

COURSE

Date and Time in Practice

10 Chapters

Philip Osztromok

Generated with Claude

DATE AND TIME IN PRACTICE

Contents

Chapter 1	Dates and Deltas	2
Chapter 2	Formatting and Parsing	7
Chapter 3	Naive vs Aware	14
Chapter 4	Time Zones	20
Chapter 5	Daylight Saving Traps	26
Chapter 6	Calendar Arithmetic	32
Chapter 7	Durations and Intervals	39
Chapter 8	Timestamps and Epochs	47
Chapter 9	Storing and Exchanging	54
Chapter 10	Capstone Toolkit	62

Date and Time in Practice

Chapter 1 · Dates and Deltas

Dates and times look simple until a program has to compare, add, store or display them. Months have different lengths, years sometimes have an extra day, and the same moment is written differently in different countries. This course is practical: each chapter takes one idea, runs it, and ends with a rule you can reuse. All examples use Python 3 and its built-in `datetime` module, so there is nothing to install.

> **WHAT WAS RUN**
>
> Every snippet in this chapter was executed, and the output shown in the comments is what Python printed. Run them yourself in a file or the interactive prompt.

Four Types, Four Jobs

The module gives you separate types for separate ideas. Choosing the right one up front avoids a lot of later conversion.

TYPE	HOLDS	EXAMPLE
date	a calendar day, no time of day	`date(2026, 11, 2)`
time	a time of day, no calendar day	`time(14, 30, 5)`
datetime	a day and a time together	`datetime(2026, 11, 2, 14, 30, 5)`
timedelta	a length of time, not a point in time	`timedelta(hours=1, minutes=30)`

```
from datetime import date, time, datetime, timedelta

d = date(2026, 11, 2)
print(d)                          # 2026-11-02
print(d.year, d.month, d.day)     # 2026 11 2
print(d.weekday(), d.isoweekday())  # 0 1   (Monday is 0 here, 1 in the ISO numbering)
print(d.strftime("%A"))           # Monday
```

```
t = time(14, 30, 5)
print(t)                            # 14:30:05

dt = datetime(2026, 11, 2, 14, 30, 5)
print(dt)                           # 2026-11-02 14:30:05
print(repr(dt))                     # datetime.datetime(2026, 11, 2, 14, 30, 5)
print(datetime.combine(d, t))       # 2026-11-02 14:30:05
```

Invalid values are rejected immediately rather than silently rolling over, which is a useful safety net:

```
try:
    date(2026, 2, 30)
except ValueError as e:
    print("ValueError:", e)   # ValueError: day 30 must be in range 1..28 for month 2 in
```

Subtracting Gives a timedelta

Two points in time subtract to give a **length** of time. That length is a `timedelta`, and you can add it to or subtract it from a date or datetime to move forward or back.

```
a = date(2026, 11, 2)
b = date(2027, 1, 1)

gap = b - a
print(gap)                          # 60 days, 0:00:00
print(gap.days)                     # 60
print(type(gap))                    # <class 'datetime.timedelta'>

print(a + timedelta(days=30))       # 2026-12-02
print(a - timedelta(weeks=2))       # 2026-10-19
print(a < b, a == date(2026, 11, 2))   # True True

td = timedelta(hours=1, minutes=30)
print(td, td.total_seconds())       # 1:30:00 5400.0
```

```
print(timedelta(seconds=-90))           # -1 day, 23:58:30
print(datetime(2026, 11, 2, 14, 30, 5) + timedelta(hours=10))
# 2026-11-03 00:30:05
```

Two details catch people out. First, a `timedelta` stores only days, seconds and microseconds, so there is no "months" or "years" argument: a month is not a fixed length. Second, a negative delta prints oddly (`-1 day, 23:58:30` is minus 90 seconds), because the days part is negative and the rest is positive. Use `total_seconds()` when you want one plain number.

> **.SECONDS IS NOT THE TOTAL**
>
> The `.seconds` attribute of a timedelta is only the seconds left over after whole days are removed. For a span of a day or more it is the wrong thing to read. Use `.total_seconds()`.

Turning Dates into Text and Back

`strftime` ("string from time") formats a date as text using percent codes. `strptime` ("string parsed to time") does the reverse: you give it the text and the format that describes it. The two use the same codes.

CODE	MEANING	EXAMPLE
`%Y` `%m` `%d`	4-digit year; month 01-12; day 01-31	`2026` `11` `02`
`%H` `%M` `%S`	hour (24-hour); minute; second	`14` `30` `05`
`%A` `%B`	full weekday name; full month name	`Monday` `November`

```
dt = datetime(2026, 11, 2, 14, 30, 5)
print(dt.strftime("%d/%m/%Y %H:%M"))     # 02/11/2026 14:30
print(dt.strftime("%A %d %B %Y"))        # Monday 02 November 2026
print(dt.isoformat())                      # 2026-11-02T14:30:05

p = datetime.strptime("2026-11-02 14:30", "%Y-%m-%d %H:%M")
print(p, p == datetime(2026, 11, 2, 14, 30))   # 2026-11-02 14:30:00 True

print(date.fromisoformat("2026-11-02"))              # 2026-11-02
```

```
print(datetime.fromisoformat("2026-11-02T14:30:05"))   # 2026-11-02 14:30:05
```

The format must describe the text exactly, and the same text means different dates under different formats. Both lines below parse successfully and give a different day:

```
print(datetime.strptime("02/11/2026", "%d/%m/%Y").date())   # 2026-11-02  (2 November)
print(datetime.strptime("11/02/2026", "%m/%d/%Y").date())   # 2026-11-02  (also 2 Novemb

try:
    datetime.strptime("2026-13-02", "%Y-%m-%d")
except ValueError as e:
    print("ValueError:", e)   # ValueError: time data '2026-13-02' does not match format
```

So the text `02/11/2026` and the text `11/02/2026` both mean 2 November, but only because each was given the matching format. Handed the wrong format, the first would be read as 11 February. This is why you should store and exchange dates in the unambiguous ISO form (`2026-11-02`) and keep the day-first or month-first forms for display only.

A HABIT WORTH FORMING

Keep the right type for the job: `date` when there is no time of day, `timedelta` for lengths. Subtract to measure, add a timedelta to move. Prefer ISO text when storing, and catch `ValueError` when parsing anything a person typed.

Challenges

EXERCISE 1

Given `born = date(1990, 5, 17)` and `today = date(2026, 11, 2)`, work out the person's age in whole years and the total number of days they have lived. Take care that the age is right when the birthday has not yet happened this year. Hint: tuples compare element by element, so `(today.month, today.day) < (born.month, born.day)` is a neat test.

View solution

EXERCISE 2

A shift starts at 09:15 and ends at 17:50 on the same day (use `datetime` objects for 2026-11-02). Print the length of the shift as a timedelta, as a decimal number of hours, and as whole hours plus remaining minutes. Predict each output before you run it.

View solution

EXERCISE 3

Write `next_weekday(d, target)` that returns the next date strictly after `d` that falls on weekday number `target` (Monday is 0, Sunday is 6). If `d` is already that weekday, return the date a week later. Test it with `date(2026, 11, 2)`, which is a Monday, for a Monday, a Friday and a Sunday, and print the Friday's name with `strftime`.

View solution

QUICK REFERENCE

- **Types:** `date`, `time`, `datetime` are points; `timedelta` is a length
- **Subtract** two points to get a timedelta; **add** a timedelta to move a point
- **Read a delta** with `.days` for whole days, `.total_seconds()` for everything
- **Format:** `strftime(fmt)` to text, `strptime(text, fmt)` from text
- **ISO:** `isoformat()` and `fromisoformat()` for unambiguous storage
- **Bad input** raises `ValueError`; catch it when parsing user text

COMING NEXT

Date and Time in Practice 2 looks at formatting and parsing in more depth: the full set of format codes, ambiguous inputs, and how to handle messy dates safely.

Date and Time in Practice

Chapter 2 · Formatting and Parsing

Date and Time in Practice 1 introduced `strftime` and `strptime`. This chapter goes deeper: the format codes you will actually use, the ways date text can be ambiguous, the stricter and safer `fromisoformat`, and a pattern for handling input that does not parse. The examples use Python 3.14, and every snippet was run; the output shown in the comments is what Python printed.

The Format Codes Worth Knowing

All the examples below format the same moment, `datetime(2026, 3, 5, 9, 7, 4, 123456)`, which is a Thursday.

CODE	MEANING	OUTPUT
`%y`	2-digit year	`26`
`%a` `%b`	short weekday; short month	`Thu` `Mar`
`%I` `%p`	hour 01-12; AM or PM	`09` `AM`
`%f`	microseconds, 6 digits	`123456`
`%j`	day of the year, 001-366	`064`
`%w` `%u`	weekday number: Sunday is 0; Monday is 1	`4` `4`
`%V` `%G`	ISO week number; ISO year of that week	`10` `2026`
`%%`	a literal percent sign	`%`

```
from datetime import date, datetime

dt = datetime(2026, 3, 5, 9, 7, 4, 123456)

print(dt.strftime("%d/%m/%y"))          # 05/03/26
print(dt.strftime("%a %b %d"))         # Thu Mar 05
print(dt.strftime("%I:%M %p"))         # 09:07 AM
print(dt.strftime("%H:%M:%S.%f"))     # 09:07:04.123456
```

```
print(dt.strftime("%G-W%V-%u"))       # 2026-W10-4
print(dt.strftime("%j"))               # 064

# An f-string accepts the same codes after a colon
print(f"{dt:%d %B %Y}")                # 05 March 2026
print(dt.isoformat(timespec="minutes"))              # 2026-03-05T09:07
print(dt.isoformat(sep=" ", timespec="seconds"))     # 2026-03-05 09:07:04
```

NAMES DEPEND ON THE LOCALE

`%A`, `%B`, `%a`, `%b` and `%p` produce names from the machine's current locale. The output above is English because the program ran in an English locale; on a machine set to another language the same code gives different words, and text in that language will not parse on an English machine. Names are fine for display, but do not use them in stored or exchanged data.

Parsing with strptime

The format must describe the text. Names, 12-hour clocks and two-digit years all parse, but each has a catch worth knowing.

```
print(datetime.strptime("5 March 2026", "%d %B %Y"))   # 2026-03-05 00:00:00
print(datetime.strptime("5 Mar 26", "%d %b %y"))        # 2026-03-05 00:00:00
print(datetime.strptime("3:07 PM", "%I:%M %p"))        # 1900-01-01 15:07:00
```

Three things are visible there. Fields missing from the format are filled with defaults, so a time with no date becomes 1 January 1900, a placeholder you should ignore (take `.time()` from it). Text with no time of day gives midnight. And a two-digit year has to be guessed:

```
print(datetime.strptime("69", "%y").year)   # 1969
print(datetime.strptime("68", "%y").year)   # 2068
```

Python maps 69 to 99 onto the 1900s and 00 to 68 onto the 2000s. A birth year written `68` becomes 2068, which is in the future. Prefer four-digit years whenever you have a choice.

Ambiguous Text

Parsing succeeds whenever the text fits the format, even if the format is the wrong one. The same text can be valid under two formats and mean two different days:

```
text = "1/2/26"
print(datetime.strptime(text, "%d/%m/%y").date())   # 2026-02-01   (day first: 1 February
print(datetime.strptime(text, "%m/%d/%y").date())   # 2026-01-02   (month first: 2 Janua
```

Nothing in the code can tell you which was meant. Only the context can: where the data came from, or whether another date in the same file has a first number above 12. When you must accept day-first or month-first text, make the choice explicit and documented, or ask the person. When you can choose, switch to ISO `YYYY-MM-DD`, where the order is fixed.

