

Day 21 Standard library deep dive

Self-Study Guide · About 80 minutes · With answer key

By the end you will be able to

- Count, group, and queue data with Counter, defaultdict, namedtuple, and deque instead of hand-written loops.
- Combine and group iterables with chain, combinations, permutations, and groupby from itertools.
- Work out differences between dates and times, and turn dates into text and back again.
- Find and replace text with regular expressions, and read command-line options with argparse.

What to remember from earlier

- To count, write `counts[item] = counts.get(item, 0) + 1` for each item. ([Day 8: Dictionaries, 8.9 Counting with a Dictionary](#))
- To group, add a key with an empty list the first time you see it, then append each item to that key's list. ([Day 8: Dictionaries, 8.10 Grouping with a Dictionary](#))
- `sorted(items, key=...)` sorts by whatever the key function returns for each item, often a lambda. ([Day 10: Functions, part 2, 10.8 Sorting with key](#))
- Subtracting two dates gives a `timedelta`, and its `.days` is the gap in whole days. ([Day 13: Modules, packages and environments, 13.7 Dates and Times with datetime](#))
- `sys.argv` is the list of words typed to start a program, with the script's name first. ([Day 13: Modules, packages and environments, 13.8 The Computer and the Program: os and sys](#))
- A generator or other iterator gives each value once; a second loop over it finds nothing left. ([Day 18: Iterators, generators and decorators, 18.1 What a for Loop Really Does: iter\(\) and next\(\)](#))
- The hint `X | None` means a value of type `X`, or `None`, so check for `None` before using it. ([Day 19: Type hints and code style, 19.3 Values That Might Be None](#))

Make a `day21` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days. Nothing needs installing today: every module on this page comes with Python.

On Day 8 you wrote eight lines to count words and find the most common one. On Day 4 you worked out the rule for leap years. Both jobs are so common that Python already has them, written, tested, and fast, in its **standard library**: the collection of modules that comes with every Python installation (Day 13 toured `math`, `random`, `datetime`, `os`, and `sys`). Today you meet five more modules that replace whole loops you have been writing by hand: `collections` for counting and grouping, `itertools` for combining iterables, `datetime` (properly this time) for calendars and clocks, `re` for finding patterns in text, and `argparse` for reading command-line options. The habit to take away is as important as any one module: before you write a utility yourself, look for it in the standard library first.

21.1 Counting with Counter

Counting things is everywhere: votes, words, page visits, dice rolls. Day 8's pattern works, but you write it every time, with an empty dictionary, a loop, and `counts.get(item, 0) + 1`. Then you write Day 5's maximum loop to find the winner, and a second, longer loop if you want the top three. The `collections` module ("specialized collection types") has a type that does all of that.

A `Counter` is a dictionary made for counting. Hand it any iterable (a list, a string, a generator), and it counts each different item: the items become the keys, and their counts become the values. Because it is a dictionary, everything from Day 8 still works on it: `[]`, `in`, `items()`, `len()`.

```
>>> from collections import Counter
>>> votes = Counter(["pizza", "tacos", "pizza", "sushi", "pizza", "tacos"])
>>> votes
Counter({'pizza': 3, 'tacos': 2, 'sushi': 1})
>>> votes["pizza"]
3
>>> votes["curry"]
0
>>> len(votes)
3
>>> votes.most_common(2)
[('pizza', 3), ('tacos', 2)]
>>> votes.total()
6
>>> votes["sushi"] += 1
>>> votes.update(["curry", "pizza"])
>>> votes
Counter({'pizza': 4, 'tacos': 2, 'sushi': 2, 'curry': 1})
```

- `Counter([...])` does the whole Day 8 loop in one call. It shows the counts from the most common down.
- `votes["curry"]` gives 0 for an item that was never counted, instead of a `KeyError`. Reading a missing key does **not** add it: `len(votes)` is still 3.
- `most_common(n)` returns the `n` most common items as a list of `(item, count)` tuples, biggest count first. Without `n`, it returns them all.
- `total()` adds up all the counts: six votes in all.
- Because a missing count reads as 0, `votes["sushi"] += 1` works even for a new item, with no `get` needed. `update(iterable)` counts more items into the same `Counter`.

When two items have the same count, `most_common` keeps them in the order they were first counted. So a tie always comes out the same way, run after run.

WORKED EXAMPLE The fruit stall's best sellers

A market stall wrote down each sale as one word. Which three fruits sold best?

fruit_sales.py

```

1 from collections import Counter
2
3 sales = "apple pear apple plum pear apple kiwi plum apple pear".split()
4 counts = Counter(sales)
5
6 print(f"{counts.total()} sales of {len(counts)} kinds of fruit")
7 for fruit, count in counts.most_common(3):
8     print(fruit, count, "*" * count)
9 print("Kiwis:", counts["kiwi"], "Mangoes:", counts["mango"])

```

```

10 sales of 4 kinds of fruit
apple 4 ****
pear 3 ***
plum 2 **
Kiwis: 1 Mangoes: 0

```

- Line 3 splits the string into a list of ten words (Day 3), and line 4 counts them: apple 4, pear 3, plum 2, kiwi 1.
- Line 6: `total()` is 10, and `len(counts)` is 4, the number of **different** fruits, because a `Counter` is a dictionary with one key per fruit.
- Line 7 loops over the three `(fruit, count)` pairs from `most_common(3)`, unpacking each one (Day 7). `"*" * count` repeats a star `count` times (Day 3), a tiny bar chart.
- Line 9: no mangoes were sold, so `counts["mango"]` is 0, with no error.

Compare Day 8: an empty dictionary, a loop with `get`, and a separate maximum loop, which found only the single winner. Here the counting is one call, and the top three is one more.

KEY IDEA

`Counter(iterable)` counts every item in one call. A missing item counts as 0, `most_common(n)` gives the top `n` as `(item, count)` pairs, `total()` adds up all the counts, and `update()` counts more items.

WATCH OUT

Counting a string when you meant to count its words. A string is an iterable of characters (Day 5), so a `Counter` counts letters, and spaces:

```

>>> from collections import Counter
>>> Counter("to be or not to be")
Counter({' ': 5, 'o': 4, 't': 3, 'b': 2, 'e': 2, 'r': 1, 'n': 1})
>>> Counter("to be or not to be".split())
Counter({'to': 2, 'be': 2, 'or': 1, 'not': 1})

```

The first `Counter` says the most common "word" is a space. To count words, split the text into a list of words first, with `split()`.

PAUSE AND TRY**Question 1** Tracing

What does this program print?

counter_trace.py

```
1 from collections import Counter
2
3 letters = Counter("mississippi")
4 print(letters.most_common(2))
5 print(letters["p"], letters["z"], len(letters))
6 letters.update("pop")
7 print(letters.most_common(1), letters.total())
```

21.2 Default Values with defaultdict

Day 8's grouping pattern has the same shape as counting: before you can append to a key's list, you check whether the key exists and, if not, add it with an empty list. Those two lines run only the first time each key appears, yet you write them every time. A `defaultdict` (also from `collections`) is a dictionary that fills in a missing key for you, the moment you use it.

You create it with a **default factory**: a function that `defaultdict` calls, with no arguments, to make the starting value for each new key. Day 10 showed that a function is a value you can pass around without calling it. The built-in types work as factories, because calling one with no arguments makes an "empty" value:

```
>>> from collections import defaultdict
>>> int()
0
>>> list()
[]
>>> minutes = defaultdict(int)
>>> minutes["reading"] += 30
>>> minutes["math"] += 45
>>> minutes["reading"] += 20
>>> minutes
defaultdict(<class 'int'>, {'reading': 50, 'math': 45})
>>> dict(minutes)
{'reading': 50, 'math': 45}
```

- `int()` gives 0 and `list()` gives []. So `defaultdict(int)` starts each new key at 0, and `defaultdict(list)` starts each new key with an empty list. Pass the function itself, `int` or `list`, with no parentheses.
- `minutes["reading"] += 30` reads a key that does not exist yet. Instead of raising `KeyError`, the `defaultdict` calls `int()`, stores the 0, and then adds 30.
- Printing it shows the factory, `<class 'int'>`, and then the ordinary dictionary. `dict(minutes)` turns it into a plain dictionary when you want plainer output.

WORKED EXAMPLE Names by first letter

Here is Day 8's grouping job with a `defaultdict(list)`. The variable annotation (Day 19) says what the dictionary will hold, since it starts empty:

`group_by_letter.py`

```
1 from collections import defaultdict
2
3 names = ["Ben", "Ana", "Bea", "Cal", "Abe", "Bo"]
4
5 by_letter: defaultdict[str, list[str]] = defaultdict(list)
6 for name in names:
7     by_letter[name[0]].append(name)
8
9 for letter, group in by_letter.items():
10     print(letter, group)
```

```
B ['Ben', 'Bea', 'Bo']
A ['Ana', 'Abe']
C ['Cal']
```

name	name[0]	What line 7 does
Ben	B	B is new: make [], then append Ben
Ana	A	A is new: make [], then append Ana
Bea	B	B exists: append Bea
Cal	C	C is new: make [], then append Cal
Abe	A	A exists: append Abe
Bo	B	B exists: append Bo

The `if key not in groups`: lines from Day 8 are gone. Line 7 does both jobs:

`by_letter[name[0]]` is either the existing list or a brand-new empty one, and `.append(name)` adds to it. The groups appear in the order their keys were first used.

`Counter` and `defaultdict(int)` can both count. Use `Counter` when you want counts and the extras (`most_common`, `total`); use `defaultdict(int)` or `defaultdict(float)` when you are adding up amounts by category, as in Day 8's spending totals.

KEY IDEA

`defaultdict(factory)` calls the factory to make a starting value whenever a missing key is used: `defaultdict(list)` for grouping (`groups[key].append(item)`), `defaultdict(int)` for counting or totals. Pass the function, not a call.

WATCH OUT

Looking up a key adds it. With a `defaultdict`, merely reading a missing key creates it:

defaultdict_lookup.py

```

1 from collections import defaultdict
2
3 by_letter: defaultdict[str, list[str]] = defaultdict(list)
4 for name in ["Ben", "Ana", "Bea"]:
5     by_letter[name[0]].append(name)
6
7 for letter in "ABZ":
8     if by_letter[letter]:
9         print(letter, by_letter[letter])
10    else:
11        print(letter, "nobody")
12 print(len(by_letter), "letters:", list(by_letter))

```

```

A ['Ana']
B ['Ben', 'Bea']
Z nobody
3 letters: ['B', 'A', 'Z']

```

The check on line 8 asked about Z, so Z now exists with an empty list, and the dictionary claims 3 letters where only 2 have names. To ask about a key without creating it, use `letter in by_letter` (Day 8's `in` checks keys) or `by_letter.get(letter, [])`, which never adds anything.

