *keyword-only arguments*.

Changed in version 3.6: Added the *module* parameter.

Changed in version 3.7: Remove the *verbose* parameter and the `_source` attribute.

Changed in version 3.7: Added the *defaults* parameter and the `_field_defaults` attribute.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Named tuples are especially useful for assigning field names to result tuples returned by the *csv* or *sqlite3* modules:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

classmethod somenamedtuple.**_make**(*iterable*)

Class method that makes a new instance from an existing sequence or iterable.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple._asdict()
Return a new *OrderedDict* which maps field names to their corresponding values:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

Changed in version 3.1: Returns an *OrderedDict* instead of a regular *dict*.

somenamedtuple._replace(***kwargs*)
Return a new instance of the named tuple replacing specified fields with new values:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum],␣
↪timestamp=time.now())
```

somenamedtuple._fields
Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple._fields_defaults
Dictionary mapping field names to default values.

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._fields_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

To retrieve a field whose name is stored in a string, use the *getattr()* function:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in tut-unpacking-arguments):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

The subclass shown above sets `__slots__` to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the *_fields* attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Docstrings can be customized by making direct assignments to the `__doc__` fields:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

Changed in version 3.5: Property docstrings became writeable.

Default values can be implemented by using *_replace()* to customize a prototype instance:

```
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
>>> janes_account = default_account._replace(owner='Jane')
```

See also:

- Recipe for named tuple abstract base class with a metaclass mix-in by Jan Kaliszewski. Besides providing an *abstract base class* for named tuples, it also supports an alternate *metaclass*-based constructor that is convenient for use cases where named tuples are being subclassed.
- See *types.SimpleNamespace()* for a mutable namespace based on an underlying dictionary instead of a tuple.
- See *typing.NamedTuple()* for a way to add type hints for named tuples.

8.3.6 `OrderedDict` objects

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

`class collections.OrderedDict([items])`

Return an instance of a dict subclass, supporting the usual *dict* methods. An *OrderedDict* is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the

original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

New in version 3.1.

popitem(*last=True*)
The *popitem()* method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if *last* is true or FIFO (first-in, first-out) order if false.

move_to_end(*key*, *last=True*)
Move an existing *key* to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises *KeyError* if the *key* does not exist:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

New in version 3.2.

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using *reversed()*.

Equality tests between *OrderedDict* objects are order-sensitive and are implemented as `list(od1.items())==list(od2.items())`. Equality tests between *OrderedDict* objects and other *Mapping* objects are order-insensitive like regular dictionaries. This allows *OrderedDict* objects to be substituted anywhere a regular dictionary is used.

Changed in version 3.5: The items, keys, and values *views* of *OrderedDict* now support reverse iteration using *reversed()*.

Changed in version 3.6: With the acceptance of **PEP 468**, order is retained for keyword arguments passed to the *OrderedDict* constructor and its `update()` method.

`OrderedDict` Examples and Recipes

Since an ordered dictionary remembers its insertion order, it can be used in conjunction with sorting to make a sorted dictionary:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

The new sorted dictionaries maintain their sort order when entries are deleted. But when new keys are added, the keys are appended to the end and the sort is not maintained.

It is also straight-forward to create an ordered dictionary variant that remembers the order the keys were *last* inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        OrderedDict.__setitem__(self, key, value)
```

An ordered dictionary can be combined with the *Counter* class so that the counter remembers the order elements are first encountered:

```
class OrderedCounter(Counter, OrderedDict):
    'Counter that remembers the order elements are first encountered'

    def __repr__(self):
        return '%s(%r)' % (self.__class__.__name__, OrderedDict(self))

    def __reduce__(self):
        return self.__class__, (OrderedDict(self),)
```

8.3.7 UserDict objects

The class, *UserDict* acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from *dict*; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

class collections.UserDict([*initialdata*])

Class that simulates a dictionary. The instance's contents are kept in a regular dictionary, which is accessible via the *data* attribute of *UserDict* instances. If *initialdata* is provided, *data* is initialized with its contents; note that a reference to *initialdata* will not be kept, allowing it be used for other purposes.

In addition to supporting the methods and operations of mappings, *UserDict* instances provide the following attribute:

data

A real dictionary used to store the contents of the *UserDict* class.

8.3.8 UserList objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from *list*; however, this class can be easier to work with because the underlying list is accessible as an attribute.

class collections.UserList([*list*])

Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via

the *data* attribute of *UserList* instances. The instance's contents are initially set to a copy of *list*, defaulting to the empty list `[]`. *list* can be any iterable, for example a real Python list or a *UserList* object.

In addition to supporting the methods and operations of mutable sequences, *UserList* instances provide the following attribute:

data
A real *list* object used to store the contents of the *UserList* class.

Subclassing requirements: Subclasses of *UserList* are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

8.3.9 `UserString` objects

The class, *UserString* acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from *str*; however, this class can be easier to work with because the underlying string is accessible as an attribute.

class collections.**UserString**(*seq*)
Class that simulates a string object. The instance's content is kept in a regular string object, which is accessible via the *data* attribute of *UserString* instances. The instance's contents are initially set to a copy of *seq*. The *seq* argument can be any object which can be converted into a string using the built-in *str()* function.

In addition to supporting the methods and operations of strings, *UserString* instances provide the following attribute:

data
A real *str* object used to store the contents of the *UserString* class.

Changed in version 3.5: New methods `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable`, and `maketrans`.

8.4 `collections.abc` — Abstract Base Classes for Containers

New in version 3.3: Formerly, this module was part of the *collections* module.

Source code: Lib/_collections_abc.py

This module provides *abstract base classes* that can be used to test whether a class provides a particular interface; for example, whether it is hashable or whether it is a mapping.

8.4.1 Collections Abstract Base Classes

The collections module offers the following *ABCs*:

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		`__contains__`	
Hashable		`__hash__`	
Iterable		`__iter__`	
Iterator	*Iterable*	`__next__`	`__iter__`
Reversible	*Iterable*	`__reversed__`	
Generator	*Iterator*	`send`, `throw`	`close`, `__iter__`, `__next__`
Sized		`__len__`	
Callable		`__call__`	
Collection	*Sized*, *Iterable*, *Container*	`__contains__`, `__iter__`, `__len__`	
Sequence	*Reversible*, *Collection*	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	*Sequence*	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited *Sequence* methods and `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
ByteString	*Sequence*	`__getitem__`, `__len__`	Inherited *Sequence* methods
Set	*Collection*	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, and `isdisjoint`
MutableSet	*Set*	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Inherited *Set* methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
Mapping	*Collection*	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
MutableMapping	*Mapping*	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Inherited *Mapping* methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
MappingView	*Sized*		`__len__`
ItemsView	*MappingView*, *Set*		`__contains__`, `__iter__`
KeysView	*MappingView*, *Set*		`__contains__`, `__iter__`
ValuesView	*MappingView*, *Collection*		`__contains__`, `__iter__`
Awaitable		`__await__`	
Coroutine	*Awaitable*	`send`, `throw`	`close`
AsyncIterable		`__aiter__`	
AsyncIterator	*AsyncIterable*	`__anext__`	`__aiter__`
AsyncGenerator	*AsyncIterator*	`asend`, `athrow`	`aclose`, `__aiter__`, `__anext__`

class collections.abc.**Container**
class collections.abc.**Hashable**
class collections.abc.**Sized**
class collections.abc.**Callable**

ABCs for classes that provide respectively the methods `__contains__()`, `__hash__()`, `__len__()`, and `__call__()`.

class collections.abc.**Iterable**

ABC for classes that provide the `__iter__()` method.

Checking `isinstance(obj, Iterable)` detects classes that are registered as *Iterable* or that have an `__iter__()` method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is *iterable* is to call `iter(obj)`.

class collections.abc.**Collection**
ABC for sized iterable container classes.

New in version 3.6.

class collections.abc.**Iterator**
ABC for classes that provide the *__iter__()* and *__next__()* methods. See also the definition of *iterator*.

class collections.abc.**Reversible**
ABC for iterable classes that also provide the `__reversed__()` method.

