

Day 8 Dictionaries

Self-Study Guide · About 70 minutes · With answer key

By the end you will be able to

- Model a real-world record, such as a contact or a product, as a dictionary of key-value pairs.
- Read, add, update, and delete entries, and handle a missing key with `in` or `get()` instead of a `KeyError`.
- Loop over a dictionary's keys, values, and items, including nested dictionaries and lists of dictionaries.
- Count and group values with a dictionary.

What to remember from earlier

- A list holds values in order, you read one by its index, and `append()` adds a value to the end. ([Day 6: Lists](#))
- `b = a` gives the same list a second name, so changing it through one name shows through the other; `a.copy()` makes a separate list. ([Day 6: Lists](#), [6.9 Aliasing Versus Copying](#))
- A tuple can never be changed, so it is hashable and can go in a set, but a list cannot. ([Day 7: Tuples, sets and unpacking](#), [7.7 Fast Membership Tests](#))
- Unpacking stores the parts of a pair in two names at once, as in `name, age = ("Ana", 15)`, and works in a `for` loop too. ([Day 7: Tuples, sets and unpacking](#), [7.3 Unpacking: Many Names at Once](#))
- An accumulator is set before a loop, updated inside it, and used after it. ([Day 5: Loops](#), [5.6 Counting and Summing](#))
- To find the largest value, start from `None` and replace it when highest is `None` or the new value is bigger. ([Day 5: Loops](#), [5.7 Finding the Largest Value](#))
- `split()` breaks a string into a list of words, and `join()` glues a list of strings together. ([Day 3: Strings](#), [3.7 Splitting and Joining](#))

Make a `day08` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days.

On Day 6 you stored many values in a list, and on Day 7 you met tuples and sets. All three find a value either by its **position** (index 0, 1, 2, ...) or not at all (a set has no positions). But much of the information in real life is looked up by **name**: the price of "tea", the phone number of "Ana", the capital of "Japan". Today you learn the collection Python made for exactly that: the **dictionary**.

8.1 Key-Value Pairs

Suppose a café wants to store its prices. With Day 6's tools you might keep two lists side by side:

```
1 items = ["tea", "coffee", "muffin"]
2 prices = [2.50, 3.00, 2.75]
```

To find the price of a muffin, you would search `items` for `"muffin"`, note its index (2), and then read `prices[2]`. That works, but it is fragile. Add an item to one list and forget the other, and every price after it belongs to the wrong item. And every lookup needs a search.

A paper dictionary solves the same problem for words: you look up a word and find its meaning next to it. You never ask for "the 4,512th word". Python's dictionary works the same way.

A **dictionary** (type `dict`) stores **key-value pairs**. The **key** is what you look things up by, such as `"muffin"`. The **value** is what is stored with it, such as `2.75`. Each pair is called an **entry**. You write a dictionary with curly braces, a colon between each key and its value, and commas between the entries:

```
1 prices = {"tea": 2.50, "coffee": 3.00, "muffin": 2.75}
```

- `{` and `}` start and end the dictionary.
- `"tea": 2.50` is one entry: the key `"tea"`, a colon, then its value `2.50`.
- Commas separate the entries, as they separate the items of a list.

To read a value, put its key in square brackets after the dictionary, the same brackets you used for indexes on Days 3 and 6: `prices["muffin"]` means "the value stored with the key `"muffin"`". This is called **looking up** a key.

```
>>> ages = {"Ana": 15, "Ben": 16, "Chloe": 15}
>>> ages["Ben"]
16
>>> ages["Chloe"]
15
>>> len(ages)
3
>>> ages
{'Ana': 15, 'Ben': 16, 'Chloe': 15}
>>> type(ages)
<class 'dict'>
>>> empty = {}
>>> len(empty)
0
```

- `ages["Ben"]` gives 16, the value paired with `"Ben"`.
- `len()` counts the **entries** (the pairs), so `len(ages)` is 3, not 6.
- The REPL shows the dictionary in the same form you type it, with single quotes as it does for strings.
- `type(ages)` is `dict`, Python's short name for dictionary.
- `{}` with nothing inside is an **empty dictionary**. That is why Day 7 made an empty set with `set()`: the empty braces were already taken by `dict`.

A dictionary remembers the order in which entries were added, so it prints and loops in that order. But that order is only for display. There is no "first entry" to ask for by number, because you always look up by key.

WORKED EXAMPLE Café prices

Here is the café again as a dictionary. Watch how each line finds a price by name, without any searching.

menu_prices.py

```
1 prices = {"tea": 2.50, "coffee": 3.00, "muffin": 2.75}
2 print(prices)
3 print("Tea costs", prices["tea"])
4 total = prices["coffee"] + prices["muffin"]
5 print(f"Coffee and a muffin: ${total:.2f}")
```

```
{'tea': 2.5, 'coffee': 3.0, 'muffin': 2.75}
Tea costs 2.5
Coffee and a muffin: $5.75
```

- Line 1 builds the dictionary with three entries. The keys are strings, and the values are floats.
- Line 2 prints the whole dictionary. Python shows 2.50 as 2.5 and 3.00 as 3.0, as it does for any float (Day 2). The values are the same numbers.
- Line 3 looks up "tea" and gets 2.5.
- Line 4 looks up two keys and adds the two values: 3.0 + 2.75 is 5.75.
- Line 5 shows the total with two decimal places, using the `:.2f` format specifier from Day 3.

The key inside the brackets does not have to be typed into the code. It can be any expression that gives a key, such as a variable holding what the user typed. This run types `Muffin`, with a space before it and a capital M:

lookup_price.py

```
1 prices = {"tea": 2.50, "coffee": 3.00, "muffin": 2.75}
2 item = input("Item: ").strip().lower()
3 print(f"One {item} costs ${prices[item]:.2f}")
```

```
Item:  Muffin
One muffin costs $2.75
```

`strip()` and `lower()` (Day 3) turn `Muffin` into `muffin`, which matches the key exactly. Keys are compared like `==` compares strings: `"Muffin"` and `"muffin"` are different keys. Cleaning typed text before you use it as a key is a habit worth forming.

A dictionary's keys are **unique**: each key appears at most once. If you write the same key twice, the later value wins:

```
>>> ages = {"Ana": 15, "Ben": 16, "Ana": 17}
>>> ages
{'Ana': 17, 'Ben': 16}
>>> len(ages)
2
```

There is only one `"Ana"` entry, holding 17. That makes sense: if one key could have two values, `ages["Ana"]` would not know which one to give back. Values, on the other hand, can repeat freely; Ana and Chloe were both 15 earlier.

KEY IDEA

A dictionary stores key-value pairs: `{"tea": 2.50, "coffee": 3.00}`. You look a value up by its key with `prices["tea"]`, not by position. Each key appears once; values can repeat.

WATCH OUT

Asking for an entry by position. Lists and strings have indexes, so it is natural to try index 0 for "the first price":

`by_position.py`

```
1 prices = {"tea": 2.50, "coffee": 3.00, "muffin": 2.75}
2 print(prices[0])
```

```
Traceback (most recent call last):
  File "../day08/by_position.py", line 2, in <module>
    print(prices[0])
           ~~~~~^^^
KeyError: 0
```

To a dictionary, the brackets always hold a **key**. Python looks for an entry whose key is the number 0, finds none, and stops with a **KeyError**, an error that means "there is no entry with this key". The message shows the key it looked for: `0`. Use the key itself: `prices["tea"]`. (8.4 shows how to deal with keys that might be missing.)

PAUSE AND TRY**Question 1** Tracing

What does this program print?

`room_codes.py`

```
1 rooms = {"math": "B12", "art": "A3", "science": "C7"}
2 subject = "art"
3 print(rooms[subject], len(rooms))
4 print(rooms["science"] + rooms["math"])
```

8.2 Which Types Can Be Keys

So far every key has been a string. Keys can be other types too, but not every type, and the reason is the same one you met with sets on Day 7.

A set finds a value fast by working out its **hash**, a number computed from the value, and looking straight in the one place that number points to. A dictionary does exactly this with its **keys**. When you store an entry, Python computes the key's hash and keeps the entry in the place it points to. When you look the key up later, Python computes the same hash and goes straight there. That is why a lookup is fast even in a dictionary with a million entries: there is no searching.

The plan only works if a key's hash never changes. If a key could change after it was stored, its hash would point to the wrong place, and the entry would be lost. So, as with sets, a key must be **hashable**: a value that cannot change.

Can be a key (immutable)	Cannot be a key (mutable)
str: "tea"	list: ["row A", 3]
int: 3	set: {"a", "b"}
float: 9.5	dict: {"a": 1}
bool: True	
tuple of immutable values: ("A", 3)	

Strings and numbers cannot be changed (Days 2 and 3), so they make good keys. Lists, sets, and dictionaries can change, so they cannot. A tuple cannot change (Day 7), so it can be a key as long as everything inside it is also unchangeable.

```
>>> medals = {1: "gold", 2: "silver", 3: "bronze"}
>>> medals[2]
'silver'
>>> board = {(0, 0): "rook", (0, 1): "knight"}
>>> board[(0, 1)]
'knight'
>>> readings = {9.5: "low", 12.0: "high"}
>>> readings[12.0]
'high'
```

- `medals` uses whole numbers as keys. `medals[2]` looks like a list index but is not one: 2 is a key, and there is no index 0 here at all.
- `board` uses tuples as keys, a natural way to name a square by its (row, column) position. To look one up, write the same tuple: `board[(0, 1)]`.
- Floats can be keys too, though whole numbers and strings are far more common.