PAUSE AND TRY

Question 2 Tracing

What does this program print?

defaultdict_trace.py

```

1 from collections import defaultdict
2
3 purchases = [("food", 12.5), ("games", 20.0), ("food", 8.25), ("books", 9.99)]
4 spending: defaultdict[str, float] = defaultdict(float)
5 for category, amount in purchases:
6     spending[category] += amount
7
8 print(spending["food"], spending["toys"])
9 print(len(spending), sorted(spending))

```

21.3 Records with namedtuple

Day 7 stored a student as a tuple, `("Ana", 10, 92)`. That is compact and cannot change by accident, but code like `student[2]` makes the reader stop and count: is position 2 the grade or the score? A dictionary has names, but nothing stops a typo creating a new key, and Day 17's `@dataclass` is a whole class. For a small, fixed record, there is something in between.

`namedtuple` (from `collections`) makes a new tuple type whose positions also have names. You call it with the type's name and a list of field names, and it gives back a class (Day 16), which you name in `CapWords`:

```
>>> from collections import namedtuple
>>> Student = namedtuple("Student", ["name", "grade", "score"])
>>> ana = Student("Ana", 10, 92)
>>> ana
Student(name='Ana', grade=10, score=92)
>>> ana.score
92
>>> ana[0]
'Ana'
>>> name, grade, score = ana
>>> grade
10
>>> ana == ("Ana", 10, 92)
True
>>> ana._replace(score=95)
Student(name='Ana', grade=10, score=95)
>>> ana._asdict()
{'name': 'Ana', 'grade': 10, 'score': 92}
```

- `Student("Ana", 10, 92)` makes one record, like calling any class. Its printed form names every field.
- `ana.score` reads a field by name. It is still a tuple, so `ana[0]`, unpacking (Day 7), and `==` with a plain tuple all still work.
- Like any tuple, it cannot change. `_replace(field=value)` makes a **new** record with one field changed. `_asdict()` gives the fields as a dictionary. The leading underscore is only there so these method names cannot clash with your field names.

WORKED EXAMPLE Ranking a class

`class_ranking.py`

```
1 from collections import namedtuple
2
3 Student = namedtuple("Student", ["name", "grade", "score"])
4
5 students = [
6     Student("Ana", 10, 92),
7     Student("Ben", 11, 78),
8     Student(name="Chloe", grade=10, score=85),
9 ]
10
11 best = max(students, key=lambda student: student.score)
12 print(f"Top score: {best.name} with {best.score}")
13
14 for student in sorted(students, key=lambda student: student.score):
15     print(student.name, student.score)
16
17 tenth = [student.name for student in students if student.grade == 10]
18 print("Grade 10:", tenth)
```

```

Top score: Ana with 92
Ben 78
Chloe 85
Ana 92
Grade 10: ['Ana', 'Chloe']

```

Line 8 shows that fields can also be passed by keyword (Day 10). `max` and `sorted` with `key=` (Day 10) read `student.score`, which says exactly what is being compared; with plain tuples it would be `student[2]`. The comprehension on line 17 (Day 11) keeps the names of the students whose `grade` is 10.

You want	Use
A few values together, used once, close by	A plain tuple (Day 7)
A small, fixed record that never changes	A <code>namedtuple</code>
A record that changes, or needs methods or rules	A <code>@dataclass</code> or a class (Day 17)
Keys you do not know in advance	A dictionary (Day 8)

`namedtuple` fields have no type hints. If you want hints, the `typing` module (Day 19) offers a class form, written like a data class:

`namedtuple_hints.py`

```

1 from typing import NamedTuple
2
3
4 class Student(NamedTuple):
5     name: str
6     grade: int
7     score: int
8
9
10 ana = Student("Ana", 10, 92)
11 print(ana, ana.score)

```

```
Student(name='Ana', grade=10, score=92) 92
```

KEY IDEA

`namedtuple("Name", ["field", ...])` makes a tuple type whose items can be read by name (`record.field`) as well as by position. Records cannot change; `_replace()` makes a changed copy.

WATCH OUT

Trying to change a field. A named tuple is still a tuple:

namedtuple_change.py

```
1 from collections import namedtuple
2
3 Student = namedtuple("Student", ["name", "grade", "score"])
4
5 ana = Student("Ana", 10, 92)
6 ana.score = 95
7 print(ana)
```

AttributeError: can't set attribute

Make a new record instead: `ana = ana._replace(score=95)`. If the record really needs to change often, it is a better fit for a data class. Older versions of Python may word the message differently.

PAUSE AND TRY

Question 3 Multiple choice

`Point = namedtuple("Point", ["x", "y"])` and `p = Point(3, 4)`. Which statement is **false**?

- A. `p.x + p[1]` is 7
- B. `p == (3, 4)` is True
- C. `p.x = 5` changes `p` to `Point(x=5, y=4)`
- D. `p._replace(y=0)` gives `Point(x=3, y=0)` and leaves `p` unchanged

21.4 Queues with deque

Day 6 kept a deli queue in a list, serving each customer with `pop(0)`. That works, but it gets slow as the list grows. A list stores its items in one row of numbered slots. Removing slot 0 means moving every other item one place to the left so that the numbering still starts at 0: with 100,000 people waiting, one `pop(0)` moves 99,999 items. Adding at the front with `insert(0, item)` does the same, the other way. Only the **end** of a list is cheap to change.

A **deque** (from `collections`, pronounced "deck", short for "double-ended queue") is built so that **both** ends are cheap. Adding or removing at either end takes the same short time, however long it is.

```
>>> from collections import deque
>>> line = deque(["Ana", "Ben"])
>>> line.append("Chloe")
>>> line.appendleft("Dev")
>>> line
deque(['Dev', 'Ana', 'Ben', 'Chloe'])
>>> line.popleft()
'Dev'
>>> line.pop()
'Chloe'
>>> line
deque(['Ana', 'Ben'])
>>> line[0]
'Ana'
>>> len(line)
2
```

Job	List	deque
Add at the back	<code>append(x)</code>	<code>append(x)</code>
Remove from the back	<code>pop()</code>	<code>pop()</code>
Add at the front	<code>insert(0, x)</code> , slow	<code>appendleft(x)</code>
Remove from the front	<code>pop(0)</code> , slow	<code>popleft()</code>
Read an item in the middle	<code>items[500]</code> , fast	<code>items[500]</code> , slower

`len`, `in`, `for` loops, and indexing work on a `deque` as on a list. A `deque` gives up fast access to the middle in return for fast ends, so use it for queues, and a list when you mostly read by position.

WORKED EXAMPLE A help desk queue

A **queue** is first in, first out: people join at the back and are served from the front. The loop runs while anyone is waiting, because an empty `deque` is `falsy` (Day 4):

help_desk.py

```
1 from collections import deque
2
3 waiting = deque(["Ana", "Ben", "Chloe"])
4 served = 0
5
6 while waiting:
7     customer = waiting.popleft()
8     served += 1
9     print(f"Serving {customer}; {len(waiting)} still waiting")
10    if customer == "Ana":
11        waiting.append("Dev")
12
13 print("Served", served, "customers")
```

```
Serving Ana; 2 still waiting
Serving Ben; 2 still waiting
Serving Chloe; 1 still waiting
Serving Dev; 0 still waiting
Served 4 customers
```

Ana is served first and leaves two waiting. Then, on line 11, Dev joins at the back, so after Ben is served there are still two waiting: Chloe and Dev. Dev, who arrived last, is served last.

A deque can also have a `maxlen`, a largest size. When a full deque gets a new item at one end, it drops an item from the **other** end to make room. That makes "keep only the last few" a one-liner:

recent_pages.py

```
1 from collections import deque
2
3 recent: deque[str] = deque(maxlen=3)
4 for page in ["home", "news", "sport", "weather", "news"]:
5     recent.append(page)
6     print(list(recent))
```

```
['home']
['home', 'news']
['home', 'news', 'sport']
['news', 'sport', 'weather']
['sport', 'weather', 'news']
```

Once three pages are stored, each `append` on the right pushes the oldest page out on the left. `news` appears twice in the history because a deque keeps repeats, like a list. (Printing `list(recent)` keeps the output short; printing the deque itself would also show `maxlen=3`.)

KEY IDEA

A deque adds and removes quickly at both ends: `append` and `pop` at the back, `appendleft` and `popleft` at the front. Use it for queues instead of `pop(0)`. With `maxlen`, a full deque drops an item from the opposite end.

WATCH OUT

Serving from an empty queue. This loop serves three times, but only two people are waiting:

empty_queue.py

```
1 from collections import deque
2
3 waiting = deque(["Ana", "Ben"])
4 for _ in range(3):
5     print("Serving", waiting.popleft())
```

```
Serving Ana
Serving Ben
Traceback (most recent call last):
...
IndexError: pop from an empty deque
```

Loop while waiting: instead, or check if waiting: before each `popleft()`, as you would before `pop(0)` on a list.

PAUSE AND TRY

Question 4 Tracing

What does this program print?

deque_trace.py

```
1 from collections import deque
2
3 orders: deque[int] = deque(maxlen=3)
4 for number in [101, 102, 103, 104]:
5     orders.append(number)
6 orders.appendleft(100)
7 print(orders)
8 print(orders.popleft(), orders.pop(), len(orders))
```

21.5 Chaining and Choosing: chain, combinations, permutations

The `itertools` module ("tools for iterators") holds ready-made building blocks for loops. Day 18 used its `islice` to take the first few items of an endless generator. Three more answer questions that would otherwise need nested loops:

- `chain(a, b, ...)` gives every item of `a`, then every item of `b`, and so on, as one stream. The iterables can be of different kinds: a list, then a tuple, then a generator.
- `combinations(items, r)` gives every way to **choose** `r` items, where order does not matter: choosing A and B is the same choice as B and A, so only one of them appears.
- `permutations(items, r)` gives every way to **arrange** `r` items, where order matters: A then B is different from B then A. Leave out `r` to arrange all the items.