New in version 3.6.

class collections.abc.**Generator**
ABC for generator classes that implement the protocol defined in **PEP 342** that extends iterators with the `send()`, `throw()` and `close()` methods. See also the definition of *generator*.

New in version 3.5.

class collections.abc.**Sequence**
class collections.abc.**MutableSequence**
class collections.abc.**ByteString**
ABCs for read-only and mutable *sequences*.

Implementation note: Some of the mixin methods, such as `__iter__()`, `__reversed__()` and `index()`, make repeated calls to the underlying `__getitem__()` method. Consequently, if `__getitem__()` is implemented with constant access speed, the mixin methods will have linear performance; however, if the underlying method is linear (as it would be with a linked list), the mixins will have quadratic performance and will likely need to be overridden.

Changed in version 3.5: The index() method added support for *stop* and *start* arguments.

class collections.abc.**Set**
class collections.abc.**MutableSet**
ABCs for read-only and mutable sets.

class collections.abc.**Mapping**
class collections.abc.**MutableMapping**
ABCs for read-only and mutable *mappings*.

class collections.abc.**MappingView**
class collections.abc.**ItemsView**
class collections.abc.**KeysView**
class collections.abc.**ValuesView**
ABCs for mapping, items, keys, and values *views*.

class collections.abc.**Awaitable**
ABC for *awaitable* objects, which can be used in `await` expressions. Custom implementations must provide the `__await__()` method.

Coroutine objects and instances of the *Coroutine* ABC are all instances of this ABC.

Note: In CPython, generator-based coroutines (generators decorated with *types.coroutine()* or *asyncio.coroutine()*) are *awaitables*, even though they do not have an `__await__()` method. Using

`isinstance(gencoro, Awaitable)` for them will return `False`. Use *inspect.isawaitable()* to detect them.

New in version 3.5.

class collections.abc.**Coroutine**
ABC for coroutine compatible classes. These implement the following methods, defined in coroutine-objects: `send()`, `throw()`, and `close()`. Custom implementations must also implement `__await__()`. All *Coroutine* instances are also instances of *Awaitable*. See also the definition of *coroutine*.

Note: In CPython, generator-based coroutines (generators decorated with *types.coroutine()* or *asyncio.coroutine()*) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Coroutine)` for them will return `False`. Use *inspect.isawaitable()* to detect them.

New in version 3.5.

class collections.abc.**AsyncIterable**
ABC for classes that provide `__aiter__` method. See also the definition of *asynchronous iterable*.

New in version 3.5.

class collections.abc.**AsyncIterator**
ABC for classes that provide `__aiter__` and `__anext__` methods. See also the definition of *asynchronous iterator*.

New in version 3.5.

class collections.abc.**AsyncGenerator**
ABC for asynchronous generator classes that implement the protocol defined in **PEP 525** and **PEP 492**.

New in version 3.6.

These ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full *Set* API, it is only necessary to supply the three underlying abstract methods: `__contains__()`, `__iter__()`, and `__len__()`. The ABC supplies the remaining methods such as `__and__()` and `isdisjoint()`:

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)
```

(continues on next page)

(continued from previous page)

```
    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

Notes on using *Set* and *MutableSet* as a mixin:

(1) Since some set operations create new sets, the default mixin methods need a way to create new instances from an iterable. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal classmethod called `_from_iterable()` which calls `cls(iterable)` to produce a new set. If the *Set* mixin is being used in a class with a different constructor signature, you will need to override `_from_iterable()` with a classmethod that can construct new instances from an iterable argument.

(2) To override the comparisons (presumably for speed, as the semantics are fixed), redefine `__le__()` and `__ge__()`, then the other operations will automatically follow suit.

(3) The *Set* mixin provides a `_hash()` method to compute a hash value for the set; however, `__hash__()` is not defined because not all sets are hashable or immutable. To add set hashability using mixins, inherit from both *Set()* and *Hashable()*, then define `__hash__ = Set._hash`.

See also:

- OrderedSet recipe for an example built on *MutableSet*.
- For more about ABCs, see the *abc* module and **PEP 3119**.

8.5 heapq — Heap queue algorithm

Source code: Lib/heapq.py

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all k, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, `heap[0]`.

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our pop method returns the smallest item, not the largest (called a "min heap" in textbooks; a "max heap" is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: `heap[0]` is the smallest item, and `heap.sort()` maintains the heap invariant!

To create a heap, use a list initialized to `[]`, or you can transform a populated list into a heap via function *heapify()*.

The following functions are provided:

heapq.**heappush**(*heap*, *item*)
Push the value *item* onto the *heap*, maintaining the heap invariant.

heapq.**heappop**(*heap*)
Pop and return the smallest item from the *heap*, maintaining the heap invariant. If the heap is empty, *IndexError* is raised. To access the smallest item without popping it, use `heap[0]`.

heapq.**heappushpop**(*heap*, *item*)
Push *item* on the heap, then pop and return the smallest item from the *heap*. The combined action runs more efficiently than *heappush()* followed by a separate call to *heappop()*.

heapq.**heapify**(*x*)
Transform list *x* into a heap, in-place, in linear time.

heapq.**heapreplace**(*heap*, *item*)
Pop and return the smallest item from the *heap*, and also push the new *item*. The heap size doesn't change. If the heap is empty, *IndexError* is raised.

This one step operation is more efficient than a *heappop()* followed by *heappush()* and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with *item*.

The value returned may be larger than the *item* added. If that isn't desired, consider using *heappushpop()* instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

The module also offers three general purpose functions based on heaps.

heapq.**merge**(**iterables*, *key=None*, *reverse=False*)
Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an *iterator* over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

Has two optional arguments which must be specified as keyword arguments.

key specifies a *key function* of one argument that is used to extract a comparison key from each input element. The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the input elements are merged as if each comparison were reversed. To achieve behavior similar to `sorted(itertools.chain(*iterables), reverse=True)`, all iterables must be sorted from largest to smallest.

Changed in version 3.5: Added the optional *key* and *reverse* parameters.

heapq.**nlargest**(*n*, *iterable*, *key=None*)
Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key, reverse=True)[:n]`.

heapq.**nsmallest**(*n*, *iterable*, *key=None*)
Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key)[:n]`.

The latter two functions perform best for smaller values of *n*. For larger values, it is more efficient to use the *sorted()* function. Also, when `n==1`, it is more efficient to use the built-in *min()* and *max()* functions. If repeated usage of these functions is required, consider turning the iterable into an actual heap.

8.5.1 Basic Examples

A heapsort can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is similar to `sorted(iterable)`, but unlike *sorted()*, this implementation is not stable.

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.5.2 Priority Queue Implementation Notes

A priority queue is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- Tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

Another solution to the problem of non-comparable tasks is to create a wrapper class that ignores the task item and only compares the priority field:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the entry as removed and add a new entry with the revised priority:

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.5.3 Theory

Heaps are arrays for which `a[k] <= a[2*k+1]` and `a[k] <= a[2*k+2]` for all k, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that `a[0]` is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are k, not `a[k]`:

```
                                0

             1                                  2

      3              4                5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

In the tree above, each cell *k* is topping `2*k+1` and `2*k+2`. In a usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell "wins" over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the "next" winner is to move some loser (let's say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an O(n log n) sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not "better" than the last 0'th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the "win" condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing "runs" (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised[1]. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you'll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0'th item on disk and get an input which may not fit in the current tournament (because the value "wins" over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a 'heap' module around. :-)

8.6 `bisect` — Array bisection algorithm

Source code: Lib/bisect.py

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over the more common approach. The module is called *`bisect`* because it uses a basic bisection algorithm to do its

[1] The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at "progressing" the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)

work. The source code may be most useful as a working example of the algorithm (the boundary conditions are already right!).

The following functions are provided:

bisect.bisect_left(*a*, *x*, *lo=0*, *hi=len(a)*)
: Locate the insertion point for *x* in *a* to maintain sorted order. The parameters *lo* and *hi* may be used to specify a subset of the list which should be considered; by default the entire list is used. If *x* is already present in *a*, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to `list.insert()` assuming that *a* is already sorted.

  The returned insertion point *i* partitions the array *a* into two halves so that `all(val < x for val in a[lo:i])` for the left side and `all(val >= x for val in a[i:hi])` for the right side.

bisect.bisect_right(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.bisect(*a*, *x*, *lo=0*, *hi=len(a)*)
: Similar to *bisect_left()*, but returns an insertion point which comes after (to the right of) any existing entries of *x* in *a*.

  The returned insertion point *i* partitions the array *a* into two halves so that `all(val <= x for val in a[lo:i])` for the left side and `all(val > x for val in a[i:hi])` for the right side.

bisect.insort_left(*a*, *x*, *lo=0*, *hi=len(a)*)
: Insert *x* in *a* in sorted order. This is equivalent to `a.insert(bisect.bisect_left(a, x, lo, hi), x)` assuming that *a* is already sorted. Keep in mind that the O(log n) search is dominated by the slow O(n) insertion step.

bisect.insort_right(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.insort(*a*, *x*, *lo=0*, *hi=len(a)*)
: Similar to *insort_left()*, but inserting *x* in *a* after any existing entries of *x*.

See also:

SortedCollection recipe that uses bisect to build a full-featured collection class with straight-forward search methods and support for a key-function. The keys are precomputed to save unnecessary calls to the key function during searches.

8.6.1 Searching Sorted Lists

The above *bisect()* functions are useful for finding insertion points but can be tricky or awkward to use for common searching tasks. The following five functions show how to transform them into the standard lookups for sorted lists:

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
```

(continues on next page)

(continued from previous page)

```
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.6.2 Other Examples

The *bisect()* function can be useful for numeric table lookups. This example uses *bisect()* to look up a letter grade for an exam score (say) based on a set of ordered numeric breakpoints: 90 and up is an 'A', 80 to 89 is a 'B', and so on:

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

Unlike the *sorted()* function, it does not make sense for the *bisect()* functions to have *key* or *reversed* arguments because that would lead to an inefficient design (successive calls to bisect functions would not "remember" all of the previous key lookups).

Instead, it is better to search a list of precomputed keys to find the index of the record in question:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]         # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.7 array — Efficient arrays of numeric values

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a *type code*, which is a single character. The following type codes are defined:

Type code	C Type	Python Type	Minimum size in bytes	Notes
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	Py_UNICODE	Unicode character	2	(1)
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	(2)
'Q'	unsigned long long	int	8	(2)
'f'	float	float	4	
'd'	double	float	8	

Notes:

(1) The 'u' type code corresponds to Python's obsolete unicode character (Py_UNICODE which is wchar_t). Depending on the platform, it can be 16 bits or 32 bits.

'u' will be removed together with the rest of the Py_UNICODE API.

Deprecated since version 3.3, will be removed in version 4.0.

(2) The 'q' and 'Q' type codes are available only if the platform C compiler used to build Python supports C long long, or, on Windows, __int64.

New in version 3.3.

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the itemsize attribute.

The module defines the following type:

class array.array(*typecode*[, *initializer*])

A new array whose items are restricted by *typecode*, and initialized from the optional *initializer* value, which must be a list, a *bytes-like object*, or iterable over elements of the appropriate type.

If given a list or string, the initializer is passed to the new array's *fromlist()*, *frombytes()*, or *fromunicode()* method (see below) to add initial items to the array. Otherwise, the iterable initializer is passed to the *extend()* method.

array.typecodes

A string with all available type codes.

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, *TypeError* is raised. Array objects also implement the buffer interface, and may be used wherever *bytes-like objects* are supported.

The following data items and methods are also supported:

`array.typecode`
The typecode character used to create the array.

`array.itemsize`
The length in bytes of one array item in the internal representation.

`array.append(`*x*`)`
Append a new item with value *x* to the end of the array.

`array.buffer_info()`
Return a tuple `(address, length)` giving the current memory address and the length in elements of the buffer used to hold array's contents. The size of the memory buffer in bytes can be computed as `array.buffer_info()[1] * array.itemsize`. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain `ioctl()` operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

Note: When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in bufferobjects.

`array.byteswap()`
"Byteswap" all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, *RuntimeError* is raised. It is useful when reading data from a file written on a machine with a different byte order.

`array.count(`*x*`)`
Return the number of occurrences of *x* in the array.

`array.extend(`*iterable*`)`
Append items from *iterable* to the end of the array. If *iterable* is another array, it must have *exactly* the same type code; if not, *TypeError* will be raised. If *iterable* is not an array, it must be iterable and its elements must be the right type to be appended to the array.

`array.frombytes(`*s*`)`
Appends items from the string, interpreting the string as an array of machine values (as if it had been read from a file using the *fromfile()* method).

New in version 3.2: *fromstring()* is renamed to *frombytes()* for clarity.

`array.fromfile(`*f*, *n*`)`
Read *n* items (as machine values) from the *file object* *f* and append them to the end of the array. If less than *n* items are available, *EOFError* is raised, but the items that were available are still inserted into the array. *f* must be a real built-in file object; something else with a `read()` method won't do.

`array.fromlist(`*list*`)`
Append items from the list. This is equivalent to `for x in list: a.append(x)` except that if there is a type error, the array is unchanged.

`array.fromstring()`
Deprecated alias for *frombytes()*.

`array.fromunicode(`*s*`)`
Extends this array with data from the given unicode string. The array must be a type `'u'` array; otherwise a *ValueError* is raised. Use `array.frombytes(unicodestring.encode(enc))` to append Unicode data to an array of some other type.

`array.index(`*x*`)`
Return the smallest *i* such that *i* is the index of the first occurrence of *x* in the array.

`array.insert(`*i*, *x*`)`
Insert a new item with value *x* in the array before position *i*. Negative values are treated as being relative to the end of the array.

`array.pop(`[*i*]`)`
Removes the item with the index *i* from the array and returns it. The optional argument defaults to `-1`, so that by default the last item is removed and returned.

`array.remove(`*x*`)`