TEXT	DAY FIRST	MONTH FIRST	VERDICT
`03/04/2026`	3 April	4 March	ambiguous
`13/04/2026`	13 April	invalid	only one reading works
`2026-04-03`	always year, month, day		unambiguous

Strict and Lenient Parsing

`strptime` is stricter than you might expect about extra text, and more relaxed than you might expect about padding:

```
# Leading zeros are optional when parsing
print(datetime.strptime("2026-3-5", "%Y-%m-%d"))      # 2026-03-05 00:00:00

# Anything left over is an error
```

```
try:
    datetime.strptime("2026-03-05 extra", "%Y-%m-%d")
except ValueError as e:
    print("ValueError:", e)            # ValueError: unconverted data remains:  extra

# So is a date that does not exist
try:
    datetime.strptime("2026-02-29", "%Y-%m-%d")
except ValueError as e:
    print("ValueError:", e)            # ValueError: day 29 must be in range 1..28 for m

# Stray spaces around the text are also an error, so strip first
print(datetime.strptime("  2026-03-05".strip(), "%Y-%m-%d"))   # 2026-03-05 00:00:00
```

fromisoformat: the Safe Default

When the text is ISO 8601, use `fromisoformat`. It needs no format string, so there is nothing to get wrong. From Python 3.11 it accepts most ISO forms, not only the one `isoformat()` writes. Note the difference between `date` and `datetime`:

```
for s in ["2026-03-05", "2026-03-05T09:07:04", "2026-03-05 09:07:04", "20260305", "2026-
    for fn in (date.fromisoformat, datetime.fromisoformat):
        try:
            print(fn.__qualname__, s, "->", fn(s))
        except ValueError:
            print(fn.__qualname__, s, "-> ValueError")
# date.fromisoformat 2026-03-05 -> 2026-03-05
# datetime.fromisoformat 2026-03-05 -> 2026-03-05 00:00:00
# date.fromisoformat 2026-03-05T09:07:04 -> ValueError
# datetime.fromisoformat 2026-03-05T09:07:04 -> 2026-03-05 09:07:04
# date.fromisoformat 2026-03-05 09:07:04 -> ValueError
# datetime.fromisoformat 2026-03-05 09:07:04 -> 2026-03-05 09:07:04
# date.fromisoformat 20260305 -> 2026-03-05
# datetime.fromisoformat 20260305 -> 2026-03-05 00:00:00
# date.fromisoformat 2026-W10-4 -> 2026-03-05
# datetime.fromisoformat 2026-W10-4 -> 2026-03-05 00:00:00
```

A date-only function rejects text that carries a time, so use `datetime.fromisoformat(s).date()` if the input might have either. The ISO week form `2026-W10-4` (year, week 10, day 4) is understood too. Versions of Python before 3.11 were stricter, so check your version if a form that works here fails elsewhere.

Handling Bad Input

Anything a person typed, or another system sent, can fail to parse. The pattern is to try, catch `ValueError`, and decide what failure means for your program: reject, ask again, or try another format. Do not use a bare `except`, which would hide real bugs.

```
def parse_date(text, formats):
    """Return the first format that fits as a date, or None."""
    for f in formats:
        try:
            return datetime.strptime(text.strip(), f).date()
        except ValueError:
            pass
    return None

formats = ["%Y-%m-%d", "%d %b %Y", "%B %d, %Y"]
for t in ["2026-11-02", "2 Nov 2026", "November 2, 2026", "2/11/2026", "2026-02-30"]:
    print(repr(t), parse_date(t, formats))
# '2026-11-02' 2026-11-02
# '2 Nov 2026' 2026-11-02
# 'November 2, 2026' 2026-11-02
# '2/11/2026' None
# '2026-02-30' None
```

The list of formats is a deliberate choice: `2/11/2026` is rejected because no listed format is day-first with slashes. Accepting fewer formats is safer than guessing. Order matters when two formats could fit the same text, since the first match wins.

> **A HABIT WORTH FORMING**

Display in whatever style suits the reader, but store and exchange ISO text. Parse with `fromisoformat` when the input is ISO, and with an explicit, short list of formats otherwise. Treat a parse failure as normal data, not as a surprise.

Challenges

EXERCISE 1

Write `readings(text)` that tries the text as both day-first (`%d/%m/%Y`) and month-first (`%m/%d/%Y`) and returns the sorted list of distinct dates that are valid. Then print, for each of `03/04/2026`, `13/04/2026`, `05/05/2026` and `31/02/2026`, the dates in ISO form and a verdict of ambiguous, clear or invalid. Decide what you expect for `05/05/2026` before running it.

View solution

EXERCISE 2

Write `to_iso(text)` that converts text such as `05 Mar 2026 09:07 PM` into ISO form with minutes precision (`2026-03-05T21:07`), and returns `"invalid: "` followed by the text if it does not parse. Test it with that example, with `12:00 AM` and `12:00 PM` on the same day, with `13:00 PM`, and with `31 Apr 2026 01:00 AM`. Predict what the two twelve o'clock cases give.

View solution

EXERCISE 3

Using `datetime(2026, 3, 5, 9, 7, 4)`, build a single line of text in the form `Thu 05 Mar 2026, week 10, day 064` with one `strftime` call. Then check your answer by parsing the date part back with `strptime` and confirming it equals the original date. Hint: one code gives the ISO week and one gives the day of the year; a comma is an ordinary character.

View solution

QUICK REFERENCE

- **Common codes:** `%y` short year, `%a %b` short names, `%I %p` 12-hour clock, `%f` microseconds, `%j` day of year, `%V` ISO week
- **Names and AM/PM** follow the locale: display only, never store
- **Missing fields** default (time of day to midnight, a bare time to 1 January 1900)
- **Two-digit years:** 69-99 become 19xx, 00-68 become 20xx; prefer four digits
- **Ambiguity:** `1/2/26` fits both day-first and month-first; only context decides
- **strptime errors:** leftover text and impossible dates raise `ValueError`; strip spaces first
- **ISO input:** `fromisoformat`; use `datetime.fromisoformat(s).date()` if a time may be present

COMING NEXT

Date and Time in Practice 3 looks at the difference between naive and aware datetimes: what attaching a time zone really does, why mixing the two fails, and how comparing works.

Date and Time in Practice

Chapter 3 · Naive vs Aware

Every `datetime` you have made so far was *naive*: a calendar date and a clock reading with nothing to say where in the world that reading applies. This chapter covers the other kind, *aware* datetimes, which carry a time zone, and the rules for using the two together. The examples use Python 3.14, and every snippet was run; the output shown in the comments is what Python printed.

What "Aware" Means

A datetime is aware when its `tzinfo` is set and that object can report an offset from UTC. The simplest tools are in the `datetime` module itself: `timezone.utc`, and `timezone(timedelta(...))` for a fixed offset.

```
from datetime import datetime, timezone, timedelta

naive = datetime(2026, 3, 5, 9, 0)
print(naive.tzinfo, naive.utcoffset())          # None None
print(naive.isoformat())                        # 2026-03-05T09:00:00

utc = datetime(2026, 3, 5, 9, 0, tzinfo=timezone.utc)
print(utc)                                      # 2026-03-05 09:00:00+00:00
print(utc.utcoffset())                          # 0:00:00
print(utc.isoformat())                          # 2026-03-05T09:00:00+00:00

ist = timezone(timedelta(hours=5, minutes=30), "IST")
t = datetime(2026, 3, 5, 14, 30, tzinfo=ist)
print(t, t.tzname(), t.utcoffset())             # 2026-03-05 14:30:00+05:30 IST 5:30:00
```

A naive value is one clock reading; an aware value names one exact moment in history, because the offset pins the reading to UTC. That is the whole difference, and everything below follows from it.

	NAIVE	AWARE
`tzinfo`	`None`	set
`utcoffset()`	`None`	a `timedelta`
Names an exact moment?	no, only a reading	yes
ISO text ends with	nothing	`+00:00`, `+05:30`, `Z` …

THE RELIABLE TEST

Do not decide by `tzinfo` alone. Python's own definition is that a value is aware only if `tzinfo` is not `None` *and* `utcoffset()` does not return `None`. A small helper keeps that in one place.

```
def is_aware(d):
    return d.tzinfo is not None and d.utcoffset() is not None

print(is_aware(naive), is_aware(utc))           # False True
```

Reading the Offset from Text

`fromisoformat` produces whichever kind the text describes. An offset or a trailing `Z` gives an aware value (`Z` is understood from Python 3.11); no offset gives a naive one.

```
for s in ["2026-03-05T09:00:00", "2026-03-05T09:00:00Z",
          "2026-03-05T09:00:00+00:00", "2026-03-05T14:30:00+05:30"]:
    d = datetime.fromisoformat(s)
    print(s, "->", repr(d.tzinfo), d.utcoffset())
# 2026-03-05T09:00:00 -> None None
# 2026-03-05T09:00:00Z -> datetime.timezone.utc 0:00:00
# 2026-03-05T09:00:00+00:00 -> datetime.timezone.utc 0:00:00
# 2026-03-05T14:30:00+05:30 -> datetime.timezone(datetime.timedelta(seconds=19800)) 5:30
```

Two Ways to Change a Zone, One of Them Wrong

There are two operations that look alike and do opposite things.

- `replace(tzinfo=...)` keeps the clock reading and swaps the label. It names a *different* moment.
- `astimezone(...)` keeps the moment and rewrites the reading for another zone. It names the *same* moment.

```
t = datetime(2026, 3, 5, 14, 30, tzinfo=ist)

print(t.replace(tzinfo=timezone.utc))     # 2026-03-05 14:30:00+00:00  (a different mome
print(t.astimezone(timezone.utc))         # 2026-03-05 09:00:00+00:00  (the same moment)
```

Use `replace` only to attach a zone to a naive value whose zone you genuinely know. Use `astimezone` to convert something already aware. Calling `astimezone` on a *naive* value is allowed, but Python then assumes the machine's local zone, so the same program gives different answers on different machines. Avoid it.

Mixing the Two Fails

There is no sensible answer to "how far is 09:00 from 09:00 UTC", because a bare 09:00 might be in any zone. Python refuses rather than guess:

```
for op in ("<", "-"):
    try:
        print(eval("naive %s utc" % op))
    except TypeError as e:
        print("TypeError:", e)
# TypeError: can't compare offset-naive and offset-aware datetimes
# TypeError: can't subtract offset-naive and offset-aware datetimes

print(naive == utc)                       # False
```

Notice the last line. Ordering and subtraction raise, but `==` does not: it quietly returns `False`, even though the clock readings are the same. A bug that only uses equality will not announce itself, so keep the two kinds apart at the boundaries of your program instead of relying on errors.

Comparing Aware Values

Aware values compare by the moment they name, not by their readings. Two different readings can be equal, and two identical readings can differ:

```
a = datetime(2026, 3, 5, 14, 30, tzinfo=ist)         # 14:30 at +05:30
b = datetime(2026, 3, 5, 9, 0, tzinfo=timezone.utc)   # 09:00 at +00:00
print(a == b, a < b, a - b)                  # True False 0:00:00

x = datetime(2026, 3, 5, 9, 0, tzinfo=timezone(timedelta(hours=-5)))
y = datetime(2026, 3, 5, 9, 0, tzinfo=timezone.utc)
print(x == y, x - y)                         # False 5:00:00
print(x.astimezone(timezone.utc))            # 2026-03-05 14:00:00+00:00
```

The first pair are the same instant written two ways, so they are equal with a difference of zero. The second pair have identical readings but sit five hours apart. Because equal instants also hash equally, they behave as one item in a set or dictionary key, and sorting orders them by instant.

> **GETTING "NOW" CORRECTLY**
>
> `datetime.now()` returns a naive value in the machine's local zone. `datetime.now(timezone.utc)` returns an aware one. Prefer the second whenever the value will be stored, compared or sent elsewhere.

A Working Policy

- Decide at each boundary (input, storage, output) whether values are naive or aware, and convert there.
- Inside the program, keep one kind. For moments in time, that means aware.

- Never attach a zone with `replace` unless you know the zone the reading was in.
- Reserve naive values for things that really are readings with no place, such as "the shop opens at 09:00" before it is tied to a day and a location.

Challenges

EXERCISE 1

Write `describe(text)` that parses an ISO string with `fromisoformat` and returns the text followed by either `naive`, or `aware` with the offset in hours and the same moment written in UTC. Test it with `2026-03-05T09:00:00`, `2026-03-05T09:00:00Z`, `2026-03-05T14:30:00+05:30` and `2026-03-05T23:00:00-04:00`. Predict the UTC date of the last one before running it.

View solution

EXERCISE 2

Write `compare(a, b)` that returns `"before"`, `"same instant"` or `"after"`, and returns `"cannot compare"` when one value is naive and the other aware (catch the `TypeError`). Try it on a naive pair, on aware values `09:00 UTC` and `04:00 at -5 hours`, on `09:00 UTC` and `09:00 at -5 hours` in both orders, and on a naive and an aware pair. Finally print the result of `==` between the naive and aware pair.

View solution

EXERCISE 3

Create `09:00` on 5 March 2026 with a fixed offset of `+05:30`. Produce two new values from it: one with `astimezone(timezone.utc)` and one with `replace(tzinfo=timezone.utc)`. Print all three in ISO form, print whether the original equals each of the other two, and print the difference between the relabelled one and the original. Explain in a sentence which operation changed the moment.