Each result is a tuple, and no item is used twice in one result. Like Day 18's generators, these functions are **lazy**: they return an iterator that makes each result when the loop asks for it. To look at all the results at once, wrap the call in `list()`.

```
>>> from itertools import chain, combinations, permutations
>>> morning = ["Ana", "Ben"]
>>> afternoon = ("Chloe", "Dev")
>>> list(chain(morning, afternoon))
['Ana', 'Ben', 'Chloe', 'Dev']
>>> list(combinations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(permutations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
>>> list(permutations("AB"))
[('A', 'B'), ('B', 'A')]
```

`chain` saves building a joined list with `+` just to loop over it, and it works where `+` cannot, such as a list followed by a tuple. Three letters give three pairs to choose but six pairs to arrange, because each choice can be arranged two ways.

WORKED EXAMPLE Fixtures for a league

Four teams each play every other team once. Then the league decides that each pair plays twice, once at each team's home ground.

fixtures.py

```
1 from itertools import combinations, permutations
2
3 teams = ["Lions", "Owls", "Foxes", "Bears"]
4
5 print("League: each pair plays once")
6 for number, pair in enumerate(combinations(teams, 2), start=1):
7     print(f"{number}. {pair[0]} v {pair[1]}")
8
9 home_and_away = list(permutations(teams, 2))
10 print("Home and away:", len(home_and_away), "matches")
11 print("First three:", home_and_away[:3])
```

League: each pair plays once

1. Lions v Owls
2. Lions v Foxes
3. Lions v Bears
4. Owls v Foxes
5. Owls v Bears
6. Foxes v Bears

Home and away: 12 matches

First three: [('Lions', 'Owls'), ('Lions', 'Foxes'), ('Lions', 'Bears')]

- Lines 6 and 7: `combinations(teams, 2)` gives each pair once, in the order the teams appear, and `enumerate(..., start=1)` (Day 5) numbers them. Lions v Owls appears, and Owls v Lions does not.
- Line 9: in home-and-away fixtures, Lions at home to Owls is a different match from Owls at home to Lions, so order matters: `permutations`. `list()` collects the results so that `len` can count them and a slice (Day 6) can show the first three.

The counts follow a rule. With n items, there are $n \times (n - 1)$ ordered pairs: $4 \times 3 = 12$. Each unordered pair is counted twice in that, so there are $n \times (n - 1) / 2$ combinations: $12 / 2 = 6$.

The numbers grow fast: permutations of 10 items has 3,628,800 results. Because the iterator is lazy, looping over it uses almost no memory, but `list()` would store every result at once. Loop over big results; only `list()` small ones.

KEY IDEA

`chain` joins iterables end to end. `combinations(items, r)` chooses r items where order does not matter; `permutations(items, r)` arranges them, so order matters. All three return lazy iterators; use `list()` to see small results.

WATCH OUT

Using up the iterator before the loop. Counting the matches first leaves nothing to loop over:

used_up.py

```
1 from itertools import combinations
2
3 matches = combinations(["Lions", "Owls", "Foxes"], 2)
4 print("Matches:", len(list(matches)))
5 for team_a, team_b in matches:
6     print(team_a, "v", team_b)
7 print("Done")
```

```
Matches: 3
Done
```

`list(matches)` on line 4 runs the iterator to its end (Day 18), so the `for` loop on line 5 finds nothing, and no match is printed. Store the list once, `matches = list(combinations(...))`, and use that list for both jobs.

PAUSE AND TRY

Question 5 Tracing

What does this program print?

choose_trace.py

```
1 from itertools import chain, combinations, permutations
2
3 toppings = ["ham", "egg", "kale", "corn"]
4 print(len(list(combinations(toppings, 3))))
5 print(list(combinations(toppings, 3))[0])
6 print(len(list(permutations(toppings, 2))))
7 print(list(chain("ab", [1, 2])))
```

21.6 Grouping Runs with groupby

How long was the longest rainy spell? Which days in a row had the same weather? These questions are about **runs**: items next to each other that share something. `groupby` from `itertools` splits an iterable into runs. For each run it gives a pair: the shared value, called the **key**, and an iterator over the items in that run.

groupBy_runs.py

```
1 from itertools import groupby
2
3 weather = ["sun", "sun", "rain", "rain", "rain", "sun", "snow"]
4 for kind, days in groupby(weather):
5     print(kind, len(list(days)))
```

```
sun 2
rain 3
sun 1
snow 1
```

`sun` appears twice in the output, because there are two separate sunny runs. `groupby` only ever compares an item with the one just before it, so it never looks back to join the two. Each run's items come as an iterator, so line 5 uses `list(days)` to collect them before counting them with `len`.

To group by something other than the item itself, pass `key=`, a function, as with `sorted` (Day 10). And to make each group hold **every** matching item, sort by the same key first, so that matching items sit next to each other:

WORKED EXAMPLE Names by first letter, again

groupBy_letter.py

```
1 from itertools import groupby
2
3 names = ["Ben", "Ana", "Bea", "Cal", "Abe", "Bo"]
4 names.sort()
5 print(names)
6 for letter, group in groupby(names, key=lambda name: name[0]):
7     print(letter, list(group))
```

```
['Abe', 'Ana', 'Bea', 'Ben', 'Bo', 'Cal']
A ['Abe', 'Ana']
B ['Bea', 'Ben', 'Bo']
C ['Cal']
```

After sorting, all the A names are together, then the B names, then C. The `key` function takes each name's first letter, so each run is one letter. Compare 21.2's `defaultdict` version of this job:

Tool	Needs sorted input?	Gives	Best for
<code>defaultdict(list)</code>	No	A dictionary of complete groups	Grouping data in any order

groupby	Yes, for full groups	One run at a time, in order	Runs, and data already in order
---------	----------------------	-----------------------------	---------------------------------

`groupby` shines when the data is already in order, such as log lines in time order (see Putting it together), or when the runs themselves are the point.

KEY IDEA

`groupby(items, key=...)` gives `(key, run)` pairs for each run of neighboring items with the same key. Sort by the same key first if you want one group per key. Each run is an iterator; use `list()` to keep its items.

WATCH OUT

Forgetting to sort first. Without line 4's `sort()`, the same program prints:

`groupby_unsorted.py`

```
1 from itertools import groupby
2
3 names = ["Ben", "Ana", "Bea", "Cal", "Abe", "Bo"]
4 for letter, group in groupby(names, key=lambda name: name[0]):
5     print(letter, list(group))
```

```
B ['Ben']
A ['Ana']
B ['Bea']
C ['Cal']
A ['Abe']
B ['Bo']
```

No two neighbors share a first letter, so every name is a run of one. There is no error, just wrong groups. Sort by the same key first, or use a `defaultdict(list)`.

PAUSE AND TRY

Question 6 Tracing

What does this program print?

`groupby_trace.py`

```
1 from itertools import groupby
2
3 code = "".join(f"{char}{len(list(run))}" for char, run in groupby("aaabccaa"))
4 print(code)
5 longest = max(len(list(run)) for char, run in groupby("xyyyxxz"))
6 print(longest)
```

21.7 Dates, Times, and Differences

Day 13 met the `datetime` module's three main types: `date` for a calendar day, `datetime` for a day and a time, and `timedelta` for a length of time. Add a `timedelta` to a date and you get a new date; subtract two dates and you get a `timedelta`. Real programs are built on exactly that arithmetic: due dates, deadlines, shift lengths, ages.

WORKED EXAMPLE Library due dates and fines

A library lends books for 21 days and fines 25 cents for each day late:

`library_fines.py`

```
1 from datetime import date, timedelta
2
3 LOAN_DAYS = 21
4 FINE_PER_DAY = 0.25
5
6
7 def due_date(borrowed: date) -> date:
8     """Return the day a book borrowed on the given day is due back."""
9     return borrowed + timedelta(days=LOAN_DAYS)
10
11
12 def fine(borrowed: date, returned: date) -> float:
13     """Return the late fine in dollars, or 0.0 if the book is on time."""
14     days_late = (returned - due_date(borrowed)).days
15     return max(0, days_late) * FINE_PER_DAY
16
17
18 borrowed = date(2026, 2, 10)
19 print("Due back:", due_date(borrowed))
20 print(f"Returned 1 March: ${fine(borrowed, date(2026, 3, 1)):.2f}")
21 print(f"Returned 10 March: ${fine(borrowed, date(2026, 3, 10)):.2f}")
```

```
Due back: 2026-03-03
Returned 1 March: $0.00
Returned 10 March: $1.75
```

- `due_date` adds 21 days. 10 February plus 21 days: 18 days to the end of February (2026 is not a leap year, so it has 28 days), then 3 more, giving 3 March. The `timedelta` counted February's length for you.
- `fine` subtracts the due date from the return date. Returned 1 March, that is 2 days **before** the due date, so `.days` is -2; `max(0, -2)` (Day 6) turns an early return into 0 days late. Returned 10 March, it is 7 days late: $7 \times 0.25 = 1.75$.
- `LOAN_DAYS` and `FINE_PER_DAY` are constants (Day 19), so a rule change is a one-line edit.

Notice that neither function asks the computer what day it is. A real library program would call `fine(borrowed, date.today())`, with `date.today()` from Day 13. Passing the date in as a parameter, instead of calling `date.today()` inside the function, keeps the function **repeatable**: it gives the same answer for the same arguments on any day, so the examples on this page print the same output every time, and a test (Day 20) can check a fine for any return date without waiting for that date to come. The same goes for `datetime.now()`.

Dates and datetimes can also be **compared** with `<`, `>`, and `==`: earlier is smaller. And a `datetime` can be split into its `date()` and its `time()`:

shift_length.py

```

1 from datetime import datetime
2
3 start = datetime(2026, 3, 6, 22, 30)
4 end = datetime(2026, 3, 7, 6, 15)
5 shift = end - start
6 print(shift)
7 print(shift.total_seconds() / 3600, "hours")
8 print(end > start, end.date(), end.time())

```

```

7:45:00
7.75 hours
True 2026-03-07 06:15:00

```

A night shift from 10:30 p.m. to 6:15 a.m. the next morning is 7 hours 45 minutes; subtracting the datetimes handles the midnight in between. Printing a `timedelta` shows hours, minutes, and seconds. `total_seconds()` gives the whole length in seconds, as a float, so dividing by 3600 (seconds in an hour) gives hours: $27,900 / 3600 = 7.75$.

KEY IDEA

Date arithmetic is addition and subtraction: `date + timedelta` is a date, and `date - date` is a `timedelta`. Use `.days` for whole days and `total_seconds()` for an exact length. Pass "today" into functions as a parameter so they stay repeatable and testable.