Remove the first occurrence of *x* from the array.

`array.reverse()`
Reverse the order of the items in the array.

`array.tobytes()`
Convert the array to an array of machine values and return the bytes representation (the same sequence of bytes that would be written to a file by the *tofile()* method.)

New in version 3.2: *tostring()* is renamed to *tobytes()* for clarity.

`array.tofile(`*f*`)`
Write all items (as machine values) to the *file object* *f*.

`array.tolist()`
Convert the array to an ordinary list with the same items.

`array.tostring()`
Deprecated alias for *tobytes()*.

`array.tounicode()`
Convert the array to a unicode string. The array must be a type `'u'` array; otherwise a *ValueError* is raised. Use `array.tobytes().decode(enc)` to obtain a unicode string from an array of some other type.

When an array object is printed or converted to a string, it is represented as `array(typecode, initializer)`. The *initializer* is omitted if the array is empty, otherwise it is a string if the *typecode* is `'u'`, otherwise it is a list of numbers. The string is guaranteed to be able to be converted back to an array with the same type and value using *eval()*, so long as the *array* class has been imported using `from array import array`. Examples:

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

See also:

Module *struct* Packing and unpacking of heterogeneous binary data.

Module *xdrlib* Packing and unpacking of External Data Representation (XDR) data as used in some remote procedure call systems.

The Numerical Python Documentation The Numeric Python extension (NumPy) defines another array type; see http://www.numpy.org/ for further information about Numerical Python.

8.8 `weakref` — Weak references

Source code: Lib/weakref.py

The *weakref* module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, *garbage collection* is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The *WeakKeyDictionary* and *WeakValueDictionary* classes supplied by the *weakref* module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a *WeakValueDictionary*, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

WeakKeyDictionary and *WeakValueDictionary* use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. *WeakSet* implements the *set* interface, but keeps weak references to its elements, just like a *WeakKeyDictionary* does.

finalize provides a straight forward way to register a cleanup function to be called when an object is garbage collected. This is simpler to use than setting up a callback function on a raw weak reference, since the module automatically ensures that the finalizer remains alive until the object is collected.

Most programs should find that using one of these weak container types or *finalize* is all they need – it's not usually necessary to create your own weak references directly. The low-level machinery is exposed by the *weakref* module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some *file objects*, *generator*s, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects.

Changed in version 3.2: Added support for thread.lock, threading.Lock, and code objects.

Several built-in types such as *list* and *dict* do not directly support weak references but can add support through subclassing:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

Other built-in types such as *tuple* and *int* do not support weak references even when subclassed (This is an implementation detail and may be different across various Python implementations.).

Extension types can easily be made to support weak references; see weakref-support.

class weakref.ref(*object*[, *callback*])

Return a weak reference to *object*. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause *None* to be returned. If *callback* is provided and not *None*, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object's `__del__()` method.

Weak references are *hashable* if the *object* is hashable. They will maintain their hash value even after the *object* was deleted. If *hash()* is called the first time only after the *object* was deleted, the call will raise *TypeError*.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the *callback*). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

__callback__
This read-only attribute returns the callback currently associated to the weakref. If there is no callback or if the referent of the weakref is no longer alive then this attribute will have value `None`.

Changed in version 3.4: Added the *__callback__* attribute.

weakref.proxy(*object*[, *callback*])
Return a proxy to *object* which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either `ProxyType` or `CallableProxyType`, depending on whether *object* is callable. Proxy objects are not *hashable* regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. *callback* is the same as the parameter of the same name to the *ref()* function.

weakref.getweakrefcount(*object*)
Return the number of weak references and proxies which refer to *object*.

weakref.getweakrefs(*object*)
Return a list of all weak reference and proxy objects which refer to *object*.

class weakref.WeakKeyDictionary([*dict*])
Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

Note: Caution: Because a *WeakKeyDictionary* is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a *WeakKeyDictionary* because actions performed by the program during iteration may cause items in the dictionary to vanish "by magic" (as a side effect of garbage collection).

WeakKeyDictionary objects have an additional method that exposes the internal references directly. The references are not guaranteed to be "live" at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

WeakKeyDictionary.keyrefs()
Return an iterable of the weak references to the keys.

class weakref.WeakValueDictionary([*dict*])
Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

Note: Caution: Because a *WeakValueDictionary* is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a *WeakValueDictionary* because actions performed by the program during iteration may cause items in the dictionary to vanish "by magic" (as a side effect of garbage collection).

WeakValueDictionary objects have an additional method that has the same issues as the `keyrefs()` method of *WeakKeyDictionary* objects.

`WeakValueDictionary.valuerefs()`
Return an iterable of the weak references to the values.

`class weakref.WeakSet([elements])`
Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

`class weakref.WeakMethod(method)`
A custom *ref* subclass which simulates a weak reference to a bound method (i.e., a method defined on a class and looked up on an instance). Since a bound method is ephemeral, a standard weak reference cannot keep hold of it. *WeakMethod* has special code to recreate the bound method until either the object or the original function dies:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

New in version 3.4.

`class weakref.finalize(obj, func, *args, **kwargs)`
Return a callable finalizer object which will be called when *obj* is garbage collected. Unlike an ordinary weak reference, a finalizer will always survive until the reference object is collected, greatly simplifying lifecycle management.

A finalizer is considered *alive* until it is called (either explicitly or at garbage collection), and after that it is *dead*. Calling a live finalizer returns the result of evaluating `func(*arg, **kwargs)`, whereas calling a dead finalizer returns *None*.

Exceptions raised by finalizer callbacks during garbage collection will be shown on the standard error output, but cannot be propagated. They are handled in the same way as exceptions raised from an object's `__del__()` method or a weak reference's callback.

When the program exits, each remaining live finalizer is called unless its *atexit* attribute has been set to false. They are called in reverse order of creation.

A finalizer will never invoke its callback during the later part of the *interpreter shutdown* when module globals are liable to have been replaced by *None*.

`__call__()`
If *self* is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If *self* is dead then return *None*.

`detach()`
If *self* is alive then mark it as dead and return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

`peek()`
If *self* is alive then return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

`alive`
Property which is true if the finalizer is alive, false otherwise.

`atexit`
A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which *atexit* is true. They are called in reverse order of creation.

Note: It is important to ensure that *func*, *args* and *kwargs* do not own any references to *obj*, either directly or indirectly, since otherwise *obj* will never be garbage collected. In particular, *func* should not be a bound method of *obj*.

New in version 3.4.

`weakref.ReferenceType`
The type object for weak references objects.

`weakref.ProxyType`
The type object for proxies of objects which are not callable.

`weakref.CallableProxyType`
The type object for proxies of callable objects.

`weakref.ProxyTypes`
Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

`exception weakref.ReferenceError`
Exception raised when a proxy object is used but the underlying object has been collected. This is the same as the standard *ReferenceError* exception.

See also:

PEP 205 - Weak References The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

8.8.1 Weak Reference Objects

Weak reference objects have no methods and no attributes besides *ref.__callback__*. A weak reference object allows the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
```