View solution

QUICK REFERENCE

- **Naive:** no zone, `utcoffset()` is `None`; a reading, not a moment
- **Aware:** `tzinfo` set and `utcoffset()` not `None`; names an exact moment
- **Make aware:** `tzinfo=timezone.utc`, `timezone(timedelta(...))`, or ISO text with an offset or `Z`
- `astimezone` converts (same moment); `replace(tzinfo=)` relabels (different moment)
- **Mixing:** `<` and `-` raise `TypeError`; `==` silently gives `False`
- **Aware comparison** is by moment, so equal moments are equal whatever their readings
- **Now:** `datetime.now(timezone.utc)`, not `datetime.now()`

COMING NEXT

Date and Time in Practice 4 moves from fixed offsets to real time zones with `zoneinfo`: IANA names, converting between regions, and why UTC is the usual storage default.

Date and Time in Practice

Chapter 4 · Time Zones

Chapter 3 used fixed offsets such as `+05:30`. A fixed offset is the right tool when the offset really is fixed, but most places change theirs during the year. This chapter introduces real time zones through the standard-library `zoneinfo` module, which looks zones up by name in the IANA time zone database. The examples use Python 3.14, and every snippet was run; the output shown in the comments is what Python printed.

A Zone Is a Set of Rules

A `ZoneInfo` object is named by an IANA key of the form `Region/City`, such as `Europe/London` or `Asia/Tokyo`. It does not hold one offset; it holds the history and rules for a place, and answers with the offset that applies to the date you give it.

```
from datetime import datetime, timezone, timedelta
from zoneinfo import ZoneInfo, ZoneInfoNotFoundError

london = ZoneInfo("Europe/London")
summer = datetime(2026, 7, 1, 9, 0, tzinfo=london)
winter = datetime(2026, 1, 15, 9, 0, tzinfo=london)
print(summer, summer.tzname(), summer.utcoffset())   # 2026-07-01 09:00:00+01:00 BST 1:0
print(winter, winter.tzname(), winter.utcoffset())   # 2026-01-15 09:00:00+00:00 GMT 0:0
print(summer.tzinfo.key)                             # Europe/London
```

The same zone gave `+01:00` in July and `+00:00` in January. A fixed `timezone` object could never do that. The abbreviation (`BST`, `GMT`) is also a result, not an input: `tzname()` reports it, but you never construct a zone from one.

	`TIMEZONE(TIMEDELTA(...))`	`ZONEINFO("REGION/CITY")`
Offset	one value, always	depends on the date
Knows seasonal changes?	no	yes

	`TIMEZONE(TIMEDELTA(...))`	`ZONEINFO("REGION/CITY")`
Source of data	you	the IANA database
Best for	an offset that was recorded as a number	a place that people live in

Converting Between Zones

Conversion is the `astimezone` you met in Chapter 3. The moment stays the same; only the reading and label change.

```
tokyo = ZoneInfo("Asia/Tokyo")
ny = ZoneInfo("America/New_York")

print(summer.astimezone(tokyo))              # 2026-07-01 17:00:00+09:00
print(summer.astimezone(ny))                 # 2026-07-01 04:00:00-04:00
print(summer.astimezone(timezone.utc))       # 2026-07-01 08:00:00+00:00
print(summer == summer.astimezone(ny))       # True
```

The last line is the point of the exercise: three different readings, one instant, and equality holds. The offsets are not always whole hours (India is `+05:30`, Nepal `+05:45`), which is another reason never to compute a conversion by adding hours yourself.

> **ATTACHING A ZONE TO A READING**
>
> Writing `datetime(..., tzinfo=ZoneInfo(...))` is correct when you know the reading was made in that zone. It is the same idea as `replace(tzinfo=...)` from Chapter 3: it labels, it does not convert. Chapter 5 returns to the readings where even this labelling is ambiguous.

When a Name Is Not Found

Keys must match the database exactly. Abbreviations and misspellings are rejected with `ZoneInfoNotFoundError`, so user-supplied zone names should always be validated:

```
for name in ["Europe/Londn", "BST"]:
    try:
        ZoneInfo(name)
    except ZoneInfoNotFoundError as e:
        print(type(e).__name__, e)
# ZoneInfoNotFoundError 'No time zone found with key Europe/Londn'
# ZoneInfoNotFoundError 'No time zone found with key BST'
```

There is a second way to get the same error, and it has nothing to do with spelling. `zoneinfo` reads the zone database from the operating system. Windows does not ship one, so on a Windows machine every lookup fails until you install the `tzdata` package, which is a pure data package from the Python Package Index:

```
py -m pip install tzdata
```

If a program works on one machine and raises `ZoneInfoNotFoundError` for a perfectly good key on another, suspect a missing database before you suspect the key. Listing `available_timezones()` (from `zoneinfo`) shows what the current machine can resolve.

Store UTC, Show Local

The usual design is to convert to UTC as soon as a moment enters the program, store and compare it in UTC, and convert to a viewer's zone only to display it. UTC has no seasonal changes, so UTC values never become ambiguous, and sorting a column of them is correct.

```
meeting = datetime(2026, 10, 20, 15, 0, tzinfo=ny)
stored = meeting.astimezone(timezone.utc)
print(stored.isoformat())                  # 2026-10-20T19:00:00+00:00

for z in ["America/New_York", "Europe/London", "Asia/Tokyo", "Australia/Sydney"]:
    print(z, stored.astimezone(ZoneInfo(z)).strftime("%Y-%m-%d %H:%M %Z %z"))
# America/New_York 2026-10-20 15:00 EDT -0400
# Europe/London 2026-10-20 20:00 BST +0100
```

```
# Asia/Tokyo 2026-10-21 04:00 JST +0900
# Australia/Sydney 2026-10-21 06:00 AEDT +1100
```

Notice that two of the four readings fall on the next calendar day. That is normal, and it is why a stored date without its zone is not enough information to show a user.

Why Not Just Store the Offset?

A fixed offset recorded today describes today. A zone name describes a place, and the place's rules decide the offset on any other day. Compare a fixed `-5` with `America/New_York` on a July date:

```
fixed = timezone(timedelta(hours=-5))
jan = datetime(2026, 1, 15, 12, tzinfo=ny)
jul = datetime(2026, 7, 15, 12, tzinfo=ny)

print(jan.utcoffset().total_seconds() / 3600, jul.utcoffset().total_seconds() / 3600)
print(jan.astimezone(timezone.utc))         # 2026-01-15 17:00:00+00:00
print(jul.astimezone(timezone.utc))         # 2026-07-15 16:00:00+00:00
print(datetime(2026, 7, 15, 12, tzinfo=fixed).astimezone(timezone.utc))   # 2026-07-15 1
```

The fixed offset is wrong by an hour for the whole summer. If a user says "every Monday at 09:00 in New York", the thing to keep is the pair (`09:00`, `America/New_York`), not an offset.

Two aware values compare by instant, so zone choice does not affect equality when the instant is the same, and a shared clock reading in different zones is a different instant:

```
print(datetime(2026, 7, 1, 9, tzinfo=london) == datetime(2026, 7, 1, 8, tzinfo=timezone.
print(datetime(2026, 7, 1, 9, tzinfo=london) == datetime(2026, 7, 1, 9, tzinfo=ZoneInfo(
```

A WORKING RULE

Moments in the past or the near-term future that must stay fixed: store UTC. Recurring plans tied to a place ("09:00 local, every Monday"): store the local time and the zone name, and compute the UTC moment for each occurrence when you need it.

Challenges

EXERCISE 1

Write `world_clock(moment, names)` that takes one aware UTC datetime and a list of zone names, and prints one line per zone with the local date and time, the abbreviation and the offset in hours. Use the moment 23:30 UTC on 20 November 2026 with `Europe/London`, `Asia/Kolkata`, `Pacific/Auckland` and `America/Los_Angeles`. Before you run it, predict which zones will show a different calendar date from UTC.

View solution

EXERCISE 2

For `Europe/London`, `America/New_York`, `Australia/Sydney`, `Asia/Kolkata` and `Asia/Tokyo`, print the UTC offset in hours at noon on 15 January 2026 and at noon on 15 July 2026, and label each zone `changes` or `constant`. Predict the labels first, and say which zone has the larger offset in January.

View solution

EXERCISE 3

Write `get_zone(name)` that returns a `ZoneInfo`, or `None` when the name is not found, and test it with `Europe/Paris`, `BST`, `Europe/Londn` and `America/Chicago`. Then take 18:00 on 3 September 2026 in `Europe/Paris`, convert it to UTC for storage, print the stored ISO text, show it in `Europe/Paris`, `America/Chicago` and `Asia/Tokyo`, and print whether the stored value equals the original.

View solution

QUICK REFERENCE

- **Zone:** `ZoneInfo("Region/City")`, an IANA key; holds rules, not one offset
- **Convert:** `dt.astimezone(ZoneInfo(...))`; same instant, new reading
- **Label a known local reading:** `datetime(..., tzinfo=ZoneInfo(...))`
- **Bad key:** `ZoneInfoNotFoundError`; abbreviations are not keys
- **Missing database** (common on Windows): `py -m pip install tzdata`
- **Store UTC**; convert to a viewer's zone only to display
- **Recurring local plans:** keep the local time and the zone name, not an offset

COMING NEXT

Date and Time in Practice 5 looks at what happens when the rules change the clock: readings that do not exist, readings that happen twice, and the difference between adding a day and adding twenty-four hours.

Date and Time in Practice

Chapter 5 · Daylight Saving Traps

Chapter 4 showed that a zone's offset depends on the date. The moments when the offset changes are where most real-world date bugs live. In spring the clocks jump forward and an hour of readings never happens; in autumn they fall back and an hour of readings happens twice. This chapter shows what Python does at those moments, how to detect them, and why "tomorrow" and "24 hours from now" are different questions. Every snippet was run with Python 3.14; the comments show what was printed. In 2026 the New York changes are on 8 March and 1 November, and the London changes on 29 March and 25 October.

The Gap: A Reading That Does Not Exist

On 8 March 2026 New York clocks go from 01:59 straight to 03:00. A reading of 02:30 that day is not on any clock. Python will not stop you from writing it down:

```
from datetime import datetime, timedelta, timezone
from zoneinfo import ZoneInfo

ny = ZoneInfo("America/New_York")
gap = datetime(2026, 3, 8, 2, 30, tzinfo=ny)
print(gap)                                      # 2026-03-08 02:30:00-05:00
print(gap.astimezone(timezone.utc))             # 2026-03-08 07:30:00+00:00
print(gap.astimezone(timezone.utc).astimezone(ny))   # 2026-03-08 03:30:00-04:00
```

No exception was raised. Python used the offset from before the jump (`-05:00`) and built an object whose label says 02:30, but the instant it describes is 07:30 UTC, which on a real New York clock reads 03:30. The object lies about its own wall time. A round trip through UTC is the cheap way to find out what the clock really said.

The Fold: A Reading That Happens Twice

On 1 November 2026 New York clocks go from 01:59 back to 01:00. The readings from 01:00 to 01:59 occur twice, an hour apart. Python resolves the ambiguity with the `fold` attribute: `0` (the default) means the first time the reading occurred, `1` the second.

```
first  = datetime(2026, 11, 1, 1, 30, tzinfo=ny)
second = first.replace(fold=1)

print(first,  first.tzname(),  first.astimezone(timezone.utc))
# 2026-11-01 01:30:00-04:00 EDT 2026-11-01 05:30:00+00:00
print(second, second.tzname(), second.astimezone(timezone.utc))
# 2026-11-01 01:30:00-05:00 EST 2026-11-01 06:30:00+00:00
```

Two different instants, one hour apart, with identical wall readings. If a user types "01:30" on that night and you never ask which one they meant, you have to pick, and the default silently picks the first. Now the surprising part, which comes from a rule you may not know. When two aware datetimes share the same `ZoneInfo` object, Python compares and subtracts their wall readings and ignores the offsets:

```
print(first == second)                          # True
print(second - first)                           # 0:00:00
print(first.astimezone(timezone.utc) == second.astimezone(timezone.utc))   # False
print(second.astimezone(timezone.utc) - first.astimezone(timezone.utc))    # 1:00:00
```

Chapter 3 said that aware values compare by instant. That is true between different zones (or UTC), but inside one zone the shortcut applies, and at a fold it gives a wrong answer. The safe habit is to convert to UTC before comparing or subtracting anything that might straddle a change.