WATCH OUT

Reading `.seconds` as the whole length. A `timedelta` stores days and seconds **separately**:

```

>>> from datetime import timedelta
>>> trip = timedelta(days=1, hours=2)
>>> trip
datetime.timedelta(days=1, seconds=7200)
>>> trip.seconds
7200
>>> trip.total_seconds()
93600.0
>>> trip.total_seconds() / 3600
26.0

```

A trip of 1 day and 2 hours has `.seconds` of 7200, only the 2 hours left over after the whole day. `total_seconds()` gives all 93,600 seconds (86,400 for the day plus 7200), which is 26 hours. Use `total_seconds()` whenever you want the full length in seconds, minutes, or hours.

PAUSE AND TRY

Question 7 Tracing

What does this program print? 2028 is a leap year.

dates_trace.py

```

1 from datetime import date, datetime, timedelta
2
3 print(date(2028, 2, 27) + timedelta(days=3))
4 print(datetime(2026, 5, 1, 23, 50) + timedelta(minutes=25))
5 print((date(2026, 12, 25) - date(2026, 12, 1)).days)

```

21.8 Formatting and Parsing Dates

People write dates in many ways: 11/05/2026, May 11, 2026, 2026-05-11. A program needs two conversions. **Formatting** turns a date into text in the layout you choose. **Parsing** does the reverse: it reads text and builds a date from it.

Day 13 introduced `strftime` ("string from time") for formatting. Its codes, each starting with %, stand for parts of a date or time:

Code	Means	Example
%Y	Year, four digits	2026
%m	Month as a number, 01–12	05
%d	Day of the month, 01–31	11
%B	Month name	May
%b	Short month name	Mar
%A	Weekday name	Monday
%a	Short weekday name	Mon
%H	Hour, 00–23	08
%I	Hour, 01–12	08
%p	AM or PM	AM
%M	Minute, 00–59	30
%S	Second, 00–59	00

```

>>> from datetime import date, datetime
>>> exam = datetime(2026, 5, 11, 8, 0)
>>> exam.strftime("%A %d %B %Y")
'Monday 11 May 2026'
>>> exam.strftime("%a %b %d, %I:%M %p")
'Mon May 11, 08:00 AM'
>>> f"Exam day: {exam:%d/%m/%Y}"
'Exam day: 11/05/2026'
>>> exam.isoformat()
'2026-05-11T08:00:00'
>>> date.fromisoformat("2026-05-11")
datetime.date(2026, 5, 11)
>>> datetime.strptime("11/05/2026 08:00", "%d/%m/%Y %H:%M")
datetime.datetime(2026, 5, 11, 8, 0)

```

- `strftime` keeps every character that is not a code, such as spaces, commas, and slashes, so you build the layout around the codes.
- An f-string can take the same codes after a colon, `{exam:%d/%m/%Y}`, just like the `:.2f` format specifier from Day 3.
- `isoformat()` gives the international standard layout, **ISO 8601**: year, month, day, from biggest to smallest, with a `T` before any time. It is the safest layout for files, because it means the same thing in every country, and ISO dates sort correctly as plain text.
- `date.fromisoformat(text)` parses an ISO date. `datetime.strptime(text, format)` ("string parse time") parses any layout, given the codes that describe it, and returns a `datetime`.

WORKED EXAMPLE Reading US-style dates

Sign-up forms from the United States write dates as month/day/year. This function parses them, and gives back `None` for text that is not such a date, so its return hint is `date | None` (Day 19):

`parse_dates.py`

```
1 from datetime import date, datetime
2
3
4 def read_us_date(text: str) -> date | None:
5     """Return the month/day/year date in text, or None if there is none."""
6     try:
7         return datetime.strptime(text, "%m/%d/%Y").date()
8     except ValueError:
9         return None
10
11
12 for text in ["3/14/2026", "12/1/2026", "14/3/2026", "2/30/2026"]:
13     when = read_us_date(text)
14     if when is None:
15         print(text, "is not a month/day/year date")
16     else:
17         print(text, "is", when.isoformat(), f"({when:%A})")
```

```
3/14/2026 is 2026-03-14 (Saturday)
12/1/2026 is 2026-12-01 (Tuesday)
14/3/2026 is not a month/day/year date
2/30/2026 is not a month/day/year date
```

- Line 7 parses with `"%m/%d/%Y"` and then calls `.date()` to keep only the day, since the text has no time. `strptime` accepts 3 as well as 03 for `%m`.
- `14/3/2026` fails because there is no month 14, and `2/30/2026` fails because February has no 30th. In both cases `strptime` raises `ValueError`, and lines 8 and 9 catch it (Day 15) and return `None`.
- Line 17 formats with `{when:%A}` to show the weekday name.

Without the `try`, a bad date stops the program with a message that shows both the text and the format:

bad_date.py

```
1 from datetime import datetime
2
3 when = datetime.strptime("14/3/2026", "%m/%d/%Y")
4 print(when)
```

```
ValueError: time data '14/3/2026' does not match format '%m/%d/%Y'
```

KEY IDEA

Format a date with `strftime(codes)` or `f"{when:codes}"`; parse text with `datetime.strptime(text, codes)` or `date.fromisoformat(text)`. Parsing raises `ValueError` for text that does not fit, so catch it when the text comes from a person.

WATCH OUT

Mixing up %m and %M. Lower-case %m is the month; capital %M is the minute. This mistake does not always raise an error:

month_minutes.py

```
1 from datetime import datetime
2
3 when = datetime.strptime("03/14/2026", "%M/%d/%Y")
4 print(when)
```

```
2026-01-14 00:03:00
```

The 03 was read as 3 minutes. With no month in the format, `strptime` used January. The result looks like a real date, so a test with a known answer (Day 20) is the only way to catch it. Check the table: dates use %m, times use %M.

PAUSE AND TRY**Question 8** Multiple choice

Which format turns the text "14 Mar 2026 09:30" into a `datetime` with `datetime.strptime`?

- A. "%d %b %Y %H:%M"
- B. "%d %B %Y %H:%M"
- C. "%d %m %Y %H:%M"
- D. "%d %b %Y %H:%m"

21.9 Regular Expressions and re.search

Day 3's string methods find **exact** text: `note.find("555-0142")` finds that one phone number. But what if you want **any** phone number, any order code, or any email address? You know what they look like (three digits, a dash, four digits) but not what they say. A **regular expression**, or **regex**, is a pattern that describes what text looks like. The `re` module searches text for patterns.

A pattern is a string in which most characters match themselves, and a few have special meanings:

Piece	Matches	Example	Matches in text
a letter, digit	Itself	cat	cat
.	Any one character except a line break	c.t	cat, cut, c9t
\d	One digit, 0–9	\d\d	42
\w	One "word" character: a letter, a digit, or <code>_</code>	\w\w\w	a_2
\s	One whitespace character: a space, a tab, a line break	a\s b	a b
[...]	One of the characters listed, or a range such as a–z	gr[ae]y	grey, gray
[^...]	One character that is not listed	[^0-9]	x, -
+	One or more of the piece before it	\d+	7, 2026
*	Zero or more of the piece before it	ab*	a, abbb
?	Zero or one of the piece before it (optional)	colou?r	color, colour
{3}, {1,2}	Exactly 3, or from 1 to 2, of the piece before it	\d{4}	2026
^, \$	The start and the end of the text	^Hi	Hi only at the start
\b	A word boundary: the edge between a word and a non-word	\bcat\b	cat, not catalog
\., \\$, ...	A backslash makes a special character plain	\\$\d+	\$20
(...)	A group : part of the match you want back on its own	(\d+)-(\d+)	555-0142
A B	A or B	GET POST	GET, POST

So `\d{3}-\d{4}` means "three digits, a dash, four digits", which matches `555-0142` and any other number of that shape.

Patterns use many backslashes, and Day 3 taught that a backslash in a string starts an escape character: `"\n"` is a line break, and `"\b"` is a "backspace" character, not `\b`. To keep every backslash exactly as typed, write patterns as **raw strings**: put an `r` before the opening quote. A raw string treats a backslash as an ordinary character.

```
>>> len("\b"), len(r"\b")
(1, 2)
>>> print(r"\d{3}-\d{4}")
\d{3}-\d{4}
```

`re.search(pattern, text)` looks for the first place in `text` that matches. It returns a **match object** describing it, or `None` if nothing matches:

```
>>> import re
>>> note = "Call Ana at 555-0142 before Friday."
>>> match = re.search(r"\d{3}-\d{4}", note)
>>> match
<re.Match object; span=(12, 20), match='555-0142'>
>>> match.group()
'555-0142'
>>> print(re.search(r"\d{3}-\d{4}", "Call Ana on Friday."))
None
```

The match object shows its **span**, where the match starts and stops (as slice positions, Day 3), and the text that matched. `group()` gives the matched text. Searching a note with no phone number gives `None`.

WORKED EXAMPLE Finding order codes

A shop's order codes are two capital letters, a dash, and four digits, such as `AB-1234`. Groups pull out the two parts:

`order_codes.py`

```
1 import re
2
3 messages = [
4     "Hi, my order AB-1234 has not arrived.",
5     "Order number zz-9999, thanks",
6     "Where is CD-0042? It was due Monday.",
7 ]
8 for message in messages:
9     match = re.search(r"([A-Z]{2})-(\d{4})", message)
10    if match:
11        store = match.group(1)
12        number = int(match.group(2))
13        print(match.group(), "is store", store, "order", number)
14    else:
15        print("No order code in:", message)
```

```
AB-1234 is store AB order 1234
No order code in: Order number zz-9999, thanks
CD-0042 is store CD order 42
```

- In `r"([A-Z]{2})-(\d{4})"`, `[A-Z]{2}` is two capital letters and `\d{4}` four digits. The parentheses make them groups 1 and 2.
- `match.group()` (or `group(0)`) is the whole match; `group(1)` and `group(2)` are the parts. Groups are always strings, so line 12 converts the number with `int()`: `"0042"` becomes 42.
- The second message has `zz-9999`. `[A-Z]` means capital letters only, so lower-case `zz` does not match, `search` returns `None`, and the `else` runs.
- `search` finds a match anywhere in the text, such as in the middle of a sentence.

KEY IDEA

A regular expression describes what text looks like. Write it as a raw string, `r"..."`. `re.search(pattern, text)` returns a match object for the first match, or `None`; `group()` is the matched text, and `group(1)` is the first group in parentheses.

