

Day 6 Lists

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

By the end you will be able to

- Create lists, read items and slices by index, and change items in place.
- Add and remove items with `append`, `insert`, `extend`, `pop`, `remove`, and `clear`, and summarize a list with `len`, `in`, `min`, `max`, and `sum`.
- Choose between `sort()` and `sorted()`, and explain why changing one list can change "another" one.
- Store a grid as a list of lists and read or change any cell with two indexes.

What to remember from earlier

- Indexes start at 0, -1 is the last position, and a slice `[start:stop:step]` stops before stop. ([Day 3: Strings](#))
- Strings are immutable, so assigning to one position of a string is a `TypeError`. ([Day 3: Strings, 3.5 Strings Are Immutable](#))
- An empty list is falsy, so an `if` statement treats a list with nothing in it as `False`. ([Day 4: Making decisions, 4.7 Truthy and Falsy Values](#))
- `==` asks whether two values are equal, and `is` asks whether they are the very same object. ([Day 4: Making decisions, 4.8 Equality Versus Identity](#))
- An accumulator is set before a loop, updated inside it, and used after it. ([Day 5: Loops, 5.6 Counting and Summing](#))
- In a nested loop, the inner loop runs all the way through on every pass of the outer loop. ([Day 5: Loops, 5.10 Nested Loops](#))
- `for index, item in enumerate(sequence, 1)` gives each item with its position, counting from 1. ([Day 5: Loops, 5.11 enumerate\(\): Index and Value Together](#))

Make a `day06` folder for today's files, open it in your editor, and run each script from its terminal, as on Days 1 to 5.

Day 5's temperature program could report the highest of five days, but it could not print all five again afterwards: each value was forgotten as soon as the next one arrived. Today you learn the **list**, Python's way of keeping many values together, in order, under one name, so a program can store them, change them, and come back to them.

6.1 Making a List

A teacher has five quiz scores. With what you know so far, that means five variables: `score1`, `score2`, `score3`, `score4`, `score5`. Every line that totals them must name all five, and a loop cannot help, because there is no way to say "go through `score1` to `score5`". What you want is one name for the whole collection, like one sheet of paper with the scores written down in order.

A **list** is one value that holds several values in order. Each value inside it is an **item** (also called an **element**). You write a list as items separated by commas inside square brackets:

```
1 scores = [88, 92, 79]
```

- `[` and `]` mark where the list begins and ends.
- The commas separate the items. Here there are three: 88, 92, and 79.
- The order is kept exactly as you wrote it. `[88, 92, 79]` and `[79, 88, 92]` hold the same numbers but are different lists.

You met lists briefly on Day 3, as the result of `split()`, and on Day 4, where an empty list was one of the falsy values. Now you make them yourself. Try these in the REPL:

```
>>> scores = [88, 92, 79]
>>> scores
[88, 92, 79]
>>> len(scores)
3
>>> empty = []
>>> empty
[]
>>> len(empty)
0
>>> mixed = ["Ana", 15, 1.62, True, None]
>>> mixed
['Ana', 15, 1.62, True, None]
>>> "red green blue".split()
['red', 'green', 'blue']
>>> list("abc")
['a', 'b', 'c']
```

- `len()`, which counted the characters of a string on Day 3, counts the **items** of a list.
- `[]`, with nothing between the brackets, is the **empty list**: a list with no items yet. Its length is 0. Programs often