	GAP (CLOCKS FORWARD)	FOLD (CLOCKS BACK)
Wall readings	one hour never occurs	one hour occurs twice
2026, New York	8 March, 02:00-02:59	1 November, 01:00-01:59
2026, London	29 March, 01:00-01:59	25 October, 01:00-01:59

	GAP (CLOCKS FORWARD)	FOLD (CLOCKS BACK)
Day length	23 hours	25 hours
What `fold` does	picks which offset is used for the invalid reading	picks the first or second occurrence

Detecting the Problem

A reading is ordinary when both fold values give the same offset. If they differ, the reading is special, and the order of the two offsets says which kind. In a repeated hour the first pass is on summer time, so its offset is the larger. In a gap the first (`fold=0`) offset is the old one, which is the smaller.

```
def classify(dt):
    first = dt.replace(fold=0).utcoffset()
    second = dt.replace(fold=1).utcoffset()
    if first == second:
        return "normal"
    return "repeated" if first > second else "gap"

print(classify(datetime(2026, 3, 8, 2, 30, tzinfo=ny)))    # gap
print(classify(datetime(2026, 11, 1, 1, 30, tzinfo=ny)))   # repeated
print(classify(datetime(2026, 7, 1, 12, 0, tzinfo=ny)))   # normal
```

A function like this lets a form reject an impossible time with a clear message, or ask the user "first or second 01:30?" instead of guessing.

Adding a Day Is Not Adding 24 Hours

Take noon on Saturday 7 March 2026 in New York, the day before the spring change. `timedelta(days=1)` and `timedelta(hours=24)` are the same length, and in an aware datetime they give the same result, because Python adds to the wall reading. That is the trap: the sum is a wall-clock sum, whatever you call the delta.

```
start = datetime(2026, 3, 7, 12, 0, tzinfo=ny)

print(start + timedelta(days=1))              # 2026-03-08 12:00:00-04:00
print(start + timedelta(hours=24))            # 2026-03-08 12:00:00-04:00
print((start + timedelta(days=1)) - start)    # 1 day, 0:00:00

utc_sum = start.astimezone(timezone.utc) + timedelta(hours=24)
print(utc_sum.astimezone(ny))                 # 2026-03-08 13:00:00-04:00
```

The first two lines both say "same clock time tomorrow", which is what most people mean by "a day later". The subtraction claims exactly one day passed, but only 23 hours really did. Doing the arithmetic in UTC gives the true moment 24 hours later, which on the New York clock is 13:00. Neither answer is wrong; they answer different questions:

YOU MEAN...	DO THE ARITHMETIC IN...	EXAMPLE
"the same time tomorrow" (a calendar day)	the local zone, on the wall reading	a daily reminder at 12:00
"exactly 24 hours later" (elapsed time)	UTC, then convert for display	a timeout, a 24-hour token
"how long between these two"	UTC, always	billing, shift length

A WORKING RULE

Calendar intent (a day, a week, "same time next Monday") uses wall-clock arithmetic in the local zone, then checks the result with `classify`. Elapsed-time intent (a duration, a lifetime, an interval between two events) converts to UTC first. If you cannot say which one you mean, the requirement is not finished yet.

Landing in the Gap on Purpose

Wall-clock arithmetic can walk you into a gap without anyone typing a bad time. A daily job scheduled for 02:30 local time has no valid 02:30 on 8 March. Decide the policy in advance (run at 03:00, run at 03:30, skip the day) rather than letting the library choose. The check from earlier in the chapter tells you which days need the policy:

```
def real_wall_time(dt):
    """The reading a clock would actually show for dt."""
    return dt.astimezone(timezone.utc).astimezone(dt.tzinfo)

job = datetime(2026, 3, 8, 2, 30, tzinfo=ny)
print(classify(job), real_wall_time(job))     # gap 2026-03-08 03:30:00-04:00
```

Here the result is the same as the "shift forward by the gap" policy that many schedulers choose. Whichever policy you pick, write it down, because the next developer will meet the same day.

Challenges

EXERCISE 1

In `America/New_York`, build 02:30 on each of 7, 8, 9 and 10 March 2026. For each, print the wall reading, its UTC instant, and the reading you get after a round trip through UTC and back, marking any line where the round trip does not return the original wall time. Predict which date is marked before you run it.

View solution

EXERCISE 2

A shift runs from 00:00 to 08:00 local time. Write `hours_between(start, end)` returning both the naive difference (plain subtraction) and the real elapsed hours (through UTC). Use it for shifts on 8 March and 1 November 2026 in New York, on 29 March and 25 October 2026 in London, and on 15 July 2026 in New York. Predict which rows will differ before you run it.

View solution

EXERCISE 3

Write `classify(dt)` returning `"normal"`, `"gap"` or `"repeated"` using the two fold values, and test it with 02:30 and 01:30 on the New York change days (8 March, 1 November), plus 02:30 on 1 November, plus 01:30 on the two London change days (29 March, 25 October) and noon on 4 July in

London. Then, for the repeated New York 01:30, print the abbreviation and UTC time for `fold=0` and `fold=1`.

View solution

QUICK REFERENCE

- **Gap:** a wall reading that never occurs; Python builds it anyway with the old offset
- **Fold:** a wall reading that occurs twice; `fold=0` is the first, `fold=1` the second
- **Detect:** compare `utcoffset()` for `fold=0` and `fold=1`; equal means normal
- **Real reading:** `dt.astimezone(timezone.utc).astimezone(dt.tzinfo)`
- **Same zone object:** comparison and subtraction ignore offsets, which is wrong across a fold; convert to UTC first
- **Calendar intent** ("tomorrow at noon"): wall-clock arithmetic in the local zone
- **Elapsed intent** ("24 hours later", durations): arithmetic in UTC
- **Day length** on change days: 23 or 25 hours

COMING NEXT

Date and Time in Practice 6 moves from clocks to calendars: adding months when the target day does not exist, leap years, the `calendar` module and counting business days.

Date and Time in Practice

Chapter 6 · Calendar Arithmetic

Chapters 3 to 5 dealt with clocks. This chapter deals with calendars, where the trouble is that months have different lengths, years have different lengths, and "working day" is a local convention rather than a fact. `timedelta` counts days and seconds only; it has no idea what a month is. Every snippet below was run with Python 3.14; the comments show what was printed.

Months Are Not a Fixed Length

There is no `timedelta(months=1)`, because there is no single right answer for its length. The tempting shortcut is `replace(month=...)`, which fails whenever the day does not exist in the target month:

```
import calendar
from datetime import date, timedelta

try:
    date(2026, 1, 31).replace(month=2)
except ValueError as e:
    print(e)                                  # day 31 must be in range 1..28 for month 2

print(calendar.monthrange(2026, 2))           # (calendar.SUNDAY, 28)
print(calendar.monthrange(2028, 2))           # (calendar.TUESDAY, 29)
```

`calendar.monthrange(year, month)` returns the weekday of the 1st (Monday is 0, and the constants print by name) and the number of days in the month. The second value is what you need to clamp a day number safely:

```
def add_months(d, n):
    year, month = divmod(d.year * 12 + d.month - 1 + n, 12)
    month += 1
    last = calendar.monthrange(year, month)[1]
```

```
    return date(year, month, min(d.day, last))

print(add_months(date(2026, 1, 31), 1))   # 2026-02-28
print(add_months(date(2026, 1, 31), 2))   # 2026-03-31
print(add_months(date(2026, 3, 31), -1))  # 2026-02-28
print(add_months(date(2026, 11, 30), 3))  # 2027-02-28
```

The trick is `divmod` on a single month counter (`year * 12 + month - 1`), which handles year rollover in both directions without special cases. Now the trap: clamping throws information away.

```
once  = add_months(date(2026, 1, 31), 1)
twice = add_months(once, 1)
print(twice)                              # 2026-03-28
print(add_months(date(2026, 1, 31), 2))   # 2026-03-31
```

Adding one month twice and adding two months once disagree by three days. Once 31 January became 28 February, the 31 was gone for good. Always add the full number of months to the original date, and store the original date (for example a billing anchor of "the 31st") rather than the clamped one.

> **MONTH-END IS A POLICY**
>
> "Add a month" has at least three defensible meanings: clamp to the last day (used above), roll over into the next month (31 January plus one month becomes 3 March), or refuse. Billing systems usually clamp; some contracts roll over. Pick one, name it in the function, and test the 29th, 30th and 31st.

Leap Years

A year is a leap year if it is divisible by 4, except years divisible by 100, unless they are also divisible by 400. So 2000 and 2028 are leap years; 1900 and 2026 are not. Do not hand-roll this rule when the standard library has it:

```
print(calendar.isleap(1900), calendar.isleap(2000))    # False True
print(calendar.isleap(2028), calendar.isleap(2026))    # True False
print(calendar.leapdays(2000, 2030))            # 8

def next_birthday(born, today):
    for y in (today.year, today.year + 1):
        last = calendar.monthrange(y, born.month)[1]
        cand = date(y, born.month, min(born.day, last))
        if cand >= today:
            return cand

print(next_birthday(date(2024, 2, 29), date(2026, 3, 1)))   # 2027-02-28
print(next_birthday(date(2024, 2, 29), date(2027, 3, 1)))   # 2028-02-29
```

`calendar.leapdays(y1, y2)` counts leap years from `y1` up to but not including `y2`. The same clamping idea from `add_months` handles anniversaries of 29 February. Note also that a year has 365 or 366 days, so "add 365 days" is not "add a year": from 1 January 2028 it gives 31 December 2028, one day short of 1 January 2029.

The calendar Module

Weekdays come from the date itself; names and grids come from `calendar`:

```
d = date(2026, 11, 2)
print(d.weekday(), d.isoweekday())            # 0 1
print(calendar.day_name[d.weekday()])         # Monday
print(calendar.month_name[11], calendar.month_abbr[11])   # November Nov
print(calendar.monthcalendar(2026, 11)[0])        # [0, 0, 0, 0, 0, 0, 1]
print(calendar.monthcalendar(2026, 11)[-1])       # [30, 0, 0, 0, 0, 0, 0]
```

`weekday()` counts Monday as 0; `isoweekday()` counts Monday as 1 and Sunday as 7. Mixing the two is a classic off-by-one. `monthcalendar` returns one list per week, Monday first by default, with `0` padding days that belong to neighbouring months. Be aware that `calendar.day_name` and `month_name` follow the current locale, which Chapter 2 warned about.

The nth Weekday of a Month

Rules such as "the fourth Thursday of November" or "the last Monday of May" appear in holiday and meeting rules. Compute them rather than looking them up:

```
def nth_weekday(year, month, weekday, n):
    first = date(year, month, 1)
    offset = (weekday - first.weekday()) % 7
    day = 1 + offset + 7 * (n - 1)
    if day > calendar.monthrange(year, month)[1]:
        raise ValueError("no such date")
    return date(year, month, day)

def last_weekday(year, month, weekday):
    last = date(year, month, calendar.monthrange(year, month)[1])
    return last - timedelta(days=(last.weekday() - weekday) % 7)

print(nth_weekday(2026, 11, 3, 4))         # 2026-11-26   (4th Thursday)
print(nth_weekday(2026, 5, 0, 1))          # 2026-05-04   (1st Monday)
print(last_weekday(2026, 5, 0))            # 2026-05-25   (last Monday)
try:
    nth_weekday(2026, 2, 0, 5)
except ValueError as e:
    print(e)                                   # no such date
```

The offset line finds how many days to move forward from the 1st to reach the wanted weekday; the modulo keeps it between 0 and 6. A fifth occurrence may not exist, so the function says so instead of quietly spilling into the next month.

Business Days

A business day is a weekday that is not a holiday, and the holiday list belongs to a country, a company or a contract. There is no universal answer, so the list is data you pass in. The simple, correct version loops over the days:

```
def business_days(start, end, holidays=frozenset()):
```

```
    """Business days in [start, end): end is not counted."""
    n, d = 0, start
    while d < end:
        if d.weekday() < 5 and d not in holidays:
            n += 1
        d += timedelta(days=1)
    return n

def add_business_days(d, n, holidays=frozenset()):
    step = 1 if n >= 0 else -1
    left = abs(n)
    while left:
        d += timedelta(days=step)
        if d.weekday() < 5 and d not in holidays:
            left -= 1
    return d

hol = {date(2026, 11, 26)}
print(business_days(date(2026, 11, 2), date(2026, 11, 30)))        # 20
print(business_days(date(2026, 11, 2), date(2026, 11, 30), hol))   # 19
print(add_business_days(date(2026, 11, 26), 3, hol))      # 2026-12-01
print(add_business_days(date(2026, 11, 25), 1, hol))      # 2026-11-27
print(add_business_days(date(2026, 11, 27), 0))           # 2026-11-27
```

Looping is fast enough for ranges of months or years and is easy to trust. Edge cases to decide up front: whether the end date counts (here it does not, like Python's `range`), what adding zero business days does when the start is a weekend (here: returns it unchanged), and whether a holiday that falls on a weekend is observed on the next Monday (that is an entry in the holiday list, not logic in the function).