WATCH OUT

Calling `group()` when there was no match. `re.search` returns `Match` | `None`, and `None` has no methods:

`no_match.py`

```
1 import re
2
3 message = "Order number zz-9999, thanks"
4 code = re.search(r"[A-Z]{2}-\d{4}", message).group()
5 print(code)
```

```
AttributeError: 'NoneType' object has no attribute 'group'
```

The pattern wants capital letters, so it does not match `zz-9999`, and line 4 calls `.group()` on `None`. Treat a search result like any other `X` | `None` value (Day 19): check `if match:` before using it, as `order_codes.py` does.

PAUSE AND TRY**Question 9** Multiple choice

Which pattern finds a time such as 9:05 or 14:30 in the middle of a sentence?

- A. `r"\d:\d\d"`
- B. `r"\d{1,2}:\d{2}"`
- C. `r"\d+.\d+"`
- D. `r"^\d{2}:\d{2}$"`

21.10 Finding and Replacing: `findall` and `sub`

`search` stops at the first match. Two more functions handle every match at once:

- `re.findall(pattern, text)` returns a list of every match, as strings. If the pattern has one group, the list holds just that group's text from each match. (With several groups, it holds a tuple of groups for each match.)
- `re.sub(pattern, replacement, text)` ("substitute") returns a new string in which every match is replaced. Strings are immutable (Day 3), so the original text is unchanged.

WORKED EXAMPLE Adding up payments in a note

payments.py

```

1 import re
2
3 note = "Paid $12.50 on 3 March, $8.75 on 5 March and $20.00 on 9 March."
4
5 print(re.findall(r"\$\d+\.\d{2}", note))
6 amounts = re.findall(r"\$(\d+\.\d{2})", note)
7 print(amounts)
8 print(f"Total: ${sum(float(amount) for amount in amounts):.2f}")

```

```

['$12.50', '$8.75', '$20.00']
['12.50', '8.75', '20.00']
Total: $41.25

```

- `\$` is a real dollar sign (a plain `$` would mean "end of text"), `\d+` one or more digits, `\.` a real dot, and `\d{2}` the cents. The day numbers 3, 5, and 9 have no `$` in front, so they are not matched.
- Line 6 puts the amount in a group, so `findall` returns only the numbers, without the `$`, ready for `float()`.
- Line 8 adds them with `sum` and a generator expression (Day 11): $12.50 + 8.75 + 20.00 = 41.25$.

`sub` cleans and rewrites text. In the replacement, `\1`, `\2`, and so on put back what group 1, group 2, and so on matched, so a raw string is needed there too. And an optional `flags=re.IGNORECASE` argument makes a pattern ignore the difference between capital and lower-case letters:

```

>>> import re
>>> re.sub(r"\s+", " ", "too many spaces")
'too many spaces'
>>> re.sub(r"\d{4} ", "#### ", "Card 4111 2222 3333 4444")
'Card #### #### #### 4444'
>>> re.sub(r"(\d{2})/(\d{2})/(\d{4})", r"\3-\2-\1", "Due 14/03/2026")
'Due 2026-03-14'
>>> re.findall(r"cat", "Cat, cat, CAT, catalog", flags=re.IGNORECASE)
['Cat', 'cat', 'CAT', 'cat']
>>> re.findall(r"\bcat\b", "Cat, cat, CAT, catalog", flags=re.IGNORECASE)
['Cat', 'cat', 'CAT']

```

- `\s+` matches any run of whitespace, so every run becomes one space.
- `\d{4}` (four digits **and** a space) matches the first three blocks of the card number, but not the last, which has no space after it.
- The date pattern has three groups, and `r"\3-\2-\1"` writes them back in the opposite order, turning a day/month/year date into an ISO date.
- With `IGNORECASE`, `cat` matches `Cat` and `CAT`, and also the `cat` inside `catalog`. Adding `\b` on each side keeps only whole words.

KEY IDEA

`re.findall` returns a list of every match, or of each match's group when the pattern has one group. `re.sub` replaces every match, and `\1` in the replacement puts back group 1. `flags=re.IGNORECASE` ignores case.

WATCH OUT

+ and * take as much as they can. They are **greedy**: they match the longest text that still lets the pattern succeed. To find quoted words:

```
>>> import re
>>> line = 'She said "hello" and then "goodbye".'
>>> re.findall(r'".+"', line)
['"hello" and then "goodbye"']
>>> re.findall(r'"[^"]+"', line)
['"hello"', '"goodbye"']
```

`".+"` starts at the first quote, and `."+` runs all the way to the **last** quote, swallowing the text in between. `"[^"]+"` says "a quote, then one or more characters that are not quotes, then a quote", so each match must stop at the next quote. (The patterns contain double quotes, so these strings use single quotes, which PEP 8 allows when it avoids backslashes.)

PAUSE AND TRY**Question 10** Tracing

What does this program print?

hashtags.py

```
1 import re
2
3 post = "Loving #Python and #regex! Day #21 of 25. #"
4 tags = re.findall(r"#(\w+)", post)
5 print(tags)
6 print(re.sub(r"#\w+", "[tag]", post))
```

21.11 Command-Line Arguments with argparse

Day 13 read **command-line arguments** from `sys.argv`: `python greet_args.py Ana` put "Ana" at `sys.argv[1]`. That is fine for one argument. But real commands take options in any order, such as `--times 2` or `--shout`, convert numbers, fill in defaults, explain how to use them, and print a clear message when someone types them wrongly. Doing all that by hand with `sys.argv` would take dozens of lines. The `argparse` module does it for you: you describe the arguments, and it reads `sys.argv` for you.

WORKED EXAMPLE A greeting command

greet/greet.py

```
1 import argparse
2
3 parser = argparse.ArgumentParser(description="Greet someone by name.")
4 parser.add_argument("name", help="who to greet")
5 parser.add_argument("-t", "--times", type=int, default=1, help="how often")
6 parser.add_argument("--shout", action="store_true", help="greet in capitals")
7 args = parser.parse_args()
8
9 greeting = f"Hello, {args.name}!"
10 if args.shout:
11     greeting = greeting.upper()
12 for _ in range(args.times):
13     print(greeting)
```

Run with `python greet.py Ana --times 2 --shout`:

```
HELLO, ANA!
HELLO, ANA!
```

- Line 3 makes an `ArgumentParser`, the object that reads the arguments. Its `description` appears in the help.
- Line 4 adds a **positional argument**, `name`: one the user must give, identified by its position, not by a label.
- Line 5 adds an **option**, an argument that starts with dashes and is optional. It has a short name, `-t`, and a long one, `--times`. `type=int` converts the text to an int (every argument arrives as a string, as with `input()`), and `default=1` is used when the option is left out.
- Line 6: `action="store_true"` makes `--shout` a **flag**, an option with no value after it. It is `True` if the user types it and `False` if not.
- Line 7: `parse_args()` reads `sys.argv`, checks it, and returns an object with one attribute per argument: `args.name`, `args.times`, `args.shout`. The attribute names come from the long names, without the dashes.

Options can come in any order, before or after the name: `python greet.py --shout Ben -t 2` prints `HELLO, BEN!` twice, and `python greet.py Ana` uses both defaults and prints `Hello, Ana!` once.

You also get a **help message** for free. Run `python greet.py --help` (or `-h`):

```
usage: greet.py [-h] [-t TIMES] [--shout] name

Greet someone by name.

positional arguments:
  name                who to greet

options:
  -h, --help          show this help message and exit
  -t, --times TIMES    how often
  --shout              greet in capitals
```

The first line is the **usage** summary: square brackets mark what is optional, and `name` is required. Each `help=` text from lines 4 to 6 appears beside its argument. This is Python 3.14's layout; older versions may differ a little, for example saying `optional arguments:` instead of `options:`.

When the arguments are wrong, `argparse` prints the usage line and an error, and stops the program before your code runs. Run with no name at all:

```
usage: greet.py [-h] [-t TIMES] [--shout] name
greet.py: error: the following arguments are required: name
```

Typing `-t two` stops it the same way, with a message that `'two'` is not a valid int. Compare Day 13's `greet_args.py`, which had to check `len(sys.argv)` itself.

`parse_args()` can also take a list of strings in place of the real command line, as in `parser.parse_args(["Ana", "--times", "3"])`. That is how you try a parser out, or test it (Day 20), without typing commands.

KEY IDEA

`argparse` reads command-line arguments: `add_argument("name")` for a required positional argument, `add_argument("--option", type=int, default=...)` for an option, and `action="store_true"` for a flag. `parse_args()` returns an object with one attribute per argument, and `--help` and error messages come free.

WATCH OUT

Dashes in an option's name. An option called `--max-price` becomes the attribute `max_price`, with an underscore, because a name cannot contain a dash. Writing the dash reads as a subtraction:

`dashes.py`

```
1 import argparse
2
3 parser = argparse.ArgumentParser()
4 parser.add_argument("--max-price", type=float, default=50.0)
5 args = parser.parse_args(["--max-price", "20"])
6 print(args)
7 print(args.max-price)
```

```
Namespace(max_price=20.0)
AttributeError: 'Namespace' object has no attribute 'max'
```

The first line shows what `parse_args` returned: a `Namespace` object, holding `max_price`. `args.max-price` means `args.max` minus `price`, and there is no attribute `max`. Write `args.max_price`.

PAUSE AND TRY

Question 11 Multiple choice

Using `greet.py`, which command prints `HELLO, BEN!` exactly twice?

- A. `python greet.py --name Ben --shout -t 2`
- B. `python greet.py Ben shout 2`
- C. `python greet.py -t 2 --shout Ben`
- D. `python greet.py Ben -t 2`

21.12 Look Before You Write

Every module today replaced code you could have written yourself, and every one of them has been written, tested by thousands of programmers, and fixed when someone found an edge case. Code you do not write is code you do not have to test or debug. So before you write a utility, spend two minutes looking for it:

- Read the list of modules in the Python Standard Library documentation. Even skimming the names teaches you what exists.
- Search the web for "python standard library" and the job, such as "python standard library median".
- In the REPL, `help(module)` shows a module's documentation, and `dir(module)` lists its names (Day 13).