(continues on next page)

(continued from previous page)

```
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns *None*:

```
>>> del o, o2
>>> print(r())
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

Using a separate test for "liveness" creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of *ref* objects can be created through subclassing. This is used in the implementation of the *WeakValueDictionary* to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of *ref* can be used to store additional information about an object and affect the value that's returned when the referent is accessed:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.8.2 Example

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.8.3 Finalizer Objects

The main benefit of using *finalize* is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")  #doctest:+ELLIPSIS
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

The finalizer can be called directly as well. However the finalizer will invoke the callback at most once.

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

You can unregister a finalizer using its *detach()* method. This kills the finalizer and returns the arguments passed to the constructor when it was created.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
```

(continues on next page)

(continued from previous page)

```
>>> f.detach()                                      #doctest:+ELLIPSIS
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

Unless you set the *atexit* attribute to *False*, a finalizer will be called when the program exits if it is still alive. For instance

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")  #doctest:+ELLIPSIS
<finalize object at ...; for 'Object' at ...>
>>> exit()                                          #doctest:+SKIP
obj dead or exiting
```

8.8.4 Comparing finalizers with `__del__()` methods

Suppose we want to create a class whose instances represent temporary directories. The directories should be deleted with their contents when the first of the following events occurs:

- the object is garbage collected,
- the object's `remove()` method is called, or
- the program exits.

We might try to implement the class using a `__del__()` method as follows:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

Starting with Python 3.4, `__del__()` methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to *None* during *interpreter shutdown*. So this code should work without any issues on CPython.

However, handling of `__del__()` methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded:

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

Note: If you create a finalizer object in a daemonic thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemonic thread *atexit.register()*, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.

8.9 types — Dynamic type creation and names for built-in types

Source code: Lib/types.py

This module defines utility functions to assist in dynamic creation of new types.

It also defines names for some object types that are used by the standard Python interpreter, but not exposed as builtins like *int* or *str* are.

Finally, it provides some additional type-related utility classes and functions that are not fundamental enough to be builtins.

8.9.1 Dynamic Type Creation

types.new_class(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Creates a class object dynamically using the appropriate metaclass.

The first three arguments are the components that make up a class definition header: the class name, the base classes (in order), the keyword arguments (such as `metaclass`).

The *exec_body* argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: ns`.

New in version 3.3.

types.**prepare_class**(*name*, *bases=()*, *kwds=None*)
Calculates the appropriate metaclass and creates the class namespace.

The arguments are the components that make up a class definition header: the class name, the base classes (in order) and the keyword arguments (such as `metaclass`).

The return value is a 3-tuple: `metaclass, namespace, kwds`

metaclass is the appropriate metaclass, *namespace* is the prepared class namespace and *kwds* is an updated copy of the passed in *kwds* argument with any `'metaclass'` entry removed. If no *kwds* argument is passed in, this will be an empty dict.

New in version 3.3.

Changed in version 3.6: The default value for the `namespace` element of the returned tuple has changed. Now an insertion-order-preserving mapping is used when the metaclass does not have a `__prepare__` method.

See also:

metaclasses Full details of the class creation process supported by these functions

PEP 3115 - Metaclasses in Python 3000 Introduced the `__prepare__` namespace hook

types.**resolve_bases**(*bases*)
Resolve MRO entries dynamically as specified by **PEP 560**.

This function looks for items in *bases* that are not instances of *type*, and returns a tuple where each such object that has an `__mro_entries__` method is replaced with an unpacked result of calling this method. If a *bases* item is an instance of *type*, or it doesn't have an `__mro_entries__` method, then it is included in the return tuple unchanged.

New in version 3.7.

See also:

PEP 560 - Core support for typing module and generic types

8.9.2 Standard Interpreter Types

This module provides names for many of the types that are required to implement a Python interpreter. It deliberately avoids including some of the types that arise only incidentally during processing such as the `listiterator` type.

Typical use of these names is for *isinstance()* or *issubclass()* checks.

Standard names are defined for the following types:

types.**FunctionType**
types.**LambdaType**
The type of user-defined functions and functions created by `lambda` expressions.

types.**GeneratorType**
The type of *generator*-iterator objects, created by generator functions.

types.**CoroutineType**
The type of *coroutine* objects, created by `async def` functions.

New in version 3.5.

types.**AsyncGeneratorType**
The type of *asynchronous generator*-iterator objects, created by asynchronous generator functions.

New in version 3.6.

types.CodeType
The type for code objects such as returned by *compile()*.

types.MethodType
The type of methods of user-defined class instances.

types.BuiltinFunctionType
types.BuiltinMethodType
The type of built-in functions like *len()* or *sys.exit()*, and methods of built-in classes. (Here, the term "built-in" means "written in C".)

types.WrapperDescriptorType
The type of methods of some built-in data types and base classes such as `object.__init__()` or `object.__lt__()`.

New in version 3.7.

types.MethodWrapperType
The type of *bound* methods of some built-in data types and base classes. For example it is the type of `object().__str__`.

New in version 3.7.

types.MethodDescriptorType
The type of methods of some built-in data types such as *str.join()*.

New in version 3.7.

types.ClassMethodDescriptorType
The type of *unbound* class methods of some built-in data types such as `dict.__dict__['fromkeys']`.

New in version 3.7.

class types.ModuleType(*name*, *doc=None*)
The type of *modules*. Constructor takes the name of the module to be created and optionally its *docstring*.

Note: Use *importlib.util.module_from_spec()* to create a new module if you wish to set the various import-controlled attributes.

__doc__
The *docstring* of the module. Defaults to `None`.

__loader__
The *loader* which loaded the module. Defaults to `None`.

Changed in version 3.4: Defaults to `None`. Previously the attribute was optional.

__name__
The name of the module.

__package__
Which *package* a module belongs to. If the module is top-level (i.e. not a part of any specific package) then the attribute should be set to `''`, else it should be set to the name of the package (which can be `__name__` if the module is a package itself). Defaults to `None`.

Changed in version 3.4: Defaults to `None`. Previously the attribute was optional.

class types.TracebackType(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)
The type of traceback objects such as found in `sys.exc_info()[2]`.

See the language reference for details of the available attributes and operations, and guidance on creating tracebacks dynamically.

types.FrameType
The type of frame objects such as found in tb.tb_frame if tb is a traceback object.

See the language reference for details of the available attributes and operations.

types.GetSetDescriptorType
The type of objects defined in extension modules with PyGetSetDef, such as FrameType.f_locals or array.array.typecode. This type is used as descriptor for object attributes; it has the same purpose as the *property* type, but for classes defined in extension modules.

types.MemberDescriptorType
The type of objects defined in extension modules with PyMemberDef, such as datetime.timedelta.days. This type is used as descriptor for simple C data members which use standard conversion functions; it has the same purpose as the *property* type, but for classes defined in extension modules.

CPython implementation detail: In other implementations of Python, this type may be identical to GetSetDescriptorType.

class types.MappingProxyType(*mapping*)
Read-only proxy of a mapping. It provides a dynamic view on the mapping's entries, which means that when the mapping changes, the view reflects these changes.

New in version 3.3.

key in proxy
Return True if the underlying mapping has a key *key*, else False.

proxy[key]
Return the item of the underlying mapping with key *key*. Raises a *KeyError* if *key* is not in the underlying mapping.

iter(proxy)
Return an iterator over the keys of the underlying mapping. This is a shortcut for iter(proxy.keys()).

len(proxy)
Return the number of items in the underlying mapping.

copy()
Return a shallow copy of the underlying mapping.

get(*key*[, *default*])
Return the value for *key* if *key* is in the underlying mapping, else *default*. If *default* is not given, it defaults to None, so that this method never raises a *KeyError*.

items()
Return a new view of the underlying mapping's items ((key, value) pairs).

keys()
Return a new view of the underlying mapping's keys.

values()
Return a new view of the underlying mapping's values.

8.9.3 Additional Utility Classes and Functions

class types.SimpleNamespace
A simple *object* subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike *object*, with SimpleNamespace you can add and remove attributes. If a SimpleNamespace object is initialized with keyword arguments, those are directly added to the underlying namespace.

The type is roughly equivalent to the following code:

```
class SimpleNamespace:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in keys)
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        return self.__dict__ == other.__dict__
```

`SimpleNamespace` may be useful as a replacement for `class NS: pass`. However, for a structured record type use *namedtuple()* instead.

New in version 3.3.

`types.DynamicClassAttribute`(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)
Route attribute access on a class to __getattr__.

This is a descriptor, used to define attributes that act differently when accessed through an instance and through a class. Instance access remains normal, but access to an attribute through a class will be routed to the class's __getattr__ method; this is done by raising AttributeError.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see Enum for an example).

New in version 3.4.

8.9.4 Coroutine Utility Functions

`types.coroutine`(*gen_func*)
This function transforms a *generator* function into a *coroutine function* which returns a generator-based coroutine. The generator-based coroutine is still a *generator iterator*, but is also considered to be a *coroutine* object and is *awaitable*. However, it may not necessarily implement the `__await__()` method.

If *gen_func* is a generator function, it will be modified in-place.

If *gen_func* is not a generator function, it will be wrapped. If it returns an instance of *collections.abc.Generator*, the instance will be wrapped in an *awaitable* proxy object. All other types of objects will be returned as is.

New in version 3.5.

8.10 copy — Shallow and deep copy operations

Source code: Lib/copy.py

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations (explained below).

Interface summary:

copy.copy(*x*)
Return a shallow copy of *x*.

copy.deepcopy(*x*[, *memo*])
Return a deep copy of *x*.

exception copy.error
Raised for module specific errors.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.
- A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Two problems often exist with deep copy operations that don't exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, such as data which is intended to be shared between copies.

The *deepcopy()* function avoids these problems by:

- keeping a `memo` dictionary of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, array, or any similar types. It does "copy" functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the *pickle* module.

Shallow copies of dictionaries can be made using *dict.copy()*, and of lists by assigning a slice of the entire list, for example, `copied_list = original_list[:]`.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module *pickle* for information on these methods. In fact, the *copy* module uses the registered pickle functions from the *copyreg* module.

In order for a class to define its own copy implementation, it can define special methods `__copy__()` and `__deepcopy__()`. The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the `memo` dictionary. If the `__deepcopy__()` implementation needs to make a deep copy of a component, it should call the *deepcopy()* function with the component as first argument and the memo dictionary as second argument.

See also:

Module *pickle* Discussion of the special methods used to support object state retrieval and restoration.

8.11 pprint — Data pretty printer

Source code: Lib/pprint.py

The *pprint* module provides a capability to "pretty-print" arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental

Python types, the representation may not be loadable. This may be the case if objects such as files, sockets or classes are included, as well as many other objects which are not representable as Python literals.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don't fit within the allowed width. Construct *PrettyPrinter* objects explicitly if you need to adjust the width constraint.

Dictionaries are sorted by key before the display is computed.

The *pprint* module defines one class:

class pprint.PrettyPrinter(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*)

Construct a *PrettyPrinter* instance. This constructor understands several keyword parameters. An output stream may be set using the *stream* keyword; the only method used on the stream object is the file protocol's `write()` method. If not specified, the *PrettyPrinter* adopts `sys.stdout`. The amount of indentation added for each recursive level is specified by *indent*; the default is one. Other values can cause output to look a little odd, but can make nesting easier to spot. The number of levels which may be printed is controlled by *depth*; if the data structure being printed is too deep, the next contained level is replaced by `...`. By default, there is no constraint on the depth of the objects being formatted. The desired output width is constrained using the *width* parameter; the default is 80 characters. If a structure cannot be formatted within the constrained width, a best effort will be made. If *compact* is false (the default) each item of a long sequence will be formatted on a separate line. If *compact* is true, as many items as will fit within the *width* will be formatted on each output line.

Changed in version 3.4: Added the *compact* parameter.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

The *pprint* module also provides several shortcut functions:

pprint.pformat(*object*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*)

Return the formatted representation of *object* as a string. *indent*, *width*, *depth* and *compact* will be passed to the *PrettyPrinter* constructor as formatting parameters.

Changed in version 3.4: Added the *compact* parameter.

pprint.pprint(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*)

Prints the formatted representation of *object* on *stream*, followed by a newline. If *stream* is `None`,

sys.stdout is used. This may be used in the interactive interpreter instead of the *print()* function for inspecting values (you can even reassign `print = pprint.pprint` for use within a scope). *indent*, *width*, *depth* and *compact* will be passed to the *PrettyPrinter* constructor as formatting parameters.

Changed in version 3.4: Added the *compact* parameter.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

pprint.**isreadable**(*object*)

Determine if the formatted representation of *object* is "readable," or can be used to reconstruct the value using *eval()*. This always returns `False` for recursive objects.

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)

Determine if *object* requires a recursive representation.

One more support function is also defined:

pprint.**saferepr**(*object*)

Return a string representation of *object*, protected against recursive data structures. If the representation of *object* exposes a recursive entry, the recursive reference will be represented as `<Recursion on typename with id=number>`. The representation is not otherwise formatted.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.11.1 PrettyPrinter Objects

PrettyPrinter instances have the following methods:

PrettyPrinter.**pformat**(*object*)

Return the formatted representation of *object*. This takes into account the options passed to the *PrettyPrinter* constructor.

PrettyPrinter.**pprint**(*object*)

Print the formatted representation of *object* on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new *PrettyPrinter* objects don't need to be created.

PrettyPrinter.**isreadable**(*object*)

Determine if the formatted representation of the object is "readable," or can be used to reconstruct the value using *eval()*. Note that this returns `False` for recursive objects. If the *depth* parameter of the *PrettyPrinter* is set and the object is deeper than allowed, this returns `False`.

`PrettyPrinter.isrecursive`(*object*)
Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the *saferepr()* implementation.

`PrettyPrinter.format`(*object*, *context*, *maxlevels*, *level*)
Returns three values: the formatted version of *object* as a string, a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the *id()* of objects that are part of the current presentation context (direct and indirect containers for *object* that are affecting the presentation) as the keys; if an object needs to be presented which is already represented in *context*, the third return value should be `True`. Recursive calls to the *format()* method should add additional entries for containers to this dictionary. The third argument, *maxlevels*, gives the requested limit to recursion; this will be `0` if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, *level*, gives the current level; recursive calls should be passed a value less than that of the current call.

8.11.2 Example

To demonstrate several uses of the *pprint()* function and its parameters, let's fetch information about a project from PyPI:

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/json') as resp:
...     project_info = json.load(resp)['info']
```

In its basic form, *pprint()* shows the whole object:

```
>>> pprint.pprint(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
```

(continues on next page)

(continued from previous page)

```
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

The result can be limited to a certain *depth* (ellipsis is used for deeper contents):

```
>>> pprint.pprint(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
```

(continues on next page)

(continued from previous page)

```
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

Additionally, maximum character *width* can be suggested. If a long object cannot be split, the specified width will be exceeded:

```
>>> pprint.pprint(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
                'New" section for the most recent version\n'
```

(continues on next page)

(continued from previous page)

```
               'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

8.12 reprlib — Alternate repr() implementation

Source code: Lib/reprlib.py

The *reprlib* module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

class reprlib.Repr
Class which provides formatting services useful in implementing functions similar to the built-in *repr()*; size limits for different object types are added to avoid the generation of representations which are excessively long.

reprlib.aRepr
This is an instance of *Repr* which is used to provide the *repr()* function described below. Changing the attributes of this object will affect the size limits used by *repr()* and the Python debugger.

reprlib.repr(*obj*)
This is the *repr()* method of `aRepr`. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

In addition to size-limiting tools, the module also provides a decorator for detecting recursive calls to `__repr__()` and substituting a placeholder string instead.

@reprlib.recursive_repr(*fillvalue="..."*)
Decorator for `__repr__()` methods to detect recursive calls within the same thread. If a recursive call is made, the *fillvalue* is returned, otherwise, the usual `__repr__()` call is made. For example:

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
```

(continues on next page)

(continued from previous page)

```
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

New in version 3.2.

8.12.1 Repr Objects

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

Repr.maxlevel
Depth limit on the creation of recursive representations. The default is 6.

Repr.maxdict
Repr.maxlist
Repr.maxtuple
Repr.maxset
Repr.maxfrozenset
Repr.maxdeque
Repr.maxarray
Limits on the number of entries represented for the named object type. The default is 4 for *maxdict*, 5 for *maxarray*, and 6 for the others.

Repr.maxlong
Maximum number of characters in the representation for an integer. Digits are dropped from the middle. The default is 40.

Repr.maxstring
Limit on the number of characters in the representation of the string. Note that the "normal" representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.

Repr.maxother
This limit is used to control the size of object types for which no specific formatting method is available on the *Repr* object. It is applied in a similar manner as *maxstring*. The default is 20.

Repr.repr(*obj*)
The equivalent to the built-in *repr()* that uses the formatting imposed by the instance.

Repr.repr1(*obj*, *level*)
Recursive implementation used by *repr()*. This uses the type of *obj* to determine which formatting method to call, passing it *obj* and *level*. The type-specific methods should call *repr1()* to perform recursive formatting, with `level - 1` for the value of *level* in the recursive call.

Repr.repr_TYPE(*obj*, *level*)
Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, **TYPE** is replaced by `'_'.join(type(obj).__name__.split())`. Dispatch to these methods is handled by *repr1()*. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`.

8.12.2 Subclassing Repr Objects

The use of dynamic dispatching by *Repr.repr1()* allows subclasses of *Repr* to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))         # prints '<stdin>'
```

8.13 enum — Support for enumerations

New in version 3.4.

Source code: Lib/enum.py

An enumeration is a set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over.

8.13.1 Module Contents

This module defines four enumeration classes that can be used to define unique sets of names and values: *Enum*, *IntEnum*, *Flag*, and *IntFlag*. It also defines one decorator, *unique()*, and one helper, *auto*.

class enum.Enum
Base class for creating enumerated constants. See section *Functional API* for an alternate construction syntax.

class enum.IntEnum
Base class for creating enumerated constants that are also subclasses of *int*.

class enum.IntFlag
Base class for creating enumerated constants that can be combined using the bitwise operators without losing their *IntFlag* membership. *IntFlag* members are also subclasses of *int*.

class enum.Flag
Base class for creating enumerated constants that can be combined using the bitwise operations without losing their *Flag* membership.

enum.unique()
Enum class decorator that ensures only one name is bound to any one value.

class enum.auto
Instances are replaced with an appropriate value for Enum members.