QUESTION	TOOL	WATCH FOR
Days in a month	`calendar.monthrange(y, m)[1]`	February in leap years
Is it a leap year	`calendar.isleap(y)`	1900 and 2100 are not
Add n months	your own `add_months`	clamp from the original date, not a clamped one
Weekday number	`weekday()` or `isoweekday()`	0-based versus 1-based
nth weekday of a month	your own `nth_weekday`	a fifth one may not exist
Working days between	loop with a holiday set	holidays are data, and the end date is excluded

Challenges

EXERCISE 1

Write `add_months` with clamping. Starting from 31 January 2026, print the result of adding 0 to 4 months directly, next to the result of adding one month four times in a row, and mark each row where the two differ. Predict which rows are marked before you run it.

View solution

EXERCISE 2

Someone was born on 29 February 2024. Print their birthday in each year from 2025 to 2029 (using 28 February in non-leap years) with the weekday and whether the year is a leap year. Then list the leap years from 1896 to 1912 and count the leap years from 1900 to 2100 inclusive. Predict the count first.

View solution

EXERCISE 3

With holidays on 25 and 28 December 2026, write `count_business(start, end)` and `add_business_days(d, n)`. Print the business days in December 2026, the date 10 business days after 15 December, the date 1 business day after 24 December, the result of adding 0 business days to Saturday 26 December, and the count for an empty range. Predict the December count by hand first.

View solution

QUICK REFERENCE

- **No month timedelta:** write `add_months` using `divmod` on `year * 12 + month - 1`
- **Clamp** the day with `min(day, calendar.monthrange(y, m)[1])`
- **Never chain clamped results:** add the full number of months to the original date
- **Leap year:** `calendar.isleap(y)`; divisible by 4, except centuries not divisible by 400
- **Weekdays:** `weekday()` Monday is 0; `isoweekday()` Monday is 1
- **nth weekday:** offset to the first match, plus 7 per extra week, check it still fits in the month

- **Business days:** weekday and not in a holiday set; decide whether the end date counts

COMING NEXT

Date and Time in Practice 7 looks at durations and intervals: the limits of `timedelta`, detecting overlaps, generating ranges and bucketing events into hours and days.

Date and Time in Practice

Chapter 7 · Durations and Intervals

So far a date or time has been a single point. Real programs mostly care about spans: how long something took, whether two bookings collide, which hours are free, how many events landed in each hour. This chapter covers the `timedelta` type in more detail, then builds the standard interval tools from a few lines each. Every snippet below was run with Python 3.14; the comments show what was printed.

What a timedelta Really Holds

A `timedelta` stores exactly three numbers: days, seconds and microseconds. Whatever you pass in is normalised into those, and the seconds and microseconds are always non-negative. That has one surprising consequence for negative durations:

```
from datetime import date, datetime, timedelta

print(timedelta(days=1, hours=36))              # 2 days, 12:00:00
print(timedelta(hours=-1))                      # -1 day, 23:00:00
print(timedelta(seconds=-1).days, timedelta(seconds=-1).seconds)   # -1 86399

d = timedelta(days=2, hours=3, minutes=4, seconds=5)
print(d.days, d.seconds, d.total_seconds())    # 2 11045 183845.0
```

Minus one hour is stored as "minus one day, plus 23 hours". It is correct, but printing it is confusing, and `.seconds` is not the total. The attributes `.days`, `.seconds` and `.microseconds` are the stored parts, not a breakdown for display. `.seconds` is never more than 86399, so it is not "the number of seconds" in a long duration. Use `total_seconds()` whenever you want one number.

Dividing and Formatting Durations

Dividing a `timedelta` by another `timedelta` tells you how many of one fit in the other. Floor division gives a whole count, modulo gives the remainder, and true division gives a float:

```
hour = timedelta(hours=1)
print(d // hour, d % hour)                    # 51 0:04:05
print(d / hour)                               # 51.06805555555555
print(timedelta(hours=1) * 2.5, timedelta(hours=1) / 4)   # 2:30:00 0:15:00

def hms(td):
    s = int(td.total_seconds())
    sign = "-" if s < 0 else ""
    h, r = divmod(abs(s), 3600)
    m, s = divmod(r, 60)
    return f"{sign}{h}:{m:02d}:{s:02d}"

print(hms(timedelta(days=1, hours=2, minutes=3, seconds=4)))   # 26:03:04
print(hms(-timedelta(minutes=90)))                    # -1:30:00
```

The built-in `str()` switches to "N days, H:MM:SS" once a duration passes 24 hours, which suits logs but not a timer display. The helper above keeps counting hours and puts the sign at the front, so a negative duration reads sensibly. Note that `int()` truncates toward zero, so fractions of a second are dropped, not rounded.

The Limits

A `timedelta` holds at most 999999999 days, and a `date` only runs from year 1 to year 9999. Overflow raises an error instead of wrapping:

```
print(timedelta.max)                          # 999999999 days, 23:59:59.999999
try:
    date(9999, 12, 31) + timedelta(days=1)
except OverflowError as e:
    print(e)                                  # date value out of range
try:
    timedelta(days=1_000_000_000)
except OverflowError as e:
```

```
    print(e)                                  # days=1000000000; must have magnitude <= 
```

You will rarely hit these by accident, but "no end date" values are a common way to do it: a sentinel such as `date.max` plus a grace period will overflow. Treat an open-ended interval as `None` and test for it, rather than inventing a very distant date.

Do Two Intervals Overlap?

Represent an interval as a start and an end, and decide once whether the end is inclusive. The convention that avoids most bugs is **half-open**: the start is included and the end is not, exactly like `range`. Then a meeting from 09:00 to 10:00 and another from 10:00 to 11:00 touch but do not collide. Two intervals overlap exactly when each starts before the other ends:

```
def overlaps(a_start, a_end, b_start, b_end):
    return a_start < b_end and b_start < a_end

t = lambda h, m=0: datetime(2026, 11, 2, h, m)
print(overlaps(t(9), t(10), t(10), t(11)))            # False  (they only touch)
print(overlaps(t(9), t(10, 30), t(10), t(11)))        # True
print(overlaps(t(9), t(12), t(10), t(11)))            # True   (one contains the other)

def intersect(a, b):
    s, e = max(a[0], b[0]), min(a[1], b[1])
    return (s, e) if s < e else None

print(intersect((t(9), t(11)), (t(10), t(12))))     # (datetime(2026, 11, 2, 10, 0), dat
print(intersect((t(9), t(10)), (t(10), t(12))))     # None
```

The test needs only two comparisons and no case analysis for "contains", "starts inside" or "ends inside". Both intervals must be the same kind of datetime: Chapter 3 showed that comparing naive and aware values raises an error, and for stored bookings the safe form is aware UTC values (Chapter 4).

Merging and Finding Free Time

Given many intervals, sort them by start and sweep once, extending the last interval whenever the next one begins at or before its end. Merging touching intervals as well as overlapping ones is usually what you want when counting booked time:

```
def merge(intervals):
    out = []
    for s, e in sorted(intervals):
        if out and s <= out[-1][1]:
            out[-1] = (out[-1][0], max(out[-1][1], e))
        else:
            out.append((s, e))
    return out

busy = [(t(13), t(14)), (t(9), t(10, 30)), (t(10), t(11)),
        (t(11), t(12)), (t(10, 15), t(10, 45))]
for s, e in merge(busy):
    print(s.time(), e.time())                # 09:00:00 12:00:00
                                             # 13:00:00 14:00:00

def free(busy, day_start, day_end):
    out, cur = [], day_start
    for s, e in merge(busy):
        if s > cur:
            out.append((cur, s))
        cur = max(cur, e)
    if cur < day_end:
        out.append((cur, day_end))
    return out

for s, e in free([(t(9), t(10, 30)), (t(13), t(14))], t(9), t(17)):
    print(s.time(), e.time())                # 10:30:00 13:00:00
                                             # 14:00:00 17:00:00
```

The `max(out[-1][1], e)` matters: a long interval can swallow shorter ones that follow, as the 10:15 to 10:45 entry shows. Dropping the `max` and using `e` would shrink the merged span and report phantom free time. Adding up the lengths of the unmerged intervals would also double-count overlaps, which is why you merge first and sum afterwards.

Generating Ranges

Python has no built-in date range, but a generator is three lines. Keep the same half-open rule so that consecutive ranges join without sharing a boundary:

```
def drange(start, stop, step):
    while start < stop:
        yield start
        start += step

print([x.strftime("%H:%M") for x in drange(t(9), t(11), timedelta(minutes=45))])
                                            # ['09:00', '09:45', '10:30']

print(len([date(2026, 11, 1) + timedelta(days=i) for i in range((date(2026, 12, 1) - dat
                                            # 30
print((date(2026, 11, 30) - date(2026, 11, 1)).days)   # 29
```

The last two lines show the fence-post trap: the days from 1 to 30 November inclusive number 30, but the difference between the dates is 29. Subtracting two dates gives the distance, not the count of days on the calendar. If your range includes both ends, add one; if it is half-open, the difference is the count. Note that `drange` uses a fixed step, so over a daylight saving change it should be given naive local times or aware UTC times, as Chapter 5 explained, never aware local times with a "one day" step expected to mean the same wall-clock time.

Bucketing

Bucketing means rounding each timestamp down to the start of its slot (hour, 15 minutes, day) so that you can count or group by slot. Floor division of a `timedelta` by a `timedelta` does it neatly when measured from a fixed starting point:

```
from collections import Counter

EPOCH = datetime(2000, 1, 1)

def floor_to(dt, step):
```

```
    return EPOCH + ((dt - EPOCH) // step) * step

now = datetime(2026, 11, 2, 10, 47, 31)
print(floor_to(now, timedelta(minutes=15)))     # 2026-11-02 10:45:00
print(floor_to(now, timedelta(hours=1)))        # 2026-11-02 10:00:00
print(now.replace(minute=0, second=0, microsecond=0))   # 2026-11-02 10:00:00

events = [t(9, 5), t(9, 40), t(9, 59), t(10), t(10, 1), t(11, 30)]
counts = Counter(floor_to(x, timedelta(hours=1)) for x in events)
for k in sorted(counts):
    print(k.strftime("%H:%M"), counts[k])  # 09:00 3
                                           # 10:00 2
                                           # 11:00 1
```

Because every bucket is anchored to the same fixed instant, an event at exactly 10:00:00 lands in the 10:00 bucket and not the 09:00 one, which is the half-open rule again. This works for any step that divides evenly into whole days. Buckets measured in weeks depend on which weekday your anchor falls on (2000-01-01 was a Saturday), and months and years have no fixed length at all, so group those by `(d.year, d.month)` instead. Also bucket in UTC or in naive local wall-clock time; a one-hour step through an aware local time crosses the daylight saving traps from Chapter 5.

TASK	TOOL	WATCH FOR
One number for a duration	`total_seconds()`	`.seconds` is only the leftover part, under 86400
How many hours fit	`td // timedelta(hours=1)`	floor division rounds toward minus infinity, so negatives round down too
Do two spans collide	`a_start < b_end and b_start < a_end`	decide whether the end is inclusive; prefer half-open
Total booked time	merge first, then sum	summing raw lengths double-counts overlaps
Open-ended interval	`None` for "no end"	a huge sentinel date can overflow
Group events by slot	`floor_to(dt, step)`	weeks and months are not fixed lengths

Challenges

EXERCISE 1

On 2 November 2026 the meetings are 14:00-15:00, 09:00-10:30, 10:00-11:00, 11:00-11:30, 09:15-09:45 and 16:00-16:30. Write `merge`, print the merged blocks with their lengths, the total booked time, and the sum of the raw meeting lengths. Predict both totals by hand first and explain the difference.

View solution

EXERCISE 2

The working day is 09:00-17:00 and the busy blocks are 09:00-10:30, 10:00-11:00, 12:30-13:00 and 15:00-17:00. Write `free_gaps` and print each free gap with its length. Then write `slots(gaps, length)` that lists the start times of back-to-back 45-minute meetings that fit entirely inside a gap, and print them. Predict the slot list first.