Some small finds that replace earlier days' practice:

```
>>> import calendar
>>> calendar.isleap(2024), calendar.isleap(2100)
(True, False)
>>> import statistics
>>> scores = [72, 88, 95, 88]
>>> statistics.mean(scores), statistics.median(scores), statistics.mode(scores)
(85.75, 88.0, 88)
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

`calendar.isleap` is Day 4's leap year rule, including the rule that 2100 is not a leap year.

`statistics` has the averages: the **mean** adds the scores and divides ($343 / 4 = 85.75$), the **median**

is the middle value of the sorted scores (halfway between 88 and 88), and the **mode** is the most common value. `string` has ready-made strings of letters, digits, and punctuation.

When you need to	Look at
Count things, or group them into a dictionary	<code>collections: Counter, defaultdict</code>
A queue, or "the last N items"	<code>collections: deque</code>
Pairs, arrangements, joined or grouped loops	<code>itertools</code>
Dates, times, and differences	<code>datetime, calendar</code>
Averages and spread	<code>statistics</code>
Find or change patterns in text	<code>re</code>
Command-line arguments	<code>argparse</code>
Files, CSV, and JSON	<code>pathlib, csv, json</code> (Day 14)

Writing a utility yourself is still worth doing **once**, to learn how it works, as you did on Days 4 and 8. In a real program, reach for the library.

KEY IDEA

Before writing a utility, look for it in the standard library: skim the module list, search for the job, or use `help()`. Library code is already tested, so you have less code to test and debug.

PAUSE AND TRY

Question 12 Multiple choice

A program reads a sensor every minute and must always know the average of the **last 60** readings. Which tools fit best?

- A. A list with `insert(0, reading)` and `pop()`, and a loop to add up the readings
- B. A `deque(maxlen=60)` and `statistics.mean`
- C. A `Counter` of the readings and `most_common(60)`
- D. `combinations(readings, 60)`

Putting it together

A web server writes one line per visit: the date and time, the kind of request, the page, and a status code (200 means it worked; 400 or more means an error). This command summarizes a log file. It uses one tool from each of today's modules:

`log_report.py`

```
1 """Summarize a web server's visit log."""
2
3 import argparse
4 import re
5 from collections import Counter, deque, namedtuple
6 from datetime import datetime
7 from itertools import groupby
```

```

8
9 Visit = namedtuple("Visit", ["when", "page", "status"])
10 LINE_PATTERN = r"^(\S+ \S+) (GET|POST) (\S+) (\d{3})$"
11
12
13 def read_visits(path: str) -> list[Visit]:
14     """Return the visits in a log file, skipping lines that are not visits."""
15     visits: list[Visit] = []
16     with open(path, encoding="utf-8") as file:
17         for line in file:
18             match = re.search(LINE_PATTERN, line.strip())
19             if match:
20                 when = datetime.strptime(match.group(1), "%Y-%m-%d %H:%M:%S")
21                 visits.append(Visit(when, match.group(3), int(match.group(4))))
22     return visits
23
24
25 def main() -> None:
26     parser = argparse.ArgumentParser(description="Summarize a visit log.")
27     parser.add_argument("logfile", help="the log file to read")
28     parser.add_argument("--top", type=int, default=3, help="pages to list")
29     args = parser.parse_args()
30
31     visits = read_visits(args.logfile)
32     span = visits[-1].when - visits[0].when
33     hours = span.seconds // 3600
34     print(f"{len(visits)} visits over {span.days} days and {hours} hours")
35
36     for day, group in groupby(visits, key=lambda visit: visit.when.date()):
37         print(f"{day:%a %d %b}: {len(list(group))} visits")
38
39     pages = Counter(visit.page for visit in visits)
40     for page, count in pages.most_common(args.top):
41         print(page, count)
42
43     errors: deque[Visit] = deque(maxlen=2)
44     for visit in visits:
45         if visit.status >= 400:
46             errors.append(visit)
47     for visit in errors:
48         print(f"Error {visit.status} on {visit.page} at {visit.when:%H:%M}")
49
50
51 if __name__ == "__main__":
52     main()

```

The log file, `visits.txt`, in the same folder:

```

2026-03-02 09:15:02 GET /home 200
2026-03-02 09:15:40 GET /menu 200
2026-03-02 11:02:13 POST /order 500
2026-03-03 08:30:00 GET /home 200
2026-03-03 08:31:12 GET /menu 200
2026-03-03 08:33:57 GET /menus 404
this line is not a visit
2026-03-04 18:45:30 GET /home 200
2026-03-04 18:47:05 POST /order 200

```

Run with `python log_report.py visits.txt --top 2`:

```

8 visits over 2 days and 9 hours
Mon 02 Mar: 3 visits
Tue 03 Mar: 3 visits
Wed 04 Mar: 2 visits
/home 3
/menu 2
Error 500 on /order at 11:02
Error 404 on /menus at 08:33

```

Lines	What to notice	Idea
9	Each visit is a small record that never changes: a <code>namedtuple</code>	21.3
10	<code>^</code> and <code>\$</code> make the pattern match a whole line; four groups capture the time, the request, the page, and the code	21.9
18–21	<code>search</code> returns <code>None</code> for the line that is not a visit, so <code>if match:</code> skips it	21.9
20	<code>strptime</code> parses the time, with <code>%M</code> for minutes and <code>%m</code> for the month	21.8
26–29	<code>argparse</code> reads the file name and <code>--top</code> ; <code>type=int</code> makes <code>args.top</code> the int 2	21.11
32–34	The last visit minus the first is 2 days, 9 hours, and 32 minutes; <code>.seconds</code> is right here , for the part after the whole days	21.7
36–37	The log is already in time order, so <code>groupby</code> on each visit's date gives one run per day	21.6
39–41	<code>Counter</code> counts the pages, and <code>most_common(args.top)</code> lists the top 2	21.1
43–48	A <code>deque(maxlen=2)</code> keeps only the two most recent errors	21.4

`span.seconds // 3600` is 34,323 seconds (9 hours, 32 minutes, 3 seconds) divided by 3600 and rounded down: 9 hours. The line `this line is not a visit` has no time, so the pattern does not match it, and the file's eight real visits are all counted. Without the standard library, this program would need a hand-written counting loop, a grouping loop, a date parser that knows each month's length, string slicing to pull each line apart, and code to read `sys.argv` and print a usage message.

Common mistakes

Use this table to go from a message to its usual cause. Older versions of Python may word some messages differently.

What you see	Usual cause	Fix
--------------	-------------	-----

A Counter of single letters and ' '	Counting a string instead of its words	Counter(text.split())
TypeError: first argument must be callable or None	defaultdict([]) or defaultdict(0): a value, not a function	defaultdict(list), defaultdict(int)
A defaultdict has keys nobody added	Reading a missing key creates it	Check with in or get()
AttributeError: can't set attribute	Assigning to a namedtuple field	record = record._replace(field=value)
IndexError: pop from an empty deque	popleft() or pop() on an empty deque	Loop while queue:
A loop over combinations, chain, or groupby does nothing	The iterator was already used up	Make it again, or store list(...) once
groupby repeats a key	The data was not sorted by that key	Sort by the same key first
A length in hours is too small	.seconds leaves out the whole days	total_seconds()
ValueError: time data ... does not match format ...	The text and the format codes differ	Fix the codes, or catch ValueError
AttributeError: 'NoneType' object has no attribute 'group'	re.search found no match	if match: before match.group()
A regex match runs too far	+ and * are greedy	Use [^...] to stop at a character
error: the following arguments are required	A positional argument was left out	Type it, or make it an option with a default
TypeError: 'str' object cannot be interpreted as an integer	An argparse option without type=int	Add type=int

Vocabulary

Term	Definition	Example
standard library	The collection of modules that comes with every Python installation	collections, re
collections	A standard module of specialized collection types	from collections import Counter
Counter	A dictionary that counts items; a missing item counts as 0	Counter("banana")["a"] is 3
most_common(n)	A Counter method giving the n most common (item, count) pairs	[('pizza', 3), ('tacos', 2)]
defaultdict	A dictionary that creates a starting value when a missing key is used	defaultdict(list)

default factory	The function a <code>defaultdict</code> calls to make a new key's starting value	<code>int, list</code>
namedtuple	A tuple type whose positions also have names	<code>Student("Ana", 10, 92).score</code>
deque	A double-ended queue: fast to add and remove at both ends	<code>deque(["Ana", "Ben"]).popleft()</code>
queue	A first in, first out line: join at the back, leave from the front	A help desk line
maxlen	The largest size of a <code>deque</code> ; a full one drops from the opposite end	<code>deque(maxlen=3)</code>
itertools	A standard module of building blocks for loops	<code>from itertools import chain</code>
chain	Gives the items of several iterables, one after another	<code>chain([1, 2], (3,))</code>
combinations	Every way to choose <code>r</code> items, order not mattering	<code>combinations("ABC", 2)</code>
permutations	Every way to arrange <code>r</code> items, order mattering	<code>permutations("ABC", 2)</code>
groupby	Splits an iterable into runs of neighbors with the same key	<code>groupby("aaab")</code>
run	Neighboring items that share the same key	<code>aaa in aaab</code>
total_seconds()	The whole length of a <code>timedelta</code> , in seconds	<code>timedelta(days=1).total_seconds()</code>
formatting	Turning a date or time into text in a chosen layout	<code>when.strftime("%d/%m/%Y")</code>
parsing	Reading text and building a value, such as a date, from it	<code>datetime.strptime(text, "%m/%d/%Y")</code>
strptime	Parses text into a <code>datetime</code> , given format codes	<code>datetime.strptime("3/14/2026", ...)</code>
ISO 8601	The international date layout, year-month-day	<code>2026-03-14</code>
regular expression	A pattern that describes what text looks like (also regex)	<code>\d{3}-\d{4}</code>
re	The standard module for regular expressions	<code>re.search(pattern, text)</code>
raw string	A string literal, written with <code>r</code> before it, in which backslashes are ordinary characters	<code>r"\d+"</code>
match object	What <code>re.search</code> returns for a match; <code>group()</code> gives the matched text	<code>match.group(1)</code>
group	Part of a pattern in parentheses, whose matched text you can get back on its own	<code>(\d+)</code>

greedy	Matching as much text as possible, as <code>+</code> and <code>*</code> do	<code>".+"</code>
argparse	The standard module for reading command-line arguments	<code>argparse.ArgumentParser()</code>
positional argument	A required command-line argument identified by its position	<code>name</code> in <code>greet.py Ana</code>
option	An optional command-line argument that starts with dashes	<code>--times 2</code>
flag	An option with no value, <code>True</code> when typed	<code>--shout</code>
usage	The one-line summary of how to run a command	<code>usage: greet.py [-h] ... name</code>

Summary

- The **standard library** comes with Python. Before writing a utility, look for it there: library code is already written and tested.
- `collections` has `Counter` for counting (`most_common`, a missing item counts as 0), `defaultdict` for grouping and totals without checking for missing keys, `namedtuple` for small fixed records with named fields, and `deque` for queues that are fast at both ends, with `maxlen` to keep only the last few items.
- `itertools` has `chain` to join iterables, `combinations` to choose items where order does not matter, `permutations` to arrange them where it does, and `groupby` to split data into runs, after sorting it by the same key. All return lazy iterators.
- `datetime` does calendar arithmetic (`date - date` is a `timedelta`; use `total_seconds()` for exact lengths), formats with `strftime` codes, and parses with `strptime` or `fromisoformat`. Pass "today" into functions so they stay repeatable.
- A **regular expression** describes what text looks like. With `re`, `search` finds the first match (or `None`), `findall` lists all of them, and `sub` replaces them. `argparse` reads positional arguments, options, and flags, and writes the help and error messages for you.

Practice

Do these in your `day21` folder. Type each program yourself, run it, and only then open the answer.

Question 13 Short answer

Word count with Counter. Redo the Day 8 word count with `Counter`, and print the ten most common words with their counts. Read the text from a file, and let a regular expression split it into words so that punctuation is ignored.

Question 14 Short answer

Days until a date. Ask the user for a date, and print how many days away it is. Keep asking until the user types a real date. Write the day count as a function that takes "today" as a parameter.

Question 15 Short answer

Finding email addresses. Extract every email address from a block of text with a regular expression.

Review questions

Question 16 Multiple choice

You need to know, for each customer, the list of their orders, and the orders arrive in no particular order. Which is the shortest correct tool?

- A. `Counter(orders)`
- B. `defaultdict(list)`, appending each order to its customer's list
- C. `groupby(orders, key=...)` without sorting
- D. A `deque(maxlen=10)` per customer

Question 17 Debugging

This program should group names by first letter. Find the error and fix it.

`defaultdict_called.py`