New in version 3.6: `Flag`, `IntFlag`, `auto`

8.13.2 Creating an Enum

Enumerations are created using the `class` syntax, which makes them easy to read and write. An alternative creation method is described in *Functional API*. To define an enumeration, subclass *Enum* as follows:

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...
```

Note: Enum member values

Member values can be anything: *int*, *str*, etc.. If the exact value is unimportant you may use *auto* instances and an appropriate value will be chosen for you. Care must be taken if you mix *auto* with other values.

Note: Nomenclature

- The class `Color` is an *enumeration* (or *enum*)
- The attributes `Color.RED`, `Color.GREEN`, etc., are *enumeration members* (or *enum members*) and are functionally constants.
- The enum members have *names* and *values* (the name of `Color.RED` is `RED`, the value of `Color.BLUE` is `3`, etc.)

Note: Even though we use the `class` syntax to create Enums, Enums are not normal Python classes. See *How are Enums different?* for more details.

Enumeration members have human readable string representations:

```
>>> print(Color.RED)
Color.RED
```

...while their `repr` has more information:

```
>>> print(repr(Color.RED))
<Color.RED: 1>
```

The *type* of an enumeration member is the enumeration it belongs to:

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
>>>
```

Enum members also have a property that contains just their item name:

```
>>> print(Color.RED.name)
RED
```

Enumerations support iteration, in definition order:

```
>>> class Shake(Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print(shake)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

Enumeration members are hashable, so they can be used in dictionaries and sets:

```
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```

8.13.3 Programmatic access to enumeration members and their attributes

Sometimes it's useful to access members in enumerations programmatically (i.e. situations where `Color.RED` won't do because the exact color is not known at program-writing time). `Enum` allows such access:

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

If you want to access enum members by *name*, use item access:

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

If you have an enum member and need its `name` or `value`:

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

8.13.4 Duplicating enum members and values

Having two enum members with the same name is invalid:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

However, two enum members are allowed to have the same value. Given two members A and B with the same value (and A defined first), B is an alias to A. By-value lookup of the value of A and B will return A. By-name lookup of B will also return A:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

Note: Attempting to create a member with the same name as an already defined attribute (another member, a method, etc.) or attempting to create an attribute with the same name as a member is not allowed.

8.13.5 Ensuring unique enumeration values

By default, enumerations allow multiple names as aliases for the same value. When this behavior isn't desired, the following decorator can be used to ensure each value is used only once in the enumeration:

@enum.unique

A `class` decorator specifically for enumerations. It searches an enumeration's `__members__` gathering any aliases it finds; if any are found *ValueError* is raised with the details:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

8.13.6 Using automatic values

If the exact value is unimportant you can use *auto*:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

The values are chosen by `_generate_next_value_()`, which can be overridden:

```
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>, <Ordinal.
↪WEST: 'WEST'>]
```

Note: The goal of the default `_generate_next_value_()` methods is to provide the next *int* in sequence with the last *int* provided, but the way it does this is an implementation detail and may change.

8.13.7 Iteration

Iterating over the members of an enum does not provide the aliases:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

The special attribute `__members__` is an ordered dictionary mapping names to members. It includes all names defined in the enumeration, including the aliases:

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

The `__members__` attribute can be used for detailed programmatic access to the enumeration members. For example, finding all the aliases:

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['ALIAS_FOR_SQUARE']
```

8.13.8 Comparisons

Enumeration members are compared by identity:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

Ordered comparisons between enumeration values are *not* supported. Enum members are not integers (but see *IntEnum* below):

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

Equality comparisons are defined though:

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

Comparisons against non-enumeration values will always compare not equal (again, *IntEnum* was explicitly designed to behave differently, see below):

```
>>> Color.BLUE == 2
False
```

8.13.9 Allowed members and attributes of enumerations

The examples above use integers for enumeration values. Using integers is short and handy (and provided by default by the *Functional API*), but not strictly enforced. In the vast majority of use-cases, one doesn't care what the actual value of an enumeration is. But if the value *is* important, enumerations can have arbitrary values.

Enumerations are Python classes, and can have methods and special methods as usual. If we have this enumeration:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
```

(continues on next page)

(continued from previous page)

```
...         # self is the member here
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
```

Then:

```
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'
```

The rules for what is allowed are as follows: names that start and end with a single underscore are reserved by enum and cannot be used; all other attributes defined within an enumeration will become members of this enumeration, with the exception of special methods (`__str__()`, `__add__()`, etc.), descriptors (methods are also descriptors), and variable names listed in `_ignore_`.

Note: if your enumeration defines `__new__()` and/or `__init__()` then whatever value(s) were given to the enum member will be passed into those methods. See *Planet* for an example.

8.13.10 Restricted Enum subclassing

A new *Enum* class must have one base Enum class, up to one concrete data type, and as many *object*-based mixin classes as needed. The order of these base classes is:

```
def EnumName([mix-in, ...,] [data-type,] base-enum):
    pass
```

Also, subclassing an enumeration is allowed only if the enumeration does not define any members. So this is forbidden:

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

But this is allowed:

```
>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
```

(continues on next page)

(continued from previous page)

```
>>> class Bar(Foo):
...     HAPPY = 1
...     SAD = 2
...
```

Allowing subclassing of enums that define members would lead to a violation of some important invariants of types and instances. On the other hand, it makes sense to allow sharing some common behavior between a group of enumerations. (See *OrderedEnum* for an example.)

8.13.11 Pickling

Enumerations can be pickled and unpickled:

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

The usual restrictions for pickling apply: picklable enums must be defined in the top level of a module, since unpickling requires them to be importable from that module.

Note: With pickle protocol version 4 it is possible to easily pickle enums nested in other classes.

It is possible to modify how Enum members are pickled/unpickled by defining *__reduce_ex__()* in the enumeration class.

8.13.12 Functional API

The *Enum* class is callable, providing the following functional API:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

The semantics of this API resemble *namedtuple*. The first argument of the call to *Enum* is the name of the enumeration.

The second argument is the *source* of enumeration member names. It can be a whitespace-separated string of names, a sequence of names, a sequence of 2-tuples with key/value pairs, or a mapping (e.g. dictionary) of names to values. The last two options enable assigning arbitrary values to enumerations; the others auto-assign increasing integers starting with 1 (use the `start` parameter to specify a different starting value). A new class derived from *Enum* is returned. In other words, the above assignment to `Animal` is equivalent to:

```
>>> class Animal(Enum):
...     ANT = 1
```

(continues on next page)

(continued from previous page)

```
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

The reason for defaulting to `1` as the starting number and not `0` is that `0` is `False` in a boolean sense, but enum members all evaluate to `True`.

Pickling enums created with the functional API can be tricky as frame stack implementation details are used to try and figure out which module the enumeration is being created in (e.g. it will fail if you use a utility function in separate module, and also may not work on IronPython or Jython). The solution is to specify the module name explicitly as follows:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

Warning: If `module` is not supplied, and Enum cannot determine what it is, the new Enum members will not be unpicklable; to keep errors closer to the source, pickling will be disabled.

The new pickle protocol 4 also, in some circumstances, relies on *__qualname__* being set to the location where pickle will be able to find the class. For example, if the class was made available in class SomeData in the global scope:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal')
```

The complete signature is:

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...', type=<mixed-in␣
↪class>, start=1)
```

value What the new Enum class will record as its name.

names The Enum members. This can be a whitespace or comma separated string (values will start at 1 unless otherwise specified):

```
'RED GREEN BLUE' | 'RED,GREEN,BLUE' | 'RED, GREEN, BLUE'
```

or an iterator of names:

```
['RED', 'GREEN', 'BLUE']
```

or an iterator of (name, value) pairs:

```
[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]
```

or a mapping:

```
{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}
```

module name of module where new Enum class can be found.

qualname where in module new Enum class can be found.

type type to mix in to new Enum class.

start number to start counting at if only names are passed in.

Changed in version 3.5: The *start* parameter was added.

8.13.13 Derived Enumerations

IntEnum

The first variation of *Enum* that is provided is also a subclass of *int*. Members of an *IntEnum* can be compared to integers; by extension, integer enumerations of different types can also be compared to each other:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

However, they still can't be compared to standard *Enum* enumerations:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

IntEnum values behave like integers in other ways you'd expect:

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

IntFlag

The next variation of *Enum* provided, *IntFlag*, is also based on *int*. The difference being *IntFlag* members can be combined using the bitwise operators (&, |, ^, ~) and the result is still an *IntFlag* member. However, as the name implies, *IntFlag* members also subclass *int* and can be used wherever an *int* is used. Any operation on an *IntFlag* member besides the bit-wise operations will lose the *IntFlag* membership.

New in version 3.6.

Sample *IntFlag* class:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

It is also possible to name the combinations:

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

Another important difference between *IntFlag* and *Enum* is that if no flags are set (the value is 0), its boolean evaluation is *False*:

```
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
```

Because *IntFlag* members are also subclasses of *int* they can be combined with them:

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

Flag

The last variation is *Flag*. Like *IntFlag*, *Flag* members can be combined using the bitwise operators (&, |, ^, ~). Unlike *IntFlag*, they cannot be combined with, nor compared against, any other *Flag* enumeration, nor *int*. While it is possible to specify the values directly it is recommended to use *auto* as the value and let *Flag* select an appropriate value.

New in version 3.6.

Like *IntFlag*, if a combination of *Flag* members results in no flags being set, the boolean evaluation is *False*:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
```

(continues on next page)

(continued from previous page)

```
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

Individual flags should have values that are powers of two (1, 2, 4, 8, ...), while combinations of flags won't:

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

Giving a name to the "no flags set" condition does not change its boolean value:

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

Note: For the majority of new code, *Enum* and *Flag* are strongly recommended, since *IntEnum* and *IntFlag* break some semantic promises of an enumeration (by being comparable to integers, and thus by transitivity to other unrelated enumerations). *IntEnum* and *IntFlag* should be used only in cases where *Enum* and *Flag* will not do; for example, when integer constants are replaced with enumerations, or for interoperability with other systems.

Others

While *IntEnum* is part of the *enum* module, it would be very simple to implement independently:

```
class IntEnum(int, Enum):
    pass
```

This demonstrates how similar derived enumerations can be defined; for example a `StrEnum` that mixes in *str* instead of *int*.

Some rules:

1. When subclassing *Enum*, mix-in types must appear before *Enum* itself in the sequence of bases, as in the *IntEnum* example above.

2. While *Enum* can have members of any type, once you mix in an additional type, all the members must have values of that type, e.g. *int* above. This restriction does not apply to mix-ins which only add methods and don't specify another data type such as *int* or *str*.
3. When another data type is mixed in, the `value` attribute is *not the same* as the enum member itself, although it is equivalent and will compare equal.
4. %-style formatting: *%s* and *%r* call the *Enum* class's `__str__()` and `__repr__()` respectively; other codes (such as *%i* or *%h* for IntEnum) treat the enum member as its mixed-in type.
5. Formatted string literals, *str.format()*, and *format()* will use the mixed-in type's `__format__()`. If the *Enum* class's *str()* or *repr()* is desired, use the *!s* or *!r* format codes.

8.13.14 Interesting examples

While *Enum*, *IntEnum*, *IntFlag*, and *Flag* are expected to cover the majority of use-cases, they cannot cover them all. Here are recipes for some different types of enumerations that can be used directly, or as examples for creating one's own.

Omitting values

In many use-cases one doesn't care what the actual value of an enumeration is. There are several ways to define this type of simple enumeration:

- use instances of *auto* for the value
- use instances of *object* as the value
- use a descriptive string as the value
- use a tuple as the value and a custom `__new__()` to replace the tuple with an *int* value

Using any of these methods signifies to the user that these values are not important, and also enables one to add, remove, or reorder members without having to renumber the remaining members.

Whichever method you choose, you should provide a *repr()* that also hides the (unimportant) value:

```
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
...
```

Using `auto`

Using *auto* would look like:

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

Using `object`

Using *`object`* would look like:

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

Using a descriptive string

Using a string as the value would look like:

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

Using a custom `__new__()`

Using an auto-numbering `__new__()` would look like:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

Note: The `__new__()` method, if defined, is used during creation of the Enum members; it is then replaced by Enum's `__new__()` which is used after class creation for lookup of existing members.

OrderedEnum

An ordered enumeration that is not based on *IntEnum* and so maintains the normal *Enum* invariants (such as not being comparable to other enumerations):

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True
```

DuplicateFreeEnum

Raises an error if a duplicate member name is found instead of creating an alias:

```
>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum:  %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREEN'
```

Note: This is a useful example for subclassing Enum to add or change other behaviors as well as disallowing aliases. If the only desired change is disallowing aliases, the *unique()* decorator can be used instead.

Planet

If `__new__()` or `__init__()` is defined the value of the enum member will be passed to those methods:

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass       # in kilograms
...         self.radius = radius   # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

TimePeriod

An example to show the `_ignore_` attribute in use:

```
>>> from datetime import timedelta
>>> class Period(timedelta, Enum):
...     "different lengths of time"
...     _ignore_ = 'Period i'
...     Period = vars()
...     for i in range(367):
...         Period['day_%d' % i] = i
...
>>> list(Period)[:2]
[<Period.day_0: datetime.timedelta(0)>, <Period.day_1: datetime.timedelta(days=1)>]
```

(continues on next page)

(continued from previous page)

```
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: datetime.
↪timedelta(days=366)>]
```

8.13.15 How are Enums different?

Enums have a custom metaclass that affects many aspects of both derived Enum classes and their instances (members).

Enum Classes

The `EnumMeta` metaclass is responsible for providing the `__contains__()`, `__dir__()`, `__iter__()` and other methods that allow one to do things with an *Enum* class that fail on a typical class, such as *list(Color)* or *some_enum_var in Color*. `EnumMeta` is responsible for ensuring that various other methods on the final *Enum* class are correct (such as `__new__()`, *__getnewargs__()*, `__str__()` and `__repr__()`).

Enum Members (aka instances)

The most interesting thing about Enum members is that they are singletons. `EnumMeta` creates them all while it is creating the *Enum* class itself, and then puts a custom `__new__()` in place to ensure that no new ones are ever instantiated by returning only the existing member instances.

Finer Points

Supported `__dunder__` names

`__members__` is an `OrderedDict` of `member_name:member` items. It is only available on the class.

`__new__()`, if specified, must create and return the enum members; it is also a very good idea to set the member's `_value_` appropriately. Once all the members are created it is no longer used.

Supported `_sunder_` names

- `_name_` – name of the member
- `_value_` – value of the member; can be set / modified in `__new__`
- `_missing_` – a lookup function used when a value is not found; may be overridden
- `_ignore_` – a list of names, either as a *list()* or a *str()*, that will not be transformed into members, and will be removed from the final class
- `_order_` – used in Python 2/3 code to ensure member order is consistent (class attribute, removed during class creation)
- `_generate_next_value_` – used by the *Functional API* and by *auto* to get an appropriate value for an enum member; may be overridden

New in version 3.6: `_missing_`, `_order_`, `_generate_next_value_`

New in version 3.7: `_ignore_`

To help keep Python 2 / Python 3 code in sync an `_order_` attribute can be provided. It will be checked against the actual order of the enumeration and raise an error if the two do not match:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

Note: In Python 2 code the `_order_` attribute is necessary as definition order is lost before it can be recorded.

`Enum` member type

Enum members are instances of their *Enum* class, and are normally accessed as `EnumClass.member`. Under certain circumstances they can also be accessed as `EnumClass.member.member`, but you should never do this as that lookup may fail or, worse, return something besides the *Enum* member you are looking for (this is another good reason to use all-uppercase names for members):

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

Changed in version 3.5.

Boolean value of `Enum` classes and members

Enum members that are mixed with non-*Enum* types (such as *int*, *str*, etc.) are evaluated according to the mixed-in type's rules; otherwise, all members evaluate as *True*. To make your own Enum's boolean evaluation depend on the member's value add the following to your class:

```
def __bool__(self):
    return bool(self.value)
```

Enum classes always evaluate as *True*.

`Enum` classes with methods

If you give your *Enum* subclass extra methods, like the *Planet* class above, those methods will show up in a *dir()* of the member, but not of the class:

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__class_
↪_', '__doc__', '__members__', '__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

Combining members of `Flag`

If a combination of Flag members is not named, the *repr()* will include all named flags and all named combinations of flags that are in the value:

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3)  # named combination
<Color.YELLOW: 3>
>>> Color(7)      # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```