View solution

EXERCISE 3

Write `floor_to` and `hms`. Count these events per 15-minute bucket: 09:05, 09:14:59, 09:15, 09:44, 10:00, 10:01 and 23:59:59. Then print the span between the earliest and latest event with `str()` and with `hms`, the negated span with `hms`, `hms` of minus one second, and the first event floored to a whole day. Predict which events share a bucket first.

View solution

QUICK REFERENCE

- **timedelta** stores days, seconds and microseconds; use `total_seconds()` for one number
- **Negative durations** print as "-1 day, 23:00:00"; format them yourself with a sign-first helper
- **Divide** a timedelta by a timedelta: `//` whole count, `%` remainder, `/` float
- **Limits:** 999999999 days for a timedelta, years 1 to 9999 for a date; overflow raises `OverflowError`
- **Half-open intervals** (start in, end out) make touching spans not collide and ranges join cleanly
- **Overlap test:** `a_start < b_end and b_start < a_end`

- **Merge:** sort, sweep, extend with `max` of the ends; sum only after merging
- **Bucketing:** `EPOCH + ((dt - EPOCH) // step) * step`; group months by year and month
- **Counting days:** the difference of two dates is the distance; add one for an inclusive count

COMING NEXT

Date and Time in Practice 8 looks at timestamps and epochs: Unix time, seconds versus milliseconds, the year 2038 limit and why leap seconds do not appear in Python.

Date and Time in Practice

Chapter 8 · Timestamps and Epochs

A timestamp is a moment reduced to a single number: the count of time units since a fixed starting instant. Numbers are easy to store, sort and send between systems, which is why they are everywhere. They are also easy to misread, because the number alone does not say which unit it uses. This chapter covers the Unix epoch, the unit mix-ups that cause wrong dates, exact integer timestamps, the year 2038 limit, and what happens to leap seconds. Every snippet below was run with Python 3.14; the comments show what was printed.

The Unix Epoch

Unix time counts the seconds since **00:00:00 UTC on 1 January 1970**, called the epoch. It is always relative to UTC, so a timestamp means the same instant everywhere and carries no time zone of its own. Converting needs an aware datetime, as Chapter 3 and Chapter 4 argued:

```
from datetime import datetime, timezone, timedelta

EPOCH = datetime(1970, 1, 1, tzinfo=timezone.utc)

when = datetime(2026, 11, 2, 12, 0, tzinfo=timezone.utc)
print(when.timestamp())                      # 1793620800.0
print(int(when.timestamp()))                 # 1793620800
print(datetime.fromtimestamp(1_793_620_800, tz=timezone.utc))
                                             # 2026-11-02 12:00:00+00:00
print(datetime.fromtimestamp(0, tz=timezone.utc))   # 1970-01-01 00:00:00+00:00
print(EPOCH + timedelta(seconds=-86400))        # 1969-12-31 00:00:00+00:00
print(EPOCH + timedelta(seconds=1_793_620_800))  # 2026-11-02 12:00:00+00:00
```

Always pass `tz=timezone.utc` to `fromtimestamp`. Without it you get a naive datetime in the machine's local zone, and the same number then prints differently on different servers. The reverse is just as important: calling `.timestamp()` on a *naive* datetime makes Python assume it is local time, so the result depends on where the code runs. An aware datetime has no such ambiguity.

Timestamps before the epoch are negative. Adding a `timedelta` to `EPOCH` handles them everywhere, whereas `fromtimestamp` with a negative value raises an error on some platforms (it did on the Windows machine used to test this chapter). The `EPOCH + timedelta` form is also the cleaner choice whenever a number comes from outside your program.

Seconds, Milliseconds and Beyond

Python and Unix tools use seconds, but many other systems use milliseconds (browsers, Java, many web APIs), microseconds (databases, some logs) or nanoseconds (high-resolution clocks). Feeding a millisecond value to a function that expects seconds does not give a slightly wrong date; it gives an enormous year:

```
ms = 1_793_620_800_123
print(datetime.fromtimestamp(ms / 1000, tz=timezone.utc))
                                            # 2026-11-02 12:00:00.123000+00:00
try:
    datetime.fromtimestamp(ms, tz=timezone.utc)
except (OverflowError, OSError, ValueError) as e:
    print("error:", type(e).__name__)           # error: OSError  (the type varies by pl
```

The raw millisecond number is read as seconds, which lands roughly 57,000 years from now, past year 9999. The exception type differs between operating systems, so catch all three. A louder failure is the lucky case. If the wrong unit gives a date that is merely off (a stamp in seconds read as milliseconds is January 1970), nothing crashes and the bug hides.

The cure is to know the unit from the source, and to name variables with it (`created_ms`, `expires_s`). When you really must guess, the magnitude gives it away. A present-day stamp has 10 digits in seconds, 13 in milliseconds, 16 in microseconds and 19 in nanoseconds:

```
def to_seconds(v):
    a = abs(v)
    if a < 1e11: return v            # seconds
    if a < 1e14: return v / 1e3      # milliseconds
    if a < 1e17: return v / 1e6      # microseconds
    return v / 1e9                    # nanoseconds
```

```
for v in (1_793_620_800, 1_793_620_800_123, 1_793_620_800_123_456, 1_793_620_800_123_456_
    print(v, "->", EPOCH + timedelta(seconds=to_seconds(v)))
                                            # 1793620800 -> 2026-11-02 12:00:00+00:00
                                            # 1793620800123 -> 2026-11-02 12:00:00.123
                                            # 1793620800123456 -> 2026-11-02 12:00:00.
                                            # 1793620800123456789 -> 2026-11-02 12:00:
```

This is a heuristic for data cleaning, not a design. It works for dates between roughly 1973 and the year 5138 and breaks for tiny values near the epoch. Look closely at the last line: the nanosecond input came out as `.123457`, not `.123456789`. A `datetime` holds only microseconds, so nanoseconds are rounded away on the way in.

Exact Integer Timestamps

`timestamp()` returns a float. A 64-bit float carries about 15 to 16 significant digits, and a present-day stamp in microseconds already uses 16, with nanoseconds far beyond that. For ordinary work floats are fine, but for storage, comparison and equality checks, an integer count of a fixed unit is exact. Build it from the timedelta parts, which are already integers:

```
def to_us(dt):
    d = dt - EPOCH
    return (d.days * 86400 + d.seconds) * 1_000_000 + d.microseconds

def from_us(us):
    return EPOCH + timedelta(microseconds=us)

v = datetime(2026, 11, 2, 12, 0, 0, 123456, tzinfo=timezone.utc)
print(to_us(v))                             # 1793620800123456
print(from_us(to_us(v)))                    # 2026-11-02 12:00:00.123456+00:00
print(from_us(to_us(v)) == v)               # True
```

Python also offers `time.time_ns()`, an integer count of nanoseconds, for measuring the current time without float rounding. Use it, or `time.monotonic()` for elapsed-time measurement, since

the wall clock can be adjusted backwards by a time-sync correction while your code runs. Reserve timestamps for "when did this happen" and monotonic clocks for "how long did it take".

The Year 2038 Problem

Many older systems store Unix time in a **signed 32-bit integer**. Its largest value is $2^{31} - 1 = 2147483647$, which is 03:14:07 UTC on 19 January 2038. One second later the stored value overflows and wraps to the most negative number, which reads as a date in 1901:

```
import struct

print(2**31 - 1, EPOCH + timedelta(seconds=2**31 - 1))
                                          # 2147483647 2038-01-19 03:14:07+00:00
print(EPOCH + timedelta(seconds=2**31))
                                          # 2038-01-19 03:14:08+00:00  (what it shou

def wrap32(n):
    n &= 0xFFFFFFFF
    return n - 2**32 if n >= 2**31 else n

print(wrap32(2**31), EPOCH + timedelta(seconds=wrap32(2**31)))
                                           # -2147483648 1901-12-13 20:45:52+00:00  (
print(struct.pack("<I", 2**31).hex())        # 00000080  (the top bit is the sign bit)
print(EPOCH + timedelta(seconds=2**32 - 1))  # 2106-02-07 06:28:15+00:00  (the unsigned
```

Python itself is not affected, because its integers never overflow, and modern operating systems use 64-bit time. The danger lies in anything that squeezes a timestamp into 32 bits: old database columns, file formats, embedded devices, network protocols. Dates in the future matter first: a 20-year loan, a certificate or a session expiry computed today can already pass 2038. An unsigned 32-bit field postpones the problem to 2106 but cannot hold dates before 1970. The real fix is a 64-bit field, which is good for roughly 292 billion years in seconds.

Leap Seconds

The Earth's rotation is slightly irregular, so civil time (UTC) occasionally inserts a **leap second**, written 23:59:60. The most recent one was at the end of 31 December 2016. Unix time ignores them: it pretends every day has exactly 86400 seconds, so a leap second is not counted and has no timestamp of its own. Python follows suit:

```
try:
    datetime(2016, 12, 31, 23, 59, 60)
except ValueError as e:
    print(e)                                  # second must be in 0..59, not 60

a = datetime(2016, 12, 31, tzinfo=timezone.utc)
b = datetime(2017, 1, 1, tzinfo=timezone.utc)
print((b - a).total_seconds())                # 86400.0  (the real day was 86401 seconds
print(b.timestamp(), a.timestamp())           # 1483228800.0 1483142400.0
print(b.timestamp() % 86400)                 # 0.0  (midnight is always a multiple of 864
```

The practical effect is that Unix timestamps are not a true count of elapsed seconds; across a leap second they are one second short. For scheduling, logging and almost all business software that is exactly what you want, since a day always has 86400 seconds and midnight is always a whole multiple. Only precise scientific or satellite timing needs a time scale that counts every second, and that is not what `datetime` is for. Large providers often "smear" a leap second over many hours so that their clocks never show :60. If you receive a string containing `23:59:60` from an outside source, a strict parser such as Python's will reject it; clamp it to `23:59:59` before parsing if you must accept it.

QUESTION	ANSWER	WATCH FOR
What is the epoch?	00:00:00 UTC, 1 January 1970	timestamps carry no zone; they are always UTC-based
Timestamp to datetime	`fromtimestamp(t, tz=timezone.utc)`	omitting `tz` gives local naive time
Naive datetime to timestamp	avoid; make it aware first	Python assumes local time
Seconds or milliseconds?	count the digits: 10, 13, 16, 19	a wrong unit gives a huge year or 1970
Exact storage	integer microseconds	floats lose precision at nanosecond scale

QUESTION	ANSWER	WATCH FOR
Times before 1970	negative values	use `EPOCH + timedelta` for portability
32-bit limit	03:14:07 UTC, 19 January 2038	use 64-bit fields for anything that stores future dates
Leap seconds	not counted; no `:60`	a real day can be 86401 seconds, a Unix day never is

Challenges

EXERCISE 1

An export contains the values 1793620800, 1793620800500, 1793624400000, 1793706000250000, 0 and -86400, in mixed units. Write `to_seconds` and `to_dt` (using `EPOCH + timedelta`) and print each value as an ISO 8601 string. Then print the elapsed time between the second and third values and between the first and fourth. Predict each result by hand first, and say how you recognised each unit.

View solution

EXERCISE 2

Write `wrap32` to simulate a signed 32-bit integer. Print the last good second and the date it represents, the value and date it wraps to one second later, how long remains between 2026-11-02 12:00 UTC and the last good second (and the whole days), and the last second an unsigned 32-bit field could hold. Finish by printing `wrap32` of six values from 2147483646 up to 4294967296. Predict the wrapped values first.

View solution

EXERCISE 3

Write `to_us` and `from_us`. For four moments (2026-11-02 12:00:00.123456, 1969-07-20 20:17:40, 2000-01-01 and the epoch, all UTC) print the sorted integer microsecond values and whether each round-trips exactly. Split the first one with `divmod` into days since the epoch, seconds into the day and microseconds. Then show the leap-second day of 2016: the whole days and the seconds between 2016-12-31 and 2017-01-01, and the error raised when you build a time with second 60. Predict which values are negative.

View solution

QUICK REFERENCE

- **Epoch:** 00:00:00 UTC on 1 January 1970; a timestamp is a count of units since then and has no zone
- **To datetime:** `datetime.fromtimestamp(t, tz=timezone.utc)` or `EPOCH + timedelta(seconds=t)`
- **From datetime:** use an aware value; `.timestamp()` on a naive one assumes local time
- **Units:** 10 digits seconds, 13 milliseconds, 16 microseconds, 19 nanoseconds; name variables with the unit
- **Precision:** `datetime` keeps microseconds; store exact integer counts instead of floats
- **2038:** signed 32-bit seconds overflow at 03:14:07 UTC on 19 January 2038; use 64-bit
- **Leap seconds:** ignored by Unix time and rejected by `datetime`; every Unix day is 86400 seconds
- **Elapsed time:** use `time.monotonic()`, not the wall clock

COMING NEXT

Date and Time in Practice 9 looks at storing and exchanging dates: database column types, JSON, ISO 8601 strings and the rules for APIs.