```
1 from collections import defaultdict
2
3 by_letter = defaultdict([])
4 by_letter["A"].append("Ana")
5 print(by_letter)
```

Question 18 Tracing

What does this program print?

`sales_trace.py`

```
1 from collections import Counter, namedtuple
2 from itertools import groupby
3
4 Sale = namedtuple("Sale", ["day", "item"])
5 sales = [
6     Sale("Mon", "tea"),
7     Sale("Mon", "cake"),
8     Sale("Tue", "tea"),
9     Sale("Mon", "tea"),
10 ]
11 for day, group in groupby(sales, key=lambda sale: sale.day):
12     print(day, [sale.item for sale in group])
13 print(Counter(sale.item for sale in sales).most_common(1))
```

Question 19 Short answer

Write one line that prints a list of every time, such as 9:05 or 17:45, in this text, without the room number or anything else: "Doors 9:05, talks at 10:30 and 14:30, room 101. Ends 17:45."

Question 20 Debugging

This program should print the greeting three times. Find the error and fix it.

times_str.py

```

1 import argparse
2
3 parser = argparse.ArgumentParser()
4 parser.add_argument("name")
5 parser.add_argument("--times", default=1)
6 args = parser.parse_args(["Ana", "--times", "3"])
7 for _ in range(args.times):
8     print(f"Hello, {args.name}!")

```

Question 21 Short answer

A classmate writes `def is_overdue(due: date) -> bool:` and calls `date.today()` inside it. Explain why `def is_overdue(due: date, today: date) -> bool:` is the better design, and write its body.

Go further

The Python Tutorial's Brief Tour of the Standard Library and Part II show more modules in the same spirit. When you write regular expressions often, the Regular Expression HOWTO goes much further than today.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can count items with Counter and list the most common ones. Reread: 21.1 Counting with Counter
- ☐ I can group and total values with defaultdict, and explain why reading a key can add it. Reread: 21.2 Default Values with defaultdict
- ☐ I can make a small record with namedtuple and read its fields by name. Reread: 21.3 Records with namedtuple
- ☐ I can use a deque as a queue, and keep only the last few items with maxlen. Reread: 21.4 Queues with deque
- ☐ I can join iterables with chain, and choose between combinations and permutations. Reread: 21.5 Chaining and Choosing: chain, combinations, permutations
- ☐ I can split data into runs with groupby, sorting it first when needed. Reread: 21.6 Grouping Runs with groupby
- ☐ I can add and subtract dates and times, and pass "today" into a function. Reread: 21.7 Dates, Times, and Differences
- ☐ I can format a date with strftime and parse one with strptime. Reread: 21.8 Formatting and Parsing Dates
- ☐ I can write a regular expression and find a match with re.search. Reread: 21.9 Regular Expressions and re.search
- ☐ I can list every match with re.findall and replace matches with re.sub. Reread: 21.10 Finding and Replacing: findall and sub
- ☐ I can read positional arguments, options, and flags with argparse. Reread: 21.11 Command-Line Arguments with argparse
- ☐ I can look in the standard library before writing a utility myself. Reread: 21.12 Look Before You Write

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
[('i', 4), ('s', 4)]
2 0 4
[('i', 4)] 14
```

"mississippi" has m 1, i 4, s 4, and p 2, first seen in that order. i and s tie on 4, so `most_common(2)` lists i first, because it was counted first. Line 5: p is 2, z was never counted so it reads as 0, and there are 4 different letters (reading z did not add it). Line 6 counts the letters of "pop": p goes up to 4 and a new letter, o, gets 1. Now i, s, and p all have 4, and i still comes first. The total is $11 + 3 = 14$.

Reread *21.1 Counting with Counter* if your output differed.

Question 2

```
20.75 0.0
4 ['books', 'food', 'games', 'toys']
```

`float()` gives 0.0, so each new category starts at 0.0. food gets $12.5 + 8.25 = 20.75$; games gets 20.0 and books 9.99. Line 8 reads `spending["toys"]`, which does not exist, so it is created with 0.0 and printed. That lookup added a fourth key, so `len` is 4, and `sorted` lists the four keys alphabetically.

Reread *21.2 Default Values with defaultdict* if your output differed.

Question 3

C is false. A named tuple cannot be changed, so `p.x = 5` raises `AttributeError`.

- **A** is true: `p.x` is 3 by name, `p[1]` is 4 by position, and $3 + 4 = 7$.
- **B** is true: a named tuple is a tuple, so it equals a plain tuple with the same items in the same order.
- **D** is true: `_replace` builds a new record and does not touch the old one.

Reread *21.3 Records with namedtuple* if you chose differently.

Question 4

```
deque([100, 102, 103], maxlen=3)
100 103 1
```

Step	orders afterward	Why
append 101 to 103	[101, 102, 103]	Room for three
append 104	[102, 103, 104]	Full: adding on the right drops 101
appendleft 100	[100, 102, 103]	Full: adding on the left drops 104

Printing a `deque` with a `maxlen` shows it. Then `popleft()` removes 100 from the front, `pop()` removes 103 from the back, and one item, 102, is left.

Reread 21.4 *Queues with deque* if your output differed.

Question 5

```
4
('ham', 'egg', 'kale')
12
['a', 'b', 1, 2]
```

Choosing 3 of 4 toppings is the same as choosing which 1 to leave out, so there are 4 combinations. The first follows the list's order: the first three items. Arranging 2 of 4 gives $4 \times 3 = 12$, because order matters. `chain` gives the characters of "ab" and then the items of the list, as one list of four.

Reread 21.5 *Chaining and Choosing: chain, combinations, permutations* if your output differed.

Question 6

```
a3b1c2a2
4
```

The runs of "aaabccaa" are aaa, b, cc, and aa. For each run, the f-string writes the character and the run's length, and `join` (Day 3) glues the pieces together: a3, b1, c2, a2. The final aa is a separate run from the first aaa, because a run is only neighbors. In "xyyyxxz", the runs have lengths 2, 4, 1, and 1, and `max` (Day 11, with a generator expression) picks 4.

Reread 21.6 *Grouping Runs with groupby* if your output differed.

Question 7

```
2028-03-01
2026-05-02 00:15:00
24
```

27 February 2028 plus 3 days: 28 February, 29 February (a leap year), then 1 March. 23:50 plus 25 minutes passes midnight, so the date moves on to 2 May, at 00:15. A `datetime` prints with seconds. From 1 to 25 December is 24 days: subtraction counts the gap, not both ends.

Reread 21.7 *Dates, Times, and Differences* if your output differed.

Question 8

A. `%d` reads 14, `%b` the short month name Mar, `%Y` 2026, and `%H:%M` the hour and minutes, with the spaces and colon matching the text exactly.

- **B** is wrong: `%B` wants the full month name, March, so `Mar` raises `ValueError`.
- **C** is wrong: `%m` wants a month number, such as 03, not a name.
- **D** is wrong: `%m` at the end means a month, not minutes, so the format cannot match and `strptime` raises `ValueError`.

Reread 21.8 *Formatting and Parsing Dates* if you chose differently.

Question 9

B. One or two digits for the hour, a colon, and exactly two digits for the minutes.

- **A** is wrong: `\d` is exactly one digit, so in `14:30` it matches only `4:30`, missing the `1`.
- **C** is wrong: `.` matches **any** character, so it also matches `3.5` or `12a45`. To match a real dot you would write `\.`, but a time needs a colon anyway.
- **D** is wrong: `^` and `$` mean the start and end of the text, so it matches only if the whole text is a time. It also needs exactly two hour digits, so it misses `9:05`.

Reread *21.9 Regular Expressions and re.search* if you chose differently.

Question 10

```
['Python', 'regex', '21']
Loving [tag] and [tag]! Day [tag] of 25. #
```

`#(\w+)` needs a `#` followed by at least one word character, and the group keeps only the word, so `findall` gives the three words without their `#`. `\w` includes digits, so `#21` counts. The last `#` has nothing after it, so it does not match. `sub` has no group in its pattern; it replaces each whole match, `#` included, with `[tag]`, and leaves the lone `#` alone.

Reread *21.10 Finding and Replacing: findall and sub* if your output differed.

Question 11

C. Options can come in any order: `-t 2` sets `times` to `2`, `--shout` sets `shout` to `True`, and Ben fills the positional `name`.

- **A** is wrong: there is no `--name` option. `name` is positional, so it takes no label, and `argparse` stops with an error about the unrecognized argument.
- **B** is wrong: without dashes, `shout` and `2` are just extra positional words, and the parser has room for only one, so it stops with an error.
- **D** is wrong: it prints `Hello, Ben!` twice, without capitals, because `--shout` was left out and a flag defaults to `False`.

Reread *21.11 Command-Line Arguments with argparse* if you chose differently.