The rule is only about **keys**. A **value** can be anything at all, including a list or another dictionary; 8.7 makes good use of that.

KEY IDEA

Keys must be **hashable**: values that cannot change, such as strings, numbers, and tuples. Lists, sets, and dictionaries cannot be keys. Values can be any type.

WATCH OUT

Using a list as a key. A seating chart might use a row and a seat number together as the key. Written as a list, it fails, just as a list could not go in a set on Day 7:

list_key.py

```
1 seats = {}
2 seats[["row A", 3]] = "Ana"
3 print(seats)
```

```
Traceback (most recent call last):
  File ".../day08/list_key.py", line 2, in <module>
    seats[["row A", 3]] = "Ana"
    ~~~~~^~~~~~
TypeError: cannot use 'list' as a dict key (unhashable type: 'list')
```

"Unhashable" means Python cannot compute a lasting hash for it, because a list can change. Older versions of Python print only `TypeError: unhashable type: 'list'`. The fix is to use a tuple, which holds the same two values but can never change:

tuple_key.py

```
1 seats = {}
2 seats[("row A", 3)] = "Ana"
3 seats[("row B", 1)] = "Ben"
4 print(seats)
5 print(seats[("row A", 3)])
```

```
{('row A', 3): 'Ana', ('row B', 1): 'Ben'}
Ana
```

(Line 2 adds an entry with `=`; that is the topic of the next subsection.)

PAUSE AND TRY

Question 2 Multiple choice

Which of these dictionaries **cannot** be made?

- A. `{(3, 4): "treasure"}`
- B. `{"scores": [88, 92]}`
- C. `{[3, 4]: "treasure"}`
- D. `{True: "yes", 0: "zero"}`

8.3 Adding, Updating, and Deleting

A shop's stock changes all day: new products arrive, counts go up and down, and products are dropped. Like a list (Day 6), a dictionary is **mutable**: you can change it after it is made, and it stays the same dictionary.

Adding and updating use the same line. Assign to a key in brackets:

```
1 stock["plums"] = 8
```

- If "plums" is **not** a key yet, Python **adds** a new entry.
- If "plums" **is** a key already, Python **updates** it: the old value is replaced. Keys are unique, so there is never a second "plums" entry.

You do not need `append` or any method to add an entry. Since the key says where the value goes, the assignment is enough.

Deleting has two forms:

- `del stock["plums"]` removes the entry with that key. `del` is a keyword (a word Python reserves), not a function, so there are no parentheses around the target.
- `stock.pop("plums")` also removes the entry, and **returns** the value it removed, so you can store it. Day 6's `pop` did the same for a list, using an index instead of a key.

WORKED EXAMPLE Keeping stock

Each `print` shows the dictionary after one change. Read each line and predict the next printout before you look.

stock.py

```
1 stock = {"apples": 12, "pears": 5}
2 print("Start:", stock)
3
4 stock["plums"] = 8
5 print("Add plums:", stock)
6
7 stock["pears"] = 9
8 print("Update pears:", stock)
9
10 stock["apples"] -= 3
11 print("Sell 3 apples:", stock)
12
13 del stock["plums"]
14 print("Delete plums:", stock)
15
16 sold_out = stock.pop("pears")
17 print("Popped pears:", sold_out)
18 print("End:", stock)
```

```
Start: {'apples': 12, 'pears': 5}
Add plums: {'apples': 12, 'pears': 5, 'plums': 8}
Update pears: {'apples': 12, 'pears': 9, 'plums': 8}
Sell 3 apples: {'apples': 9, 'pears': 9, 'plums': 8}
Delete plums: {'apples': 9, 'pears': 9}
Popped pears: 9
End: {'apples': 9}
```

Line	Statement	What happens
4	<code>stock["plums"] = 8</code>	"plums" is new: an entry is added at the end
7	<code>stock["pears"] = 9</code>	"pears" exists: its value 5 is replaced by 9
10	<code>stock["apples"] -= 3</code>	Reads 12, subtracts 3, stores 9 back under "apples"
13	<code>del stock["plums"]</code>	The "plums" entry is removed
16	<code>sold_out = stock.pop("pears")</code>	The "pears" entry is removed, and its value 9 is returned

Line 10 is Day 5's augmented assignment on a dictionary entry. `stock["apples"] -= 3` means `stock["apples"] = stock["apples"] - 3`: look up the old value, work out the new one, and store it under the same key. Updating "pears" kept its place in the order; only a new key goes at the end.

Now vary it. Here an update and an add look identical, and two more tools appear:

```
>>> colors = {"sky": "blue"}
>>> colors["grass"] = "green"
>>> colors["sky"] = "gray"
>>> colors
{'sky': 'gray', 'grass': 'green'}
>>> colors.pop("lemon", "not here")
'not here'
>>> colors.clear()
>>> colors
{}
```

- `colors["grass"] = "green"` adds, because `"grass"` was new. `colors["sky"] = "gray"` updates, because `"sky"` existed. The line alone does not tell you which will happen; the dictionary's current keys decide.
- `pop` with a **second argument** gives that value back when the key is missing, instead of stopping with an error. `"lemon"` is not a key, so you get `'not here'` and nothing is removed.
- `clear()` removes every entry and leaves an empty dictionary, `{}`, just as it empties a list.

KEY IDEA

`d[key] = value` adds the entry if the key is new and replaces the value if it is not. `del d[key]` removes an entry; `d.pop(key)` removes it and gives back its value.

WATCH OUT

Two names, one dictionary. Day 6 showed that `b = a` does not copy a list; it gives the same list a second name. Dictionaries are mutable too, so they behave the same way. This program tries to keep the old prices before a price rise:

alias_prices.py

```
1 prices = {"tea": 2.50, "coffee": 3.00}
2 old_prices = prices
3 prices["tea"] = 2.75
4 print("Now:", prices)
5 print("Before:", old_prices)
```

```
Now: {'tea': 2.75, 'coffee': 3.0}
Before: {'tea': 2.75, 'coffee': 3.0}
```

Line 2 does not make a second dictionary. `old_prices` and `prices` are two names for **one** dictionary, so the change on line 3 shows through both. The fix is the same as for a list: `copy()` makes a separate dictionary with the same entries.

copy_prices.py

```

1 prices = {"tea": 2.50, "coffee": 3.00}
2 old_prices = prices.copy()
3 prices["tea"] = 2.75
4 print("Now:", prices)
5 print("Before:", old_prices)

```

```

Now: {'tea': 2.75, 'coffee': 3.0}
Before: {'tea': 2.5, 'coffee': 3.0}

```

PAUSE AND TRY

Question 3 Tracing

What does this program print? Write down the dictionary after each line.

library.py

```

1 books = {"Dune": 2, "Emma": 1}
2 books["Holes"] = 4
3 books["Dune"] -= 1
4 books["Emma"] = books["Emma"] + books["Dune"]
5 removed = books.pop("Holes")
6 print(books, removed)

```

8.4 Missing Keys: KeyError and get()

Programs often look up keys that might not be there: a product nobody sells, a name nobody has saved yet, a word not counted yet. You saw in 8.1 that brackets with a missing key stop the program. You need a way to ask safely.

Here is that error again, in a program that gets as far as line 3:

missing_key.py

```

1 prices = {"tea": 2.50, "coffee": 3.00, "muffin": 2.75}
2 print("Tea:", prices["tea"])
3 print("Juice:", prices["juice"])
4 print("Done")

```

```

Tea: 2.5
Traceback (most recent call last):
  File ".../day08/missing_key.py", line 3, in <module>
    print("Juice:", prices["juice"])
                        ~~~~~^~~~~~
KeyError: 'juice'

```

Read it the way you learned on Day 1: the last line names the error and the missing key, 'juice', and the lines above point at line 3. Line 2 ran and printed. Line 4 never ran, because the program

stopped. A `KeyError` is Python refusing to guess: it does not know what the price of juice should be, so it will not make one up.