Date and Time in Practice

Chapter 9 · Storing and Exchanging

A date that is right inside your program can still go wrong the moment it leaves it: written to a database, sent as JSON, read back by a different language on a different machine. This chapter collects the habits that keep a value intact on the way out and on the way back in: one text format, one zone for instants, a separate treatment for plain dates and for future local events, and strict checks on what you accept. Every snippet below was run with Python 3.14; the comments show what was printed.

ISO 8601 as the Wire Format

ISO 8601 is the standard way to write dates and times as text, and RFC 3339 is its stricter profile used by most web APIs. Its big advantage is that the pieces run from largest to smallest (year, month, day, hour...), so there is no day/month confusion, as Chapter 2 warned. For an instant, the form is `2026-11-02T12:00:05Z` or `2026-11-02T13:00:05+01:00`: the `T` separates date from time, and the trailing `Z` or offset says which zone the clock reading is in.

```
from datetime import datetime, date, timezone
from zoneinfo import ZoneInfo

t = datetime(2026, 11, 2, 12, 0, 5, 250000, tzinfo=timezone.utc)
print(t.isoformat())                          # 2026-11-02T12:00:05.250000+00:00
print(t.isoformat(timespec="seconds"))        # 2026-11-02T12:00:05+00:00
print(t.isoformat(timespec="milliseconds"))   # 2026-11-02T12:00:05.250+00:00
print(t.isoformat().replace("+00:00", "Z"))   # 2026-11-02T12:00:05.250000Z

for s in ("2026-11-02T12:00:05Z", "2026-11-02T13:00:05+01:00",
          "2026-11-02T12:00:05", "2026-11-02", "20261102T120005Z"):
    d = datetime.fromisoformat(s)
    print(repr(s), "->", d, d.tzinfo is not None)
                                              # '2026-11-02T12:00:05Z' -> 2026-11-02 12:
                                              # '2026-11-02T13:00:05+01:00' -> 2026-11-0
                                              # '2026-11-02T12:00:05' -> 2026-11-02 12:0
                                              # '2026-11-02' -> 2026-11-02 00:00:00 Fals
```

```
                                                # '20261102T120005Z' -> 2026-11-02 12:00:0
```

Three things to notice. First, `isoformat()` writes `+00:00` for UTC, so you replace it with `Z` yourself if the receiving side expects that spelling. Second, since Python 3.11 `fromisoformat` understands `Z`, the compact form without dashes and most other ISO variants; on older versions the `Z` raises an error. Third, a string with no offset comes back *naive*, and a bare date comes back as midnight. Neither is an instant, so a receiving function must decide what to do with them rather than silently accept them (Chapter 3).

Pick **one** spelling for everything you write: UTC, a fixed number of fractional digits, one choice between `Z` and `+00:00`. A consistent format can be compared as plain text, which databases and log tools rely on. Mixed offsets cannot:

```
a = datetime.fromisoformat("2026-11-02T13:00:05+01:00")
b = datetime.fromisoformat("2026-11-02T12:00:05Z")
print(a == b, a.isoformat() < b.isoformat())  # True False

mixed = ["2026-11-02T13:00:00+01:00", "2026-11-02T12:30:00+00:00"]
print(sorted(mixed))
                                              # ['2026-11-02T12:30:00+00:00', '2026-11-0
print(sorted(mixed, key=datetime.fromisoformat))
                                              # ['2026-11-02T13:00:00+01:00', '2026-11-0
```

The two values in `a` and `b` are the same instant, yet their strings compare as different, and text sorting puts 13:00+01:00 after 12:30+00:00 although it is half an hour earlier. The fix is to normalise to UTC before writing: `dt.astimezone(timezone.utc).isoformat(timespec="seconds")`.

JSON Has No Date Type

JSON knows strings, numbers, booleans, lists, objects and null. A date therefore always travels as a string (ISO 8601) or a number (a timestamp, with the unit problems of Chapter 8), and the reader has to know which fields to convert back. Python's `json` refuses to guess:

```
import json

ev = {"id": 7, "name": "Launch", "starts": t, "day": date(2026, 11, 2)}
try:
    json.dumps(ev)
except TypeError as e:
    print("TypeError:", e)               # TypeError: Object of type datetime is not JSO

def encode(o):
    if isinstance(o, datetime):
        if o.tzinfo is None:
            raise ValueError("naive datetime")
        return o.astimezone(timezone.utc).isoformat().replace("+00:00", "Z")
    if isinstance(o, date):
        return o.isoformat()
    raise TypeError(type(o).__name__)

text = json.dumps(ev, default=encode, sort_keys=True)
print(text)
                                         # {"day": "2026-11-02", "id": 7, "name": "
back = json.loads(text)
starts = datetime.fromisoformat(back["starts"])
day = date.fromisoformat(back["day"])
print(starts == t, day)                  # True 2026-11-02
```

Notice the order of checks in `encode`: `datetime` is a subclass of `date`, so it must be tested first or every datetime would be written as a bare date. The function also *rejects* naive datetimes with a `ValueError` instead of guessing a zone, so a missing zone is caught at the boundary where the cause is still obvious. On the way back, `json.loads` only gives you strings; converting `starts` and `day` is an explicit step, and it should be driven by a schema or a list of field names, never by "this string looks like a date".

Write timestamps as numbers only when the contract says so, and then state the unit in the field name (`created_ms`). A float timestamp survives a JSON round trip (`1793620800.123456` came back identical), but other languages may parse every JSON number as a 32-bit or 53-bit value, so an integer count in a documented unit is safer than a float.

Dates in a Database

Server databases such as PostgreSQL and MySQL have proper date and time column types, and the names matter. A PostgreSQL `timestamptz` stores an instant (converted to UTC internally and shown in the session's zone), while a plain `timestamp` stores a clock reading with no zone, exactly the naive datetime of Chapter 3. Use the first for "when did this happen" and the second almost never. A `date` column is for calendar dates. SQLite has no date types at all: you choose between ISO text, an integer timestamp or a number, and you convert explicitly (the automatic converters of Python's `sqlite3` module were deprecated in Python 3.12 for this reason). The same rules apply in every database, so SQLite's built-in module is enough to practise them:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE ev (id INTEGER PRIMARY KEY, name TEXT, at_utc TEXT, at_epoch I

rows = [("a", datetime(2026, 11, 2, 12, tzinfo=timezone.utc)),
        ("b", datetime(2026, 3, 1, 9, 30, tzinfo=ZoneInfo("America/New_York")))]
for n, dt in rows:
    u = dt.astimezone(timezone.utc)
    con.execute("INSERT INTO ev (name, at_utc, at_epoch) VALUES (?,?,?)",
                (n, u.isoformat(timespec="seconds"), int(u.timestamp())))

for r in con.execute("SELECT name, at_utc, at_epoch FROM ev ORDER BY at_utc"):
    print(r)
                                            # ('b', '2026-03-01T14:30:00+00:00', 17723
                                            # ('a', '2026-11-02T12:00:00+00:00', 17936
r = con.execute("SELECT at_utc FROM ev WHERE name='b'").fetchone()[0]
print(datetime.fromisoformat(r).astimezone(ZoneInfo("America/New_York")))
                                            # 2026-03-01 09:30:00-05:00
```

Text in a fixed UTC format sorts and range-queries correctly with ordinary string comparison, and stays readable when you inspect the table by hand; an integer is more compact and equally sortable but opaque to the eye. Either is fine, provided the whole table uses the same one. Notice how a ranged query uses a half-open interval, the pattern from Chapter 7:

```
q = "SELECT name FROM ev WHERE at_utc >= ? AND at_utc < ?"
print(con.execute(q, ("2026-03-01T00:00:00+00:00", "2026-04-01T00:00:00+00:00")).fetchal
                                                  # [('b',)]
```

Keep the boundaries in the same format as the stored values (the same offset spelling and the same number of fractional digits), otherwise the text comparison is no longer reliable. Remember also that "all of March" depends on whose March: build the boundaries from the user's zone, then convert them to UTC, as Chapter 6 and Chapter 7 did.

Three Kinds of Date

Not every date is an instant, and forcing them all into UTC is a common source of bugs. Decide which of the three each field is:

KIND	EXAMPLE	STORE AS
An instant that already happened	order placed, log entry, login	UTC instant (ISO text or integer)
A calendar date	birthday, invoice date, holiday	`date` / `YYYY-MM-DD`, no time, no zone
A future local appointment	meeting at 09:00 London time next March	local date-time *and* zone name, e.g. `2027-03-28 09:00` + `Europe/London`

A birthday stored as midnight UTC will show as the previous day to anyone west of Greenwich, so a calendar date must stay a `date`. The third kind is subtler. Governments change daylight saving rules, sometimes with only weeks of notice. If you store only the converted UTC instant for a meeting booked a year ahead, and the rules then change, the instant is stale and the meeting appears at the wrong local hour. Storing the wall-clock time and the IANA zone name keeps the intent, and the instant is recomputed when it is needed:

```
rec = {"local": "2027-03-28T09:00:00", "zone": "Europe/London"}
loc = datetime.fromisoformat(rec["local"]).replace(tzinfo=ZoneInfo(rec["zone"]))
print(loc, loc.astimezone(timezone.utc))
```

```
                                                # 2027-03-28 09:00:00+01:00 2027-03-28 08:
```

The zone's rules are read at the moment of conversion, from whatever time zone database is installed, so updating that database updates every future meeting without touching your data. Past events are the opposite: they happened at a fixed instant, so UTC is final and no zone name is needed (you may still keep the user's zone for display).

Rules for APIs

- **Send and store instants in UTC**, as ISO 8601 / RFC 3339 text with an explicit `Z` or offset, with the same precision everywhere.
- **Reject, do not guess.** A string with no offset, a bare date where an instant is expected, or a format like `02/11/2026` should produce a clear error. A permissive parser turns a client's mistake into silently wrong data.
- **Document each field's kind and unit**: instant, calendar date, or local time plus zone; seconds or milliseconds for numbers.
- **Convert for display last**, in the user's zone, at the edge of the system. Everything inside stays in UTC.
- **Send zone names, not abbreviations.** `Europe/London` is unambiguous; `BST`, `CST` and `IST` each mean more than one thing.
- **Test the boundaries**: midnight, end of month, a daylight saving change, a leap day and a timestamp near a limit, as in earlier chapters.

A small validator that follows the second rule makes the behaviour concrete. It accepts anything with an offset, converts it to UTC, and says why it rejects the rest:

```
def parse_instant(text):
    try:
        dt = datetime.fromisoformat(text)
    except ValueError:
        raise ValueError("not an ISO 8601 date-time") from None
    if dt.utcoffset() is None:
        raise ValueError("missing offset")
    return dt.astimezone(timezone.utc)
```