Question 12

B. `append` on a deque with `maxlen=60` keeps exactly the last 60 readings, dropping the oldest automatically, and `statistics.mean` averages them.

- **A** is wrong: it works, but `insert(0, ...)` is the slow end of a list, and the adding-up loop is code the library already has.
- **C** is wrong: a `Counter` counts how often each value appears, and forgets the order, so it cannot tell which readings are the most recent.
- **D** is wrong: `combinations` makes every possible group of 60 readings, which is a huge number, and none of them is "the last 60".

Reread *21.4 Queues with deque* and *21.12 Look Before You Write* if you chose differently.

Question 13

One possible solution. Save a paragraph in `story.txt`, such as this one:

The garden was quiet in the morning. The bees came first, and then the birds. A bee does not stop for long: it visits a flower, takes what it needs, and moves on to the next flower. The birds were louder. They argued over the feeder, and the smallest bird always seemed to win. By noon the garden was warm, the bees were busy, and the birds had gone quiet in the shade. In the evening the garden belonged to the moths.

top_words.py

```
1 import re
2 from collections import Counter
3 from pathlib import Path
4
5
6 def top_words(text: str, how_many: int) -> list[tuple[str, int]]:
7     """Return the most common words in text, with their counts.
8
9     Words are compared in lower case, and punctuation is ignored.
10    """
11     words = re.findall(r"[a-z']+", text.lower())
12     return Counter(words).most_common(how_many)
13
14
15 text = Path("story.txt").read_text(encoding="utf-8")
16 for word, count in top_words(text, 10):
17     print(word, count)
```

```
the 15
and 4
garden 3
in 3
birds 3
to 3
was 2
quiet 2
bees 2
a 2
```

How it works:

- Line 11 does all of Day 8's cleaning in one step. `text.lower()` makes `The` and `the` the same word, and `findall(r"[a-z']+", ...)` returns every run of letters (and apostrophes, so a word like `don't` stays whole). Punctuation and spaces are never part of a match, so there is no `replace` for each punctuation mark.
- Line 12 counts the words and returns the top ten as `(word, count)` pairs, which is why the return hint is `list[tuple[str, int]]` (Day 19).
- Several words have a count of 2: `was`, `quiet`, `bees`, `a`, `it`, `flower`, and `were`. `most_common` keeps tied words in the order they first appeared, so the first four of them fill the last places in the top ten.
- Day 8 needed an empty dictionary, a counting loop, and a separate maximum loop; here the counting and ranking are one line.

Reread 21.1 Counting with Counter and 21.10 Finding and Replacing: findall and sub if your result differed.

Question 14

One possible solution:

days_until.py

```

1 from datetime import date
2
3 TODAY = date(2026, 10, 15)
4
5
6 def days_until(target: date, today: date) -> int:
7     """Return how many days from today until target (negative if past)."""
8     return (target - today).days
9
10
11 def ask_for_date(prompt: str) -> date:
12     """Keep asking until the user types a real date as YYYY-MM-DD."""
13     while True:
14         text = input(prompt)
15         try:
16             return date.fromisoformat(text)
17         except ValueError:
18             print(f"{text} is not a date like 2026-12-25. Try again.")
19
20
21 target = ask_for_date("Date (YYYY-MM-DD): ")
22 days = days_until(target, TODAY)
23 if days > 0:
24     print(f"{target:%A %d %B %Y} is {days} days away.")
25 elif days == 0:
26     print("That is today!")
27 else:
28     print(f"{target:%A %d %B %Y} was {-days} days ago.")

```

A run where the user first types two dates that do not exist:

```

Date (YYYY-MM-DD): next friday
next friday is not a date like 2026-12-25. Try again.
Date (YYYY-MM-DD): 2026-02-30
2026-02-30 is not a date like 2026-12-25. Try again.
Date (YYYY-MM-DD): 2026-12-25
Friday 25 December 2026 is 71 days away.

```

How it works:

- `days_until` subtracts the dates and returns `.days`: from 15 October to 25 December is $16 + 30 + 25 = 71$ days. Because `today` is a parameter, the function is repeatable and easy to test.
- `TODAY` is fixed at 15 October 2026 so that this page's output never changes. A real program would write `days_until(target, date.today())`, and its answer would change every day.
- `ask_for_date` is Day 15's "ask until valid" loop: `date.fromisoformat` raises `ValueError` for text that is not a date, including `2026-02-30`, so the `except` prints a message and the loop asks again. The `return` inside the `try` ends the loop once a date is read.

- The last lines handle three cases: a future date, today, and a past date, where `days` is negative and `-days` makes it positive. `{target:%A %d %B %Y}` formats the date with a weekday and month name.

Reread 21.7 *Dates, Times, and Differences* and 21.8 *Formatting and Parsing Dates* if your result differed.

Question 15

One possible solution:

find_emails.py

```
1 import re
2
3 EMAIL_PATTERN = r"[\w.+~]+@[~\w-]+\.[\w.-]*\w"
4
5
6 def find_emails(text: str) -> list[str]:
7     """Return every email address in text, in the order they appear."""
8     return re.findall(EMAIL_PATTERN, text)
9
10
11 notice = """Questions about the trip? Email ana.lee@example.com or
12 call the office. Tickets: tickets+trip@mail.example.org. Our old
13 address, info@example.net, still works. (Do not write to @school
14 or to someone@ with nothing after it.)"""
15
16 for email in find_emails(notice):
17     print(email)
```

```
ana.lee@example.com
tickets+trip@mail.example.org
info@example.net
```

How it works, piece by piece:

Piece	Matches
<code>[\w.+~]+</code>	The name before the @: letters, digits, <code>_</code> , dots, plus signs, and dashes
<code>@</code>	The @ itself
<code>[\w-]+</code>	The first part of the domain, such as <code>mail</code> or <code>example</code>
<code>\.</code>	A real dot
<code>[\w.-]*</code>	The rest of the domain, which may hold more dots, such as <code>example.org</code>
<code>\w</code>	The last character must be a letter or digit

- The final `\w` is why `tickets+trip@mail.example.org.` comes back without the full stop that ends its sentence: a match cannot end on a dot.
- `@school` has nothing before the @, and `someone@` has nothing after it, so neither matches.
- The pattern has no groups, so `findall` returns the whole matches. Inside `[...]`, a dot is a plain dot, and a dash at the end of the brackets is a plain dash.

Real email addresses can be stranger than this, so no short pattern catches every valid one. This pattern finds ordinary addresses, which is usually the job. Reread *21.9 Regular Expressions and re.search* and *21.10 Finding and Replacing: findall and sub* if your result differed.

Question 16

B. Grouping in any order is the job of `defaultdict(list)`:

`by_customer[order.customer].append(order)`, with no check for new customers.

- **A** is wrong: a `Counter` would tell you how many of each order there are, not which orders each customer made.
- **C** is wrong: `groupby` only joins neighbors, so unsorted orders give the same customer many separate runs. It works only after sorting by customer.
- **D** is wrong: `maxlen=10` would throw away all but the last ten orders, and you would still need a dictionary to find each customer's deque.

Reread *21.2 Default Values with defaultdict* and *21.6 Grouping Runs with groupby* if you chose differently.

Question 17

Python stops on line 3:

```
TypeError: first argument must be callable or None
```

`defaultdict` needs a **function** to call each time it makes a new key's starting value. `[]` is one list, not a function, so `defaultdict` refuses it. (If it accepted the list, every key would share that one list, the trap of Day 10's mutable default.) Pass the `list` type itself, which makes a fresh empty list on each call:

```
1 by_letter = defaultdict(list)
```

The same goes for counting: `defaultdict(int)`, never `defaultdict(0)`.

Reread *21.2 Default Values with defaultdict* if you missed it.

Question 18

```
Mon ['tea', 'cake']
Tue ['tea']
Mon ['tea']
[('tea', 3)]
```

`groupby` splits the sales into runs by `sale.day`: the first two are Monday, then one Tuesday, then Monday again. The data is not sorted by day, so Monday appears as two separate runs. The comprehension reads `sale.item` from each named tuple in a run. The `Counter` sees every sale, wherever it is, and counts tea 3 and cake 1, so `most_common(1)` is tea.

Reread *21.3 Records with namedtuple* and *21.6 Grouping Runs with groupby* if your output differed.

Question 19

One possible solution:

find_times.py

```

1 import re
2
3 schedule = "Doors 9:05, talks at 10:30 and 14:30, room 101. Ends 17:45."
4 print(re.findall(r"\b\d{1,2}:\d{2}\b", schedule))

```

```
['9:05', '10:30', '14:30', '17:45']
```

\d{1,2} is the hour (one or two digits), then a colon, and \d{2} the minutes. findall returns every match. The \b at each end makes the time a whole word, so a longer run of digits, such as 123:456, would not give a part of itself. 101 has no colon, so it never matches. A correct answer uses findall, a raw string, and allows one or two hour digits.

Reread 21.9 *Regular Expressions and re.search* and 21.10 *Finding and Replacing: findall and sub* if your answer differed.

Question 20

Python stops on line 7:

```
TypeError: 'str' object cannot be interpreted as an integer
```

Every command-line argument arrives as a string, so without type=int, args.times is the string "3", and range("3") is an error. (When the option is left out, args.times is the default, the int 1, so the bug hides until someone uses --times.) Add type=int:

times_str_fixed.py

```

1 import argparse
2
3 parser = argparse.ArgumentParser()
4 parser.add_argument("name")
5 parser.add_argument("--times", type=int, default=1)
6 args = parser.parse_args(["Ana", "--times", "3"])
7 for _ in range(args.times):
8     print(f"Hello, {args.name}!")

```

```

Hello, Ana!
Hello, Ana!
Hello, Ana!

```

Reread 21.11 *Command-Line Arguments with argparse* if you missed it.

Question 21

Model answer:

```

1 def is_overdue(due: date, today: date) -> bool:
2     """Return True if today is after the due date."""
3     return today > due

```

With date.today() inside, the answer depends on the day the code runs. A test (Day 20) that passes today could fail next month, and you could not test "one day late" or "due today" without waiting for those days. With today as a parameter, the function is repeatable:

`is_overdue(date(2026, 3, 3), date(2026, 3, 4))` is always `True`, and `is_overdue(date(2026, 3, 3), date(2026, 3, 3))` is always `False`. Only the top of the program calls `date.today()` and passes it in. A correct answer mentions repeatable tests and compares the dates with `>`.

Reread *21.7 Dates, Times, and Differences* if your reasoning differed.