When a missing key is normal, not a mistake, use the `get()` method instead. `prices.get(key)` gives back the value if the key is there, and `None` if it is not. Day 2's `None` means "no value here", which is exactly the answer. You can also give `get` a second argument, the **default**: the value to give back when the key is missing.

```
>>> prices = {"tea": 2.50, "coffee": 3.00, "muffin": 2.75}
>>> prices.get("tea")
2.5
>>> prices.get("juice")
>>> print(prices.get("juice"))
None
>>> prices.get("juice", 0)
0
>>> prices.get("tea", 0)
2.5
>>> prices
{'tea': 2.5, 'coffee': 3.0, 'muffin': 2.75}
```

- `prices.get("tea")` finds the key and gives 2.5, just like `prices["tea"]`.
- `prices.get("juice")` gives `None`. The REPL shows nothing for `None` (as you saw on Day 2), so the next line prints it to make it visible.
- `prices.get("juice", 0)` gives the default, 0.
- `prices.get("tea", 0)` gives 2.5: the default is used **only** when the key is missing.
- The last line shows `get` never adds an entry. It only reads.

Expression	Key is there	Key is missing
<code>prices[key]</code>	Its value	Stops with <code>KeyError</code>
<code>prices.get(key)</code>	Its value	<code>None</code>
<code>prices.get(key, 0)</code>	Its value	0

Which should you use? Brackets, when the key **must** be there and a missing key means a bug you want to hear about. `get`, when a missing key is an ordinary situation your program should handle.

WORKED EXAMPLE A price checker that never crashes

This run types `juice`:

`safe_lookup.py`

```
1 prices = {"tea": 2.50, "coffee": 3.00, "muffin": 2.75}
2 item = input("Item: ").strip().lower()
3 price = prices.get(item)
4 if price is None:
5     print(f"Sorry, we do not sell {item}.")
6 else:
7     print(f"One {item} costs ${price:.2f}")
```

```
Item: juice
Sorry, we do not sell juice.
```

Line 3 uses `get` with no default, so `price` is `None` when the item is missing. Line 4 checks for that with `is None` (Day 4) and prints a friendly message instead of a traceback. Typing `tea` would give `price` the value 2.5, so the `else` branch would print `One tea costs $2.50`.

KEY IDEA

`d[key]` stops with a `KeyError` when the key is missing. `d.get(key)` gives `None` instead, and `d.get(key, default)` gives your default. `get` never changes the dictionary.

WATCH OUT

get without a default, then math. A program keeps points per player and wants to give Ben, who has none yet, his first point:

`get_no_default.py`

```
1 points = {"Ana": 3}
2 points["Ben"] = points.get("Ben") + 1
3 print(points)
```

Traceback (most recent call last):

```
File ".../day08/get_no_default.py", line 2, in <module>
    points["Ben"] = points.get("Ben") + 1
                        ~~~~~^~
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

`points.get("Ben")` gives `None`, and `None + 1` makes no sense, so Python stops with a `TypeError` (`NoneType` is the type of `None`). Give the default that fits the math. For a count or a total, that is 0:

`get_with_default.py`

```
1 points = {"Ana": 3}
2 points["Ben"] = points.get("Ben", 0) + 1
3 points["Ana"] = points.get("Ana", 0) + 1
4 print(points)
```

```
{'Ana': 4, 'Ben': 1}
```

For Ben, `get` gives the default 0, so he gets $0 + 1 = 1$. For Ana, it gives her real value 3, so she gets 4. One line handles both cases. 8.9 builds on this line.

PAUSE AND TRY

Question 4 Tracing

Predict exactly what the REPL prints for each line. Note the capital letters in the keys.

```
>>> sizes = {"S": 10, "M": 0, "L": 4}
>>> sizes.get("M", 5)
>>> sizes.get("XL", 5)
>>> sizes.get("s", "none")
>>> print(sizes.get("XL"))
```

8.5 Looping with keys(), values(), and items()

A report card prints every subject and its grade; a receipt prints every item and its price. On Day 5 you looped over each character of a string, and on Day 6 over each item of a list. Now you need to loop over each **entry** of a dictionary.

A for loop over a dictionary gives you its **keys**, one per pass, in the order they were added:

loop_keys.py

```
1 ages = {"Ana": 15, "Ben": 16, "Chloe": 15}
2 for name in ages:
3     print(name, "is", ages[name])
```

```
Ana is 15
Ben is 16
Chloe is 15
```

The loop variable `name` holds "Ana", then "Ben", then "Chloe". Inside the body, `ages[name]` looks up the value for the current key. The lookup is safe, because every key the loop hands you is in the dictionary.

Dictionaries also have three methods that let you choose what to loop over:

Method	Gives	Use when you need
<code>keys()</code>	The keys	Only the keys (same as the plain loop)
<code>values()</code>	The values	Only the values
<code>items()</code>	Each entry as a (key, value) tuple	Both together

```
>>> ages = {"Ana": 15, "Ben": 16, "Chloe": 15}
>>> ages.keys()
dict_keys(['Ana', 'Ben', 'Chloe'])
>>> ages.values()
dict_values([15, 16, 15])
>>> ages.items()
dict_items([('Ana', 15), ('Ben', 16), ('Chloe', 15)])
>>> list(ages.keys())
['Ana', 'Ben', 'Chloe']
>>> sorted(ages)
['Ana', 'Ben', 'Chloe']
>>> sum(ages.values())
46
```

The results look like lists wrapped in a label, such as `dict_keys([...])`. They are not lists, but you can loop over them, test them with `in`, and pass them to `list()` (which makes a new list from them), `sorted()`, `sum()`, `min()`, and `max()` (Day 6). `items()` gives tuples, the pairs you met on Day 7. `sorted(ages)` gives a **new list** of the keys in sorted order.

When only the values matter, loop over `values()`. Day 5's accumulator works as usual:

loop_values.py

```
1 ages = {"Ana": 15, "Ben": 16, "Chloe": 15}
2 total = 0
3 for age in ages.values():
4     total += age
5 print("Total:", total)
6 print(f"Average: {total / len(ages):.1f}")
7 print("Oldest:", max(ages.values()))
```

```
Total: 46
Average: 15.3
Oldest: 16
```

`total` starts at 0 and adds 15, 16, and 15. `len(ages)` counts three entries, so the average is $46 / 3$. `max()` from Day 6 finds the largest value directly.

WORKED EXAMPLE A price list with items()

When you need the key **and** the value, loop over `items()` and **unpack** each pair into two names, as you unpacked tuples on Day 7. Day 5's `enumerate` loop used the same two-name form.

loop_items.py

```
1 prices = {"tea": 2.50, "coffee": 3.00, "muffin": 2.75}
2 for item, price in prices.items():
3     print(f"{item}: ${price:.2f}")
```

```
tea: $2.50
coffee: $3.00
muffin: $2.75
```

Pass	Pair from items()	item	price
1	('tea', 2.5)	"tea"	2.5
2	('coffee', 3.0)	"coffee"	3.0
3	('muffin', 2.75)	"muffin"	2.75

Each pass, `items()` hands over one (key, value) tuple, and `for item, price in` unpacks it: the key goes into `item` and the value into `price`. No lookup is needed in the body.

To go through the entries in sorted order of their keys instead of the order they were added, loop over `sorted(scores)`:

sorted_keys.py

```
1 scores = {"Zoe": 88, "Ana": 92, "Max": 75}
2 for name in sorted(scores):
3     print(name, scores[name])
```

```
Ana 92
Max 75
Zoe 88
```

`sorted(scores)` is the list `['Ana', 'Max', 'Zoe']`. The loop goes through that list, and the body looks each name up. The dictionary itself is not changed.

KEY IDEA

`for key in d:` gives each key. `for value in d.values():` gives each value.
`for key, value in d.items():` gives both, unpacked from each pair. Use `sorted(d)` for the keys in order.

WATCH OUT

Deleting entries while looping over the same dictionary. This program tries to drop every fruit that is out of stock:

remove_while_looping.py

```
1 stock = {"apples": 12, "pears": 0, "plums": 0, "figs": 4}
2 for fruit in stock:
3     if stock[fruit] == 0:
4         del stock[fruit]
5 print(stock)
```

Traceback (most recent call last):

```
File ".../day08/remove_while_looping.py", line 2, in <module>
    for fruit in stock:
        ^^^^^
```

RuntimeError: dictionary changed size during iteration

The loop is walking through the entries while line 4 removes one of them out from under it. Python cannot know where to go next, so it stops with a `RuntimeError`, the error for "something went wrong while the program was running" that fits no more specific name. The fix is to loop over a **copy of the keys**, `list(stock)`, made before the loop starts. Then deleting from the dictionary does not disturb the list the loop is walking through:

remove_from_copy.py

```
1 stock = {"apples": 12, "pears": 0, "plums": 0, "figs": 4}
2 for fruit in list(stock):
3     if stock[fruit] == 0:
4         del stock[fruit]
5 print(stock)
```

```
{'apples': 12, 'figs': 4}
```

Changing the **value** of an existing key inside the loop is fine; only adding or deleting keys changes the size.

PAUSE AND TRY

Question 5 Tracing

What does this program print? Trace `day`, `mm`, and `dry`.

`rainfall.py`

```
1 rain = {"Mon": 4, "Tue": 0, "Wed": 12, "Thu": 3}
2 dry = 0
3 for day, mm in rain.items():
4     if mm == 0:
5         dry += 1
6     elif mm > 10:
7         print(day, "was wet")
8 print(dry, sum(rain.values()))
```

8.6 in Checks Keys

8.4 used `get` to deal with a missing key. Often you want to ask the question directly, as an `if` condition: "is this key in the dictionary?" Day 3's `in` asked whether one string is inside another, and Day 6's `in` asked whether a list holds a value. With a dictionary, `in` asks about the **keys**.

```
>>> ages = {"Ana": 15, "Ben": 16, "Chloe": 15}
>>> "Ana" in ages
True
>>> "Dev" in ages
False
>>> "Dev" not in ages
True
>>> 15 in ages
False
>>> 15 in ages.values()
True
>>> "ana" in ages
False
```

- `"Ana" in ages` is `True`: `"Ana"` is a key.
- `"Dev" not in ages` is `True`, because `"Dev"` is not a key. `not in` is the opposite of `in`, as with strings.
- `15 in ages` is `False`, even though 15 is right there in the dictionary. It is a **value**, and `in` checks only keys.
- To ask about the values, check `values()`: `15 in ages.values()` is `True`.
- `"ana" in ages` is `False`, because keys match exactly.

Why only keys? Because checking a key uses the hash from 8.2, so it is instant. Checking the values means looking at every one of them in turn, which is slower in a big dictionary, so Python makes you ask for it on purpose.

WORKED EXAMPLE Check, then look up

`in` and brackets make a safe pair: check that the key is there, and only then look it up. This run types Japan, peru, and quit:

check_before_lookup.py

```
1 capitals = {"france": "Paris", "japan": "Tokyo", "kenya": "Nairobi"}
2 while True:
3     country = input("Country (or quit): ").strip().lower()
4     if country == "quit":
5         break
6     if country in capitals:
7         print("The capital is", capitals[country])
8     else:
9         print("Sorry, I do not know", country)
10 print("Goodbye!")
```

```
Country (or quit): Japan
The capital is Tokyo
Country (or quit): peru
Sorry, I do not know peru
Country (or quit): quit
Goodbye!
```

- The `while True:` loop with `break` is Day 5's way to keep asking until the user says stop.
- `lower()` turns Japan into japan, which matches a key. `capitals[country]` on line 7 is safe, because line 6 just checked that the key is there.
- `peru` is not a key, so the `else` branch runs and no lookup is attempted.

`in` with brackets and `get` do the same job in two styles. Use `in` when the "missing" case needs different code, as here. Use `get` with a default when a missing key simply means a default value, as in the points example from 8.4.

KEY IDEA

`key in d` is `True` when `key` is one of the dictionary's keys. It never looks at the values; for those, use `value in d.values()`.

WATCH OUT

Using `in` to search the values. A program wants to know whether Tokyo is a capital city:

in_value_mistake.py

```
1 capitals = {"france": "Paris", "japan": "Tokyo", "kenya": "Nairobi"}
2 city = "Tokyo"
3 if city in capitals:
4     print(city, "is a capital")
5 else:
6     print(city, "is not a capital")
```

Tokyo is not a capital

No error, just a wrong answer. "Tokyo" is a **value** in capitals, but `in` looked only at the keys "france", "japan", and "kenya". Ask the values directly:

in_value_fixed.py

```
1 capitals = {"france": "Paris", "japan": "Tokyo", "kenya": "Nairobi"}
2 city = "Tokyo"
3 if city in capitals.values():
4     print(city, "is a capital")
5 else:
6     print(city, "is not a capital")
```

Tokyo is a capital

PAUSE AND TRY

Question 6 Tracing

Predict exactly what the REPL prints for each line.

```
>>> codes = {"red": 1, "green": 2, 3: "blue"}
>>> "red" in codes
>>> 1 in codes
>>> 3 in codes
>>> "blue" in codes.values()
>>> "Green" in codes
```

8.7 Nested Dictionaries

A **record** is all the information about one thing, kept together: a student's name, grade, test scores, and address. A dictionary is a natural record, because each piece gets a **name** (the key) instead of a position you must remember. `student["grade"]` says what it means; `student[1]` would not.

But some pieces are themselves collections. A student has several scores (a list) and an address with its own parts (city, zip code). Since values can be anything (8.2), a value can be a list or another dictionary. A dictionary inside another dictionary is called a **nested dictionary**.

To reach inside, **chain** the brackets. Python works left to right: `student["address"]["city"]` first looks up "address" in `student`, which gives the inner dictionary, then looks up "city" in that.

WORKED EXAMPLE A student record

student_record.py

```

1 student = {
2     "name": "Ana Silva",
3     "grade": 10,
4     "scores": [88, 92, 79],
5     "address": {"city": "Springfield", "zip": "12345"},
6 }
7 print(student["name"], "is in grade", student["grade"])
8 print("Second score:", student["scores"][1])
9 print("City:", student["address"]["city"])
10
11 student["scores"].append(95)
12 student["address"]["zip"] = "12399"
13 print(student["scores"])
14 print(student["address"])

```

```

Ana Silva is in grade 10
Second score: 92
City: Springfield
[88, 92, 79, 95]
{'city': 'Springfield', 'zip': '12399'}

```

Lines 1–6 write one dictionary across several lines. Python allows this inside any brackets or braces, and it makes a record much easier to read: one entry per line, with the closing brace back at the left edge. The comma after the last entry is allowed and is common in this style.

Expression	Step 1 gives	Step 2 gives
student["scores"][1]	the list [88, 92, 79]	index 1: 92
student["address"]["city"]	the dictionary {'city': ..., 'zip': ...}	'Springfield'

- Line 8: the value of "scores" is a list, so the second bracket is a list **index**.
- Line 9: the value of "address" is a dictionary, so the second bracket is a **key**.
- Line 11: student["scores"] is the list itself, not a copy, so `append` (Day 6) adds 95 to the list stored inside the record.
- Line 12: assigning to student["address"]["zip"] updates the inner dictionary in place.

The pattern also works the other way round: a dictionary whose every value is a list. A gradebook maps each student's name to their scores:

gradebook.py

```

1 gradebook = {
2     "Ana": [88, 92, 79],
3     "Ben": [70, 85],
4     "Chloe": [95, 91, 98, 90],
5 }
6 for name, scores in gradebook.items():
7     average = sum(scores) / len(scores)
8     print(f"{name}: {len(scores)} scores, average {average:.1f}")
9 gradebook["Ben"].append(100)
10 print("Ben now:", gradebook["Ben"])

```

```
Ana: 3 scores, average 86.3
Ben: 2 scores, average 77.5
Chloe: 4 scores, average 93.5
Ben now: [70, 85, 100]
```

`items()` hands over each name with its whole list of scores. `sum()` and `len()` from Day 6 work on that list as on any other. After the loop, `gradebook["Ben"].append(100)` adds a score to Ben's list only.

To work out what a chain of brackets gives, read it one bracket at a time and ask, "what kind of value do I have now?" A dictionary needs a key next; a list needs an index.

KEY IDEA

A value can be a list or another dictionary. Chain brackets to reach inside, left to right: `student["address"]["city"]`. Each bracket works on the value the one before it gave.

WATCH OUT

Skipping a level. The city lives inside the address, not in the student record itself:

nested_wrong_order.py

```
1 student = {"name": "Ana Silva", "address": {"city": "Springfield", "zip":
    "12345"}}
2 print(student["city"])
```

```
Traceback (most recent call last):
  File "../day08/nested_wrong_order.py", line 2, in <module>
    print(student["city"])
          ~~~~~^~~~~~
KeyError: 'city'
```

`student` has only the keys "name" and "address". Lookups do not search inside the values; each bracket looks at one level only. Follow the path: `student["address"]["city"]`.

PAUSE AND TRY

Question 7 Tracing

What does this program print?

team_trace.py

```

1 team = {
2     "name": "Hawks",
3     "players": ["Ana", "Ben", "Raj"],
4     "coach": {"name": "Ms. Lee", "years": 4},
5 }
6 team["players"].remove("Ben")
7 team["coach"]["years"] += 1
8 print(team["players"][-1], len(team["players"]))
9 print(team["coach"]["name"], team["coach"]["years"])

```

8.8 Lists of Dictionaries

One record is a dictionary. But a to-do app has many tasks, a library has many books, and a class has many students. Each one is a record with the same keys, and you want to keep them in order, add new ones, and go through them all. That is a job for a list (Day 6), where each item is a dictionary: a **list of dictionaries**.

```

1 tasks = [
2     {"title": "Buy milk", "done": False},
3     {"title": "Walk the dog", "done": True},
4 ]

```

Reading it uses the same chaining as 8.7, but now the **index** comes first, because the outer collection is a list: `tasks[0]` is the first task's dictionary, and `tasks[0]["title"]` is its title.

WORKED EXAMPLE A to-do list

This is the shape of the to-do manager you will build on Day 12: each task has a "title" (a string) and "done" (a bool).

tasks_list.py

```

1 tasks = [
2     {"title": "Buy milk", "done": False},
3     {"title": "Walk the dog", "done": True},
4     {"title": "Read Day 8", "done": False},
5 ]
6 tasks[0]["done"] = True
7 tasks.append({"title": "Call Grandma", "done": False})
8
9 for number, task in enumerate(tasks, 1):
10     mark = "x" if task["done"] else " "
11     print(f"{number}. [{mark}] {task['title']}")

```

1. [x] Buy milk
2. [x] Walk the dog
3. [] Read Day 8
4. [] Call Grandma

- Line 6: `tasks[0]` is the "Buy milk" dictionary, and `["done"] = True` updates that one entry inside it. The other tasks are untouched.
- Line 7: `append` adds a whole new dictionary as the fourth item of the list.
- Line 9: `enumerate(tasks, 1)` (Day 5) numbers the tasks from 1, and each `task` is one dictionary.
- Line 10: a conditional expression (Day 4) picks `"x"` for a finished task and a space otherwise.
- Line 11: inside the f-string, the key is written `task['title']` with **single** quotes, because the f-string itself uses double quotes. Different quotes make it clear where the f-string ends.

Going through a list of dictionaries with a for loop gives one record per pass, and Day 5's patterns work across the records. This one adds up the pages and picks out the cheaper books:

`books_filter.py`

```
1 books = [
2     {"title": "Holes", "pages": 233, "price": 7.99},
3     {"title": "Dune", "pages": 412, "price": 10.99},
4     {"title": "Wonder", "pages": 310, "price": 8.50},
5 ]
6 total_pages = 0
7 for book in books:
8     total_pages += book["pages"]
9     if book["price"] < 9:
10         print(f"{book['title']} is under $9")
11 print("Total pages:", total_pages)
```

```
Holes is under $9
Wonder is under $9
Total pages: 955
```

`book` is one dictionary on each pass, so `book["pages"]` and `book["price"]` read that book's values. The running total adds $233 + 412 + 310 = 955$. Every book has the same keys, which is what makes one loop body work for all of them.

KEY IDEA

A list of dictionaries holds many records of the same shape. `records[i]["key"]` reads one field: the index picks the record, the key picks the field. `for record in records:` gives one dictionary per pass.

WATCH OUT

Using a key on the list itself. `tasks` is a list, so it has indexes, not keys:

`list_by_key.py`

```
1 tasks = [
2     {"title": "Buy milk", "done": False},
3     {"title": "Walk the dog", "done": True},
4 ]
5 print(tasks["title"])
```

```
Traceback (most recent call last):
  File "../day08/list_by_key.py", line 5, in <module>
    print(tasks["title"])
          ~~~~~^~~~~~
TypeError: list indices must be integers or slices, not str
```

Python says a list index must be a whole number (or a slice, as on Days 3 and 6), not a string. Pick a record first, by index or with a loop, and then use the key:

list_by_key_fixed.py

```
1 tasks = [
2     {"title": "Buy milk", "done": False},
3     {"title": "Walk the dog", "done": True},
4 ]
5 print(tasks[0]["title"])
6 for task in tasks:
7     print(task["title"])
```

```
Buy milk
Buy milk
Walk the dog
```

PAUSE AND TRY

Question 8 Multiple choice

What does this program print?

pets_trace.py

```
1 pets = [
2     {"name": "Rex", "kind": "dog", "age": 3},
3     {"name": "Tom", "kind": "cat", "age": 7},
4     {"name": "Bo", "kind": "dog", "age": 5},
5 ]
6 pets[2]["age"] += 1
7 oldest = None
8 for pet in pets:
9     if pet["kind"] == "dog":
10         if oldest is None or pet["age"] > oldest["age"]:
11             oldest = pet
12 print(oldest["name"], oldest["age"])
```

- A. Bo 6
- B. Bo 5
- C. Tom 7
- D. Rex 3

8.9 Counting with a Dictionary

A class votes for its lunch: pizza, tacos, pizza, sushi, and so on. How many votes did each food get? On Day 5 you counted with one counter variable. But here you do not know the foods in advance, so you cannot make one variable per food. You need one counter **per different value**, made as each new value turns up. A dictionary does exactly that: the key is the thing being counted, and the value is its count.

The pattern, for each item:

- If the item is not a key yet, this is its first time: add it with a count of 1.
- Otherwise, add 1 to its count.

WORKED EXAMPLE Counting votes

count_votes_if.py

```
1 votes = ["pizza", "tacos", "pizza", "sushi", "pizza", "tacos"]
2 counts = {}
3 for food in votes:
4     if food in counts:
5         counts[food] += 1
6     else:
7         counts[food] = 1
8 print(counts)
```

```
{'pizza': 3, 'tacos': 2, 'sushi': 1}
```

food	food in counts	counts after the pass
pizza	False: add	{'pizza': 1}
tacos	False: add	{'pizza': 1, 'tacos': 1}
pizza	True: +1	{'pizza': 2, 'tacos': 1}
sushi	False: add	{'pizza': 2, 'tacos': 1, 'sushi': 1}
pizza	True: +1	{'pizza': 3, 'tacos': 1, 'sushi': 1}
tacos	True: +1	{'pizza': 3, 'tacos': 2, 'sushi': 1}

`counts` is an accumulator (Day 5): it starts empty before the loop, is updated on every pass, and is used after. The difference from Day 5 is that it holds many counts at once.

The `if/else` can be shortened to one line with `get` and a default of 0, the line you met in 8.4's Watch Out: "the count so far, or 0 if there is none, plus 1". Once the counts are in, Day 5's maximum pattern finds the winner:

count_votes_get.py

```
1 votes = ["pizza", "tacos", "pizza", "sushi", "pizza", "tacos"]
2 counts = {}
3 for food in votes:
4     counts[food] = counts.get(food, 0) + 1
5 print(counts)
6
7 winner = None
8 for food, count in counts.items():
9     if winner is None or count > counts[winner]:
10         winner = food
11 print("Winner:", winner, "with", counts[winner], "votes")
```

```
{'pizza': 3, 'tacos': 2, 'sushi': 1}
Winner: pizza with 3 votes
```

- Line 4 does the whole count: for a new food, `get` gives 0, so the count becomes 1; for a food already seen, it gives the count so far.
- Lines 7–10 go through the `(food, count)` pairs. `winner` starts at `None` (Day 5) and holds the food with the most votes so far, so `counts[winner]` is the best count so far.

A close cousin of counting is **totaling by category**: instead of adding 1, add an amount. Here each purchase is a `(category, amount)` tuple, unpacked in the loop:

sum_by_category.py

```
1 purchases = [
2     ("food", 12.50),
3     ("games", 20.00),
4     ("food", 8.25),
5     ("books", 9.99),
6     ("food", 4.00),
7 ]
8 spent = {}
9 for category, amount in purchases:
10     spent[category] = spent.get(category, 0) + amount
11 for category, total in spent.items():
12     print(f"{category}: ${total:.2f}")
```

```
food: $24.75
games: $20.00
books: $9.99
```

Food is $12.50 + 8.25 + 4.00 = 24.75$. The only change from counting is `+ amount` instead of `+ 1`.

KEY IDEA

To count, start with `counts = {}` and, for each item, write `counts[item] = counts.get(item, 0) + 1`. To total by category, add an amount instead of 1.

WATCH OUT

Adding to a count that does not exist yet. `+=` looks like the obvious way to count:

count_no_start.py

```
1 votes = ["pizza", "tacos", "pizza"]
2 counts = {}
3 for food in votes:
4     counts[food] += 1
5 print(counts)
```

```
Traceback (most recent call last):
  File "../day08/count_no_start.py", line 4, in <module>
    counts[food] += 1
    ~~~~~^~~~~~
KeyError: 'pizza'
```

`counts[food] += 1` means `counts[food] = counts[food] + 1`, and the right side has to **read** `counts["pizza"]` first. On the first pass there is no such key, so Python stops with a `KeyError`. Every count has to start somewhere: use the `if/else` version or `counts.get(food, 0) + 1`.

PAUSE AND TRY**Question 9** Tracing

What does this program print? Keep track of `tally` after each roll.

count_trace.py

```
1 rolls = [3, 6, 3, 1, 6, 3]
2 tally = {}
3 for roll in rolls:
4     tally[roll] = tally.get(roll, 0) + 1
5 print(tally)
6 print(len(tally), tally.get(2, 0))
```

8.10 Grouping with a Dictionary

Counting answers "how many in each group?" Sometimes you need the group itself: **which** students are in grade 10, **which** words have five letters. **Grouping** collects items into lists, one list per group, with the dictionary's key naming the group.

The pattern is close to counting, but each value is a **list** that grows:

- Work out the item's group (its key).
- If the key is not in the dictionary yet, add it with an empty list, `[]`.
- Append the item to that key's list.

WORKED EXAMPLE Students by grade

Each student is a `(name, grade)` tuple, unpacked in the loop.

group_by_grade.py

```

1 students = [("Ana", 10), ("Ben", 11), ("Chloe", 10), ("Dev", 12), ("Eli", 11)]
2 by_grade = {}
3 for name, grade in students:
4     if grade not in by_grade:
5         by_grade[grade] = []
6     by_grade[grade].append(name)
7 print(by_grade)
8 for grade in sorted(by_grade):
9     print(f"Grade {grade}: {' '.join(by_grade[grade])}")

```

```

{10: ['Ana', 'Chloe'], 11: ['Ben', 'Eli'], 12: ['Dev']}
Grade 10: Ana, Chloe
Grade 11: Ben, Eli
Grade 12: Dev

```

name, grade	grade in by_grade	by_grade after the pass
Ana, 10	False: add []	{10: ['Ana']}
Ben, 11	False: add []	{10: ['Ana'], 11: ['Ben']}
Chloe, 10	True	{10: ['Ana', 'Chloe'], 11: ['Ben']}
Dev, 12	False: add []	{10: [...], 11: ['Ben'], 12: ['Dev']}
Eli, 11	True	{10: [...], 11: ['Ben', 'Eli'], 12: [...]}

- Lines 4–5 make sure the group exists. They run only the first time a grade turns up.
- Line 6 runs on **every** pass. `by_grade[grade]` is the list stored for that grade, and `append` adds the name to it, changing the list inside the dictionary (as in 8.7).
- Line 8 loops over the grades in sorted order, and `", ".join(...)` (Day 3) turns each list of names into one string.

The key can be anything you can work out from the item: a grade, a word's length (`len(word)`), or a name's first letter. The groups appear in the order each key was first seen.

Pattern	New key starts at	Each item does	Answers
Counting	0	adds 1 to the count	"How many in each?"
Grouping	[]	appends to the list	"Which ones in each?"

KEY IDEA

To group, for each item work out its key, add the key with an empty list if it is new, then append the item: `groups[key].append(item)`.

WATCH OUT

Making a new list every time. It is tempting to skip the `if` and write the list directly:

group_overwrite.py

```
1 students = [("Ana", 10), ("Ben", 11), ("Chloe", 10), ("Dev", 12), ("Eli", 11)]
2 by_grade = {}
3 for name, grade in students:
4     by_grade[grade] = [name]
5 print(by_grade)
```

```
{10: ['Chloe'], 11: ['Eli'], 12: ['Dev']}
```

Line 4 assigns a brand-new one-item list on **every** pass. When Chloe arrives, `by_grade[10] = ["Chloe"]` replaces Ana's list (8.3: assigning to an existing key replaces its value). Each group ends up holding only its last member. Create the empty list only when the key is new, and append every time.

PAUSE AND TRY

Question 10 Tracing

What does this program print?

group_trace.py

```
1 animals = ["bat", "cow", "bee", "ant", "cat", "bear"]
2 groups = {}
3 for animal in animals:
4     first = animal[0]
5     if first not in groups:
6         groups[first] = []
7     groups[first].append(animal)
8 print(groups["b"])
9 print(len(groups), groups.get("d", []))
```

Putting it together

A class quiz is played in teams. Each result is a record: a player's name, team, and points. The program needs each team's total (counting by category, 8.9), each team's members (grouping, 8.10), and the winning team (Day 5's maximum, over a dictionary's items, 8.5).

quiz_teams.py

```

1 results = [
2     {"name": "Ana", "team": "Red", "points": 12},
3     {"name": "Ben", "team": "Blue", "points": 9},
4     {"name": "Chloe", "team": "Red", "points": 7},
5     {"name": "Dev", "team": "Green", "points": 15},
6     {"name": "Eli", "team": "Blue", "points": 11},
7 ]
8
9 totals = {}
10 members = {}
11 for result in results:
12     team = result["team"]
13     totals[team] = totals.get(team, 0) + result["points"]
14     if team not in members:
15         members[team] = []
16     members[team].append(result["name"])
17
18 best = None
19 for team, total in totals.items():
20     names = ", ".join(members[team])
21     print(f'{team}: {total} points ({names})')
22     if best is None or total > totals[best]:
23         best = team
24 print("Winning team:", best)

```

```

Red: 19 points (Ana, Chloe)
Blue: 20 points (Ben, Eli)
Green: 15 points (Dev)
Winning team: Blue

```

The first loop goes through the list of dictionaries (8.8) and builds two dictionaries at once:

Player	team	totals after the pass	members after the pass
Ana	Red	{'Red': 12}	{'Red': ['Ana']}
Ben	Blue	{'Red': 12, 'Blue': 9}	{'Red': ['Ana'], 'Blue': ['Ben']}
Chloe	Red	{'Red': 19, 'Blue': 9}	{'Red': ['Ana', 'Chloe'], 'Blue': ['Ben']}
Dev	Green	{'Red': 19, 'Blue': 9, 'Green': 15}	adds 'Green': ['Dev']
Eli	Blue	{'Red': 19, 'Blue': 20, 'Green': 15}	'Blue': ['Ben', 'Eli']

- Line 12 stores `result["team"]` in a short name, `team`, because the next four lines all need it.
- Line 13 is the totaling line: `get(team, 0)` starts a new team at 0, then adds the player's points.
- Lines 14–16 are the grouping lines: a new team gets an empty list, and every player is appended to their team's list.
- The second loop uses `items()` to get each team with its total. It prints a line per team, using `members[team]` (the same key in the other dictionary) to list the players. It also keeps `best`, the team with the highest total so far, starting from `None`.
- Blue wins with 20, one point ahead of Red's 19, even though Dev, on Green, scored more points than any other single player. Totals decide, not single scores.

The two dictionaries share their keys (the team names), which is a common way to keep several facts about the same groups.

Vocabulary

Term	Definition	Example
dictionary (<code>dict</code>)	A collection of key-value pairs, looked up by key	<code>{"tea": 2.5}</code>
key	What you look a value up by; unique in a dictionary	<code>"tea"</code>
value	What is stored with a key; any type	<code>2.5</code>
key-value pair / entry	One key together with its value	<code>"tea": 2.5</code>
looking up	Getting the value stored with a key	<code>prices["tea"]</code>
empty dictionary	A dictionary with no entries	<code>{}</code>
KeyError	The error when a key is not in the dictionary	<code>KeyError: 'juice'</code>
hash	A number Python works out from a key to decide where the entry is kept	used by every lookup
hashable	Able to be a key: a value that cannot change	<code>str, int, tuple</code>
del	Removes an entry by key	<code>del stock["plums"]</code>
pop()	Removes an entry and returns its value	<code>stock.pop("pears")</code>
get()	Returns a key's value, or <code>None</code> or a default when the key is missing	<code>prices.get("juice", 0)</code>
default	The value <code>get</code> (or <code>pop</code>) gives back when the key is missing	the <code>0</code> in <code>get(key, 0)</code>
keys(), values(), items()	The keys, the values, or the (key, value) pairs, for looping	<code>for k, v in d.items():</code>
record	All the information about one thing, kept together	a student dictionary
nested dictionary	A dictionary stored as a value inside another	<code>student["address"]["city"]</code>
list of dictionaries	A list whose items are records with the same keys	<code>tasks[0]["title"]</code>
grouping	Collecting items into one list per key	<code>groups[key].append(item)</code>

Summary

- A **dictionary** stores **key-value pairs** and finds a value by its **key**, not by position: `prices["tea"]`. Keys are unique and must be **hashable** (strings, numbers, tuples); values can be anything.

- `d[key] = value` adds a new entry or replaces an existing value; `del d[key]` and `d.pop(key)` remove one. Like lists, dictionaries are mutable, so `b = a` shares and `a.copy()` copies.
- Brackets with a missing key stop with a **KeyError**. `key in d` checks the keys first, and `d.get(key, default)` gives a default instead. `in` never looks at values; use `d.values()` for that.
- `for key in d:`, `d.values()`, and `for key, value in d.items():` loop over a dictionary in the order entries were added. Do not add or delete keys while looping over the same dictionary.
- Values can be lists or dictionaries, so a dictionary models a **record**, and a **list of dictionaries** holds many records. Chain brackets to reach inside: `tasks[0]["title"]`.
- **Counting** uses `counts[item] = counts.get(item, 0) + 1`; **grouping** adds an empty list for a new key and appends every item.

Practice

Do these in your `day08` folder. Type each program yourself, run it, and only then open the answer. Functions arrive tomorrow on Day 9, so write each one as a plain script.

Question 11 Short answer

Word frequencies. Store a short paragraph in a variable. Count how many times each word appears, ignoring capital letters and the punctuation `.` and `,`, then print each word with its count and say which word is most common.

Question 12 Short answer

Contact book. Write a program that keeps asking for a command until the user types `quit`. `add` asks for a name and a phone number and saves them; `find` asks for a name and shows the number, or says there is no such contact; `delete` asks for a name and removes it; `list` shows every contact in alphabetical order.

Question 13 Short answer

Group by first letter. Given a list of names, group them by their first letter and print each letter, how many names it has, and the names, in alphabetical order of the letters.

Review questions

Question 14 Tracing

What does this program print? Keep track of `left` for each item and of `total`.

review_cafe_order.py

```

1 menu = {
2     "tea": {"price": 2.50, "left": 2},
3     "muffin": {"price": 2.75, "left": 0},
4 }
5 order = ["tea", "juice", "muffin", "tea", "tea"]
6 total = 0
7 for item in order:
8     if item not in menu:
9         print("No", item)
10    elif menu[item]["left"] == 0:
11        print("Out of", item)
12    else:
13        menu[item]["left"] -= 1
14        total += menu[item]["price"]
15 print(f"Total: ${total:.2f}")

```

Question 15 Debugging

This program should print each student's score. Instead it stops with an error. Explain the bug and write the fix.

review_unpack.py

```

1 scores = {"Ana": 92, "Ben": 78, "Chloe": 85}
2 for name, score in scores:
3     print(name, "scored", score)

```

Traceback (most recent call last):

File "../day08/review_unpack.py", line 2, in <module>

for name, score in scores:

^^^^^^^^^^^^^^

ValueError: too many values to unpack (expected 2)

Question 16 Multiple choice

A program stores the number of visitors for each date. A date is three whole numbers: year, month, and day. Which is the best key for one entry?

- A. [2026, 10, 15], a list
- B. (2026, 10, 15), a tuple
- C. {2026, 10, 15}, a set
- D. 2026, just the year

Question 17 Short answer

A club wants to store each member's name, age, and favorite games. A classmate suggests a list: ["Ana", 15, ["chess", "go"]]. Explain why a dictionary is a better model for one member, write that member as a dictionary, and say how you would store the whole club.

Question 18 Tracing

What does this program print?

review_pass_fail.py

```
1 scores = {"Ana": 92, "Ben": 58, "Chloe": 85, "Dev": 40, "Eli": 70}
2 results = {}
3 for name, score in scores.items():
4     outcome = "pass" if score >= 60 else "fail"
5     if outcome not in results:
6         results[outcome] = []
7     results[outcome].append(name)
8 print(results)
9 print(len(results["pass"]), results.get("merit", "none"))
```

Question 19 Multiple choice

stock = {"pens": 4, "pads": 0}. Which line stops the program with a `KeyError`?

- A. `print(stock.get("ink", 0))`
- B. `stock["ink"] = 5`
- C. `stock["ink"] += 5`
- D. `print("ink" in stock)`

Go further

The official Python Tutorial covers dictionaries in Data Structures: Dictionaries and looping over them with `items()` in Looping Techniques. Some of its examples use functions and comprehensions, which you meet on Days 9 and 11.

Reflection

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

- ☐ I can make a dictionary and look a value up by its key. Reread: 8.1 Key-Value Pairs
- ☐ I can say which types can be keys and why a list cannot. Reread: 8.2 Which Types Can Be Keys
- ☐ I can add, update, and delete entries. Reread: 8.3 Adding, Updating, and Deleting
- ☐ I can explain when a `KeyError` happens and avoid it with `get()`. Reread: 8.4 Missing Keys: `KeyError` and `get()`
- ☐ I can loop over keys, values, and items, and unpack each pair. Reread: 8.5 Looping with `keys()`, `values()`, and `items()`
- ☐ I can use `in` to check for a key, and `values()` to check for a value. Reread: 8.6 `in` Checks Keys
- ☐ I can model a record with a nested dictionary and reach inside it. Reread: 8.7 Nested Dictionaries
- ☐ I can loop over a list of dictionaries and read each record. Reread: 8.8 Lists of Dictionaries
- ☐ I can count how many times each value appears. Reread: 8.9 Counting with a Dictionary
- ☐ I can group items into lists by a key. Reread: 8.10 Grouping with a Dictionary

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
A3 3
C7B12
```

- Line 3: `subject` holds `"art"`, so `rooms[subject]` is `rooms["art"]`, which is `"A3"`. `len(rooms)` counts three entries.
- Line 4: the values are strings, so `+` joins them (Day 3): `"C7" + "B12"` is `"C7B12"`.

If you printed `art 3`, remember that looking up a key gives its **value**, not the key.

Reread *8.1 Key-Value Pairs* if your output differed.

Question 2

C. The key `[3, 4]` is a list. Lists can change, so they are not hashable, and Python stops with a `TypeError`.

- **A** is fine: `(3, 4)` is a tuple of numbers, which can never change.
- **B** is fine: the key is a string. The value is a list, and values can be anything.
- **D** is fine: a bool and an int are both unchangeable, so both can be keys.

Reread *8.2 Which Types Can Be Keys* if you chose differently.

Question 3

```
{'Dune': 1, 'Emma': 2} 4
```

Line	books after the line	Why
1	<code>{'Dune': 2, 'Emma': 1}</code>	Made with two entries
2	<code>{'Dune': 2, 'Emma': 1, 'Holes': 4}</code>	<code>"Holes"</code> is new, so it is added
3	<code>{'Dune': 1, 'Emma': 1, 'Holes': 4}</code>	<code>2 - 1</code>
4	<code>{'Dune': 1, 'Emma': 2, 'Holes': 4}</code>	<code>1 + 1</code> , using the new Dune value
5	<code>{'Dune': 1, 'Emma': 2}</code>	<code>pop</code> removes Holes and returns 4

The key step is line 4: line 3 has already changed Dune to 1, so `books["Emma"] + books["Dune"]` is `1 + 1`. If you got `'Emma': 3`, you used Dune's starting value.

Reread *8.3 Adding, Updating, and Deleting* if your output differed.

Question 4

```
>>> sizes = {"S": 10, "M": 0, "L": 4}
>>> sizes.get("M", 5)
0
>>> sizes.get("XL", 5)
5
>>> sizes.get("s", "none")
'none'
>>> print(sizes.get("XL"))
None
```

- "M" is a key, so `get` gives its value, 0. The default 5 is used only for a **missing** key, not for a value that happens to be 0.
- "XL" is missing, so the default 5 comes back.
- "s" is not "S": keys match exactly, capitals included, so the default 'none' comes back.
- With no default, a missing key gives `None`, which `print` shows.

Reread 8.4 *Missing Keys: `KeyError` and `get()`* if any answer differed.

Question 5

```
Wed was wet
1 19
```

day	mm	Branch	dry
Mon	4	neither	0
Tue	0	<code>mm == 0</code>	1
Wed	12	<code>mm > 10: prints</code>	1
Thu	3	neither	1

After the loop, `sum(rain.values())` is $4 + 0 + 12 + 3 = 19$. `items()` gives each pair, and `for day, mm in` unpacks it into the day name and the rainfall.

Reread 8.5 *Looping with `keys()`, `values()`, and `items()`* if your output differed.

Question 6

```
>>> codes = {"red": 1, "green": 2, 3: "blue"}
>>> "red" in codes
True
>>> 1 in codes
False
>>> 3 in codes
True
>>> "blue" in codes.values()
True
>>> "Green" in codes
False
```

- "red" is a key: `True`.
- 1 is only a value (the value of "red"), so `1 in codes` is `False`.

- 3 is a key (its value is "blue"): True. A dictionary may mix key types.
- "blue" is among the values, so checking `values()` gives True.
- "Green" with a capital G is not the key "green": False.

Reread 8.6 in *Checks Keys* if any answer differed.

Question 7

```
Raj 2
Ms. Lee 5
```

- Line 6: `team["players"]` is the list `["Ana", "Ben", "Raj"]`, and Day 6's `remove` takes "Ben" out of it, leaving `["Ana", "Raj"]`.
- Line 7: `team["coach"]` is the inner dictionary, and its "years" goes from 4 to 5.
- Line 8: index -1 is the last player, "Raj", and the list now has 2 players.
- Line 9: the coach's name is "Ms. Lee", which is a different key in a different dictionary from the team's "name".

Reread 8.7 *Nested Dictionaries* if your output differed.

Question 8

A. Line 6 adds 1 to the third pet's age, so Bo is 6. The loop then runs Day 5's maximum pattern over the **dogs** only. Rex (3) is the first dog, so `oldest` becomes Rex's dictionary. Tom is skipped because he is a cat. Bo is 6, which is more than 3, so `oldest` becomes Bo's dictionary.

- **B** misses line 6: `pets[2]["age"] += 1` changes Bo's dictionary in the list before the loop runs.
- **C** ignores the "kind" check: Tom is the oldest pet, but he is a cat.
- **D** stops at the first dog, but the loop keeps going and Bo is older.

Reread 8.8 *Lists of Dictionaries* if you chose differently.

Question 9

```
{3: 3, 6: 2, 1: 1}
3 0
```

roll	tally after the pass
3	{3: 1}
6	{3: 1, 6: 1}
3	{3: 2, 6: 1}
1	{3: 2, 6: 1, 1: 1}
6	{3: 2, 6: 2, 1: 1}
3	{3: 3, 6: 2, 1: 1}

The keys are whole numbers here, in the order first seen: 3, 6, 1. `len(tally)` counts the **different** rolls, 3, not the six rolls. 2 was never rolled, so `tally.get(2, 0)` gives the default 0.

Reread 8.9 *Counting with a Dictionary* if your output differed.

Question 10

```
['bat', 'bee', 'bear']
3 []
```

`animal[0]` is the first letter (Day 3). The groups fill as: `b` gets `bat`, `c` gets `cow`, `b` gets `bee`, `a` gets `ant`, `c` gets `cat`, `b` gets `bear`. So groups is

`{'b': ['bat', 'bee', 'bear'], 'c': ['cow', 'cat'], 'a': ['ant']}`. It has 3 keys. No animal starts with `d`, so `get` gives its default, the empty list `[]`.

Reread *8.10 Grouping with a Dictionary* if your output differed.

Question 11

One possible solution:

`word_frequency.py`

```
1 paragraph = """The cat sat on the mat. The dog sat on the log.
2 The cat and the dog are friends."""
3
4 text = paragraph.lower().replace(".", "").replace(",", "")
5 counts = {}
6 for word in text.split():
7     counts[word] = counts.get(word, 0) + 1
8
9 for word, count in counts.items():
10     print(word, count)
11
12 most_common = None
13 for word, count in counts.items():
14     if most_common is None or count > counts[most_common]:
15         most_common = word
16 print(f"Most common: {most_common} ({counts[most_common]} times)")
```

```
the 6
cat 2
sat 2
on 2
mat 1
dog 2
log 1
and 1
are 1
friends 1
Most common: the (6 times)
```

Line 4 cleans the text first: `lower()` makes `The` and `the` the same key, and the two `replace` calls remove the punctuation so that `mat.` and `mat` are the same word. `split()` with no argument splits at any whitespace, including the line break inside the triple-quoted string (Day 3). Line 7 is the counting line from 8.9. The last loop is Day 5's maximum pattern over the `(word, count)` pairs.

Reread *8.9 Counting with a Dictionary* if your result differed.

Question 12

One possible solution:

contact_book.py

```
1 contacts = {}
2 print("Commands: add, find, delete, list, quit")
3 while True:
4     command = input("Command: ").strip().lower()
5     if command == "quit":
6         print("Goodbye!")
7         break
8     elif command == "add":
9         name = input("Name: ").strip()
10        phone = input("Phone: ").strip()
11        if name in contacts:
12            print(f"Updated {name}.")
13        else:
14            print(f"Added {name}.")
15            contacts[name] = phone
16    elif command == "find":
17        name = input("Name: ").strip()
18        phone = contacts.get(name)
19        if phone is None:
20            print(f"No contact named {name}.")
21        else:
22            print(f"{name}: {phone}")
23    elif command == "delete":
24        name = input("Name: ").strip()
25        if name in contacts:
26            del contacts[name]
27            print(f"Deleted {name}.")
28        else:
29            print(f"No contact named {name}.")
30    elif command == "list":
31        if not contacts:
32            print("No contacts yet.")
33        for name in sorted(contacts):
34            print(f"{name}: {contacts[name]}")
35    else:
36        print("Unknown command:", command)
```

This run adds Ana and Ben, finds Ana, looks for Zoe, deletes Ben, and lists what is left:

```

Commands: add, find, delete, list, quit
Command: add
Name: Ana
Phone: 555-0142
Added Ana.
Command: add
Name: Ben
Phone: 555-0199
Added Ben.
Command: find
Name: Ana
Ana: 555-0142
Command: find
Name: Zoe
No contact named Zoe.
Command: delete
Name: Ben
Deleted Ben.
Command: list
Ana: 555-0142
Command: quit
Goodbye!

```

- The dictionary maps each name (the key) to a phone number (the value). Phone numbers are kept as strings: they are labels, not amounts, and a number like 555-0142 is not math.
- `add` uses one assignment for both adding and updating (8.3); the `in` check before it only chooses which message to print.
- `find` uses `get` and checks for `None` (8.4), and `delete` checks with `in` before `del` (8.6), so a missing name never causes a `KeyError`.
- `list` uses `if not contacts::` an empty dictionary is falsy, like the empty string and empty list (Day 4). `sorted(contacts)` gives the names in alphabetical order (8.5).

Reread 8.3 *Adding, Updating, and Deleting* and 8.6 *in Checks Keys* if your result differed.

Question 13

One possible solution:

group_names.py

```

1 names = ["Maya", "Leo", "Ana", "Mateo", "Liam", "Aria", "Zoe", "Mia"]
2
3 by_letter = {}
4 for name in names:
5     letter = name[0].upper()
6     if letter not in by_letter:
7         by_letter[letter] = []
8     by_letter[letter].append(name)
9
10 for letter in sorted(by_letter):
11     group = by_letter[letter]
12     print(f"{letter} ({len(group)}): {' '.join(group)}")

```

```
A (2): Ana, Aria
L (2): Leo, Liam
M (3): Maya, Mateo, Mia
Z (1): Zoe
```

Line 5 works out the key: the first character, in capitals so that "ana" and "Ana" would land in the same group. Lines 6–8 are the grouping pattern from 8.10. The last loop goes through `sorted(by_letter)`, so the letters come out A, L, M, Z even though M was seen first. Inside each group, the names keep the order they had in the original list.

Reread *8.10 Grouping with a Dictionary* if your result differed.

Question 14

```
No juice
Out of muffin
Out of tea
Total: $5.00
```

item	Branch	tea left	total
tea	in menu, 2 left: sell	1	2.50
juice	"juice" not in menu	1	2.50
muffin	menu["muffin"]["left"] == 0	1	2.50
tea	1 left: sell	0	5.00
tea	menu["tea"]["left"] == 0	0	5.00

The order of the checks matters: `not in` comes first, so `menu["juice"]` is never looked up and there is no `KeyError`. `menu[item]["left"]` chains two lookups: first the item's inner dictionary, then its `"left"` value. Selling changes the inner dictionary, so by the third tea there is none left.

Reread *8.6 in Checks Keys* and *8.7 Nested Dictionaries* if your output differed.

Question 15

A for loop over a dictionary gives only the **keys**. On the first pass the loop gets "Ana", and for `name, score in` tries to unpack that string into two names. A string is a sequence of characters, and "Ana" has three of them, so Python stops: too many values for two names. Loop over `items()` to get (key, value) pairs:

review_unpack_fixed.py

```
1 scores = {"Ana": 92, "Ben": 78, "Chloe": 85}
2 for name, score in scores.items():
3     print(name, "scored", score)
```

```
Ana scored 92
Ben scored 78
Chloe scored 85
```

Older versions of Python word the error slightly differently, but it is still a `ValueError` about unpacking.

Reread 8.5 *Looping with keys(), values(), and items()* if you missed it.

Question 16

B. A tuple keeps the three numbers in order and can never change, so it is hashable and can be a key. `visitors[(2026, 10, 15)]` reads naturally.

- **A** fails: a list can change, so Python stops with `TypeError: ... unhashable type: 'list'`.
- **C** fails for the same reason (a set can change), and a set also has no order, so it could not tell October 15 from a date with the same numbers in another order.
- **D** can be a key, but every date in 2026 would share the same key, and each new day would replace the last one's count, because keys are unique.

Reread 8.2 *Which Types Can Be Keys* if you chose differently.

Question 17

Model answer: with a list, each piece is found by position, so the code must remember that index 1 means age; `member[1]` says nothing, and adding a new piece in the middle would shift every index. A dictionary names each piece, so `member["age"]` explains itself and the order does not matter:

```
1 member = {"name": "Ana", "age": 15, "games": ["chess", "go"]}
```

The games stay a list as the value, because there can be any number of them. The whole club is a **list of dictionaries**, one record per member, all with the same keys, so one loop can go through every member: `for member in club:`

A correct answer says that keys name the fields (where list positions do not), keeps the games in a list value, and stores many members as a list of dictionaries.

Reread 8.7 *Nested Dictionaries* and 8.8 *Lists of Dictionaries* if your reasoning differed.

Question 18

```
{'pass': ['Ana', 'Chloe', 'Eli'], 'fail': ['Ben', 'Dev']}
3 none
```

The loop goes through the pairs in order. The conditional expression (Day 4) makes the key: Ana's 92 gives "pass", Ben's 58 gives "fail", Chloe's 85 "pass", Dev's 40 "fail", Eli's 70 "pass". Each new key starts as an empty list, and every name is appended to its group, so "pass" comes first because Ana was first. `results["pass"]` has 3 names, and "merit" is not a key, so `get` gives the default "none".

Reread 8.10 *Grouping with a Dictionary* if your output differed.

Question 19

C. `stock["ink"] += 5` means `stock["ink"] = stock["ink"] + 5`, and the right side must read `stock["ink"]` first. There is no "ink" key, so Python stops with `KeyError: 'ink'`.

- **A** is safe: `get` gives the default 0 for a missing key.
- **B** is safe: assigning to a new key adds the entry; nothing is read first.
- **D** is safe: `in` only asks the question and prints `False`.

Reread 8.4 *Missing Keys: KeyError and get()* and the Watch Out in 8.9 *Counting with a Dictionary* if you chose differently.