```
for s in ("2026-11-02T13:00:00+01:00", "2026-11-02T12:00:00", "02/11/2026 12:00"):
    try:
        print(s, "->", parse_instant(s).isoformat())
    except ValueError as e:
        print(s, "-> rejected:", e)
                                            # 2026-11-02T13:00:00+01:00 -> 2026-11-02T
                                            # 2026-11-02T12:00:00 -> rejected: missing
                                            # 02/11/2026 12:00 -> rejected: not an ISO
```

Challenges

EXERCISE 1

Write `dumps(obj)` and `loads(text, instants, days)` for JSON. `dumps` writes aware datetimes as UTC text ending in `Z` (to the second), writes dates as `YYYY-MM-DD`, refuses naive datetimes with a

fields back. Round-trip an order placed at 09:30 New York time on 2026-11-02 with a ship date of 2026-11-05, print the JSON and the converted values, and say whether the round-tripped moment equals the original and whether it keeps the original offset. Predict the JSON first.

View solution

EXERCISE 2

Build an in-memory SQLite table of meetings with the columns title, `at_utc` (ISO text in UTC) and zone. Add four meetings given as local times in four zones, list them ordered by `at_utc`, select the ones that fall in December 2026 (UTC) using a half-open range, and print each meeting in its own zone. Finally show that three timestamps with different offsets sort wrongly as text but correctly with a key function. Predict the stored UTC values and which meetings fall in December before you run it.

View solution

EXERCISE 3

Part A: write `parse_instant` so it returns UTC for strings with an offset or `Z`, and rejects, with a reason, strings with no offset, a bare date, a day/month/year string, and an impossible month. Run it on

six inputs. Part B: for a meeting booked as local `2027-03-28 09:00` in `Europe/London`, compute the UTC instant from the local time and zone, and then show what you would get if the zone's rules were changed to stay on UTC all year. Explain why storing local time plus zone survives that change while a stored UTC instant does not.

View solution

QUICK REFERENCE

- **Format:** ISO 8601 / RFC 3339, `2026-11-02T12:00:05Z`; one spelling and one precision everywhere
- **Write:** `dt.astimezone(timezone.utc).isoformat(timespec="seconds")`; naive values are rejected, not guessed
- **Read:** `datetime.fromisoformat(s)` (Python 3.11+ accepts `Z`); check `utcoffset()` is not `None`
- **JSON:** no date type; use a `default=` function to write, convert named fields to read; test `datetime` before `date`
- **Databases:** use a zone-aware instant column; SQLite needs explicit ISO text or integers; query with half-open ranges
- **Three kinds:** instant (UTC), calendar date (`date`), future local appointment (local time + IANA zone name)
- **Mixed offsets:** compare as datetimes, never as text
- **APIs:** UTC on the wire, zone names not abbreviations, convert to local time only for display

COMING NEXT

Date and Time in Practice 10 is the capstone: a scheduling toolkit for recurring events across time zones, built from everything in the course and tested.

Date and Time in Practice

Chapter 10 · Capstone Toolkit

This final chapter puts the course together. You will build a small scheduling toolkit with four parts: one function that turns "local date, wall time and zone" into a correct moment, a generator for weekly and monthly recurring events, a search for the next occurrence after a given moment, and a finder for working hours that several zones share. Then you will test it against the traps from earlier chapters. Every snippet was run with Python 3.14; the comments show what was printed.

The Design in Four Rules

1. **A recurring event is a rule, not a list of instants.** "Mondays at 09:00 in London" is stored as weekdays, a wall time and a zone name (Chapter 9, future local appointments).
2. **Generate each occurrence from the local date.** Step through calendar dates in the rule's zone, attach the wall time, then convert. Never add `timedelta(days=7)` to one UTC instant (Chapter 5).
3. **Return aware values only**, and reject naive input at the boundary (Chapter 3).
4. **Decide every edge case once, in one place:** a wall time that does not exist, a wall time that happens twice, and a day number that a month does not have.

Step 1: Resolving a Local Time

The core of the toolkit is a function that turns a date, a wall time and a zone name into one definite moment. Its policy for the two awkward cases is deliberately simple: a time that falls in a spring gap moves forward by the length of the gap, and a time that happens twice takes its first occurrence (`fold=0`).

```
from datetime import datetime, date, time, timedelta, timezone
from zoneinfo import ZoneInfo
import calendar

UTC = timezone.utc
```

```
def resolve(d, t, zone):
    """Local date + wall time + zone -> aware datetime. A time inside a gap moves forwar
    z = ZoneInfo(zone)
    dt = datetime.combine(d, t, tzinfo=z)
    return dt.astimezone(UTC).astimezone(z)

print(resolve(date(2027, 3, 28), time(1, 30), "Europe/London"))
                                                    # 2027-03-28 02:30:00+01:00
r = resolve(date(2026, 10, 25), time(1, 30), "Europe/London")
print(r, r.fold)                                    # 2026-10-25 01:30:00+01:00 0
```

The round trip through UTC is what repairs the gap: `zoneinfo` gives the non-existent 01:30 the offset from before the change, so converting to UTC and back lands on 02:30 BST. For an ambiguous time the same call returns the first of the two, offset +01:00. If your product needs a different policy (skip the occurrence, or use the second one), this is the one function to change.

Step 2: Recurrence

A weekly rule walks forward one calendar day at a time and keeps the days whose weekday is in the set. A monthly rule fixes a day number and clamps it to the last day of shorter months (Chapter 6), so "the 31st" becomes the 28th in February instead of failing.

```
def weekly(start, t, zone, weekdays, count):
    """First `count` occurrences on the given weekdays (0=Mon) from start, at wall time
    out, d = [], start
    while len(out) < count:
        if d.weekday() in weekdays:
            out.append(resolve(d, t, zone))
        d += timedelta(days=1)
    return out

def monthly(year, month, t, zone, day, count):
    """Same day-of-month each month, clamped to the month's last day."""
    out = []
    for _ in range(count):
        last = calendar.monthrange(year, month)[1]
        out.append(resolve(date(year, month, min(day, last)), t, zone))
```

```
            month += 1
            if month == 13:
                year, month = year + 1, 1
        return out

    for o in weekly(date(2027, 3, 22), time(9), "Europe/London", {0, 3}, 4):
        print(o, o.astimezone(UTC).strftime("%H:%M"))
                                                    # 2027-03-22 09:00:00+00:00 09:00
                                                    # 2027-03-25 09:00:00+00:00 09:00
                                                    # 2027-03-29 09:00:00+01:00 08:00
                                                    # 2027-04-01 09:00:00+01:00 08:00
    print([str(x.date()) for x in monthly(2027, 1, time(12), "UTC", 31, 3)])
                                                    # ['2027-01-31', '2027-02-28', '2027-0
```

The weekly output is the whole lesson of the course in four lines: the wall time stays at 09:00 while the UTC time moves from 09:00 to 08:00 when British Summer Time begins on 28 March. A calendar app that stored the first occurrence as a UTC instant and repeated it every 7 days would show 10:00 after the change.

Step 3: The Next Occurrence

Most real code does not need the whole series, only "when is the next one after now?". Start from today's date *in the rule's zone*, which may differ from the UTC date, and take the first candidate that is strictly later than `now`. Looking ahead 8 days is enough to see every weekday at least once.

```
    def next_after(now, t, zone, weekdays):
        if now.tzinfo is None:
            raise ValueError("now must be aware")
        d = now.astimezone(ZoneInfo(zone)).date()
        for _ in range(8):
            if d.weekday() in weekdays:
                cand = resolve(d, t, zone).astimezone(UTC)
                if cand > now:
                    return cand
            d += timedelta(days=1)
        raise ValueError("no weekdays given")
```

```
now = datetime(2026, 11, 2, 23, 30, tzinfo=UTC)       # Monday night in UTC, already Tue
print(next_after(now, time(9), "Asia/Tokyo", {1}))
                                                      # 2026-11-03 00:00:00+00:00
```

Step 4: Finding a Shared Window

To schedule one meeting for people in several zones, check each UTC hour of the day and keep the hours that fall inside everyone's working hours. Because it works from the actual date, daylight saving is accounted for without any special code:

```
def overlap(zones, start_h, end_h, d):
    """UTC hours on date d at which every zone is inside [start_h, end_h) local time."""
    ok = []
    base = datetime(d.year, d.month, d.day, tzinfo=UTC)
    for h in range(24):
        u = base + timedelta(hours=h)
        if all(start_h <= u.astimezone(ZoneInfo(z)).hour < end_h for z in zones):
            ok.append(u)
    return ok

pair = ["Europe/London", "America/New_York"]
print([x.strftime("%H:%M") for x in overlap(pair, 9, 17, date(2026, 11, 2))])
                                                      # ['14:00', '15:00', '16:00']
print([x.strftime("%H:%M") for x in overlap(pair, 9, 17, date(2026, 3, 16))])
                                                      # ['13:00', '14:00', '15:00', '16:00']
print([x.strftime("%H:%M") for x in overlap(pair + ["Asia/Tokyo"], 9, 17, date(2026, 11,
                                                      # []
```

The same two cities have a three-hour shared window on 2 November and a four-hour window on 16 March, because on 16 March New York is already on daylight time and London is not, so the gap between them is four hours instead of five. Add Tokyo and there is no shared hour at all. An empty result is a legitimate answer, so the caller must handle it (for example by widening the hours or rotating the unfriendly time between the zones).

Testing the Toolkit

Date code is easy to get almost right, so the tests are not an extra: they are how you know it works. Choose cases at the boundaries, state the expected value by hand, and use fixed dates rather than the current time so the tests give the same result every day:

```
import unittest

class T(unittest.TestCase):
    def test_wall_time_stable_utc_moves(self):
        occ = weekly(date(2027, 3, 22), time(9), "Europe/London", {0}, 2)
        self.assertEqual([o.hour for o in occ], [9, 9])
        self.assertEqual([o.astimezone(UTC).hour for o in occ], [9, 8])
    def test_gap(self):
        r = resolve(date(2027, 3, 28), time(1, 30), "Europe/London")
        self.assertEqual((r.hour, r.minute, r.utcoffset()), (2, 30, timedelta(hours=1)))
    def test_fold(self):
        r = resolve(date(2026, 10, 25), time(1, 30), "Europe/London")
        self.assertEqual((r.fold, r.utcoffset()), (0, timedelta(hours=1)))
    def test_month_clamp(self):
        m = monthly(2027, 1, time(12), "UTC", 31, 3)
        self.assertEqual([x.day for x in m], [31, 28, 31])
    def test_overlap(self):
        o = overlap(pair, 9, 17, date(2026, 11, 2))
        self.assertEqual([x.hour for x in o], [14, 15, 16])
    def test_overlap_dst_mismatch(self):
        o = overlap(pair, 9, 17, date(2026, 3, 16))
        self.assertEqual([x.hour for x in o], [13, 14, 15, 16])

unittest.main(argv=["x"], verbosity=1)
                                                    # ......
                                                    # Ran 6 tests in 0.000s
                                                    # OK
```

Writing the last test taught a useful lesson: the author's first guess for the expected value was three hours, and the test failed with four. The test was right and the guess was wrong, because the gap between the cities that week was four hours, not five. That is exactly why you work out the expected value from first principles and then let the code argue with you.

Course Checklist

- **Types and text (1-2):** use ISO 8601 for text; never rely on locale-dependent or day/month-ambiguous formats.
- **Naive vs aware (3):** a naive value is not a moment; attach a zone or reject it.
- **Zones (4):** IANA names, UTC for storage and comparison, local time for display.
- **Daylight saving (5):** a day is not always 24 hours; handle the gap and the repeated hour on purpose.
- **Calendar (6-7):** clamp month ends, mind leap years, use half-open intervals.
- **Timestamps (8):** know the unit, and the limits of the type that holds them.
- **Storage (9):** instants in UTC, calendar dates as dates, future local events as wall time plus zone name.
- **Always:** test the boundaries, with fixed dates, and work out the expected answer by hand first.

Challenges

EXERCISE 1

A team stand-up happens every Tuesday and Thursday at 10:00 New York time, starting on 2026-10-27. Using the toolkit's `resolve` and `weekly`, list the first six occurrences, and for each print the New York time with its zone abbreviation, the UTC time, and the London time with its abbreviation. Predict, before running, which rows change their UTC hour and why London's time is the same as UTC for every row.

View solution

EXERCISE 2

Use `resolve` to schedule a daily 01:30 London job for 27 to 30 March 2027, printing the local time with its abbreviation and the UTC time for each day. Then build 01:30 on 25 October 2026 in London with `fold=0` and `fold=1` and print both in local and UTC time. Predict which day's 01:30 is moved and to what, and how far apart the two folds are. Finish by comparing the two fold values with `==` and explain the result.

View solution

EXERCISE 3

Write `next_after(now, t, zone, weekdays)`, which returns the first occurrence strictly after an aware `now` as a UTC datetime and rejects a naive `now`. Then write a `unittest` suite with five tests: a later time on the same day, a `now` that is exactly on the occurrence, a search that crosses a clock change, a zone whose date is ahead of the UTC date, and a naive input. Work out all expected values by hand before running the suite.

View solution

QUICK REFERENCE

- **Recurring event:** store weekdays or day number + wall time + IANA zone name, never a list of UTC instants
- **Resolve:** `datetime.combine(d, t, tzinfo=z).astimezone(UTC).astimezone(z)` repairs a gap and picks the first of a repeated hour
- **Generate:** step through local calendar dates, attach the wall time, then convert
- **Month ends:** `min(day, calendar.monthrange(y, m)[1])`
- **Next occurrence:** start from today's date in the rule's zone and require the result to be strictly after `now`
- **Shared window:** test each UTC hour against every zone's local hours; an empty list is a valid result
- **Compare:** convert to UTC first; same-zone aware values compare by wall clock and ignore `fold`
- **Test:** fixed dates, boundary cases, expected values worked out by hand

COURSE COMPLETE

That is the end of Date and Time in Practice. You can now read, write, convert, store and schedule dates and times without the usual surprises. The best next step is to take a date function from one of your own projects and run it through the boundary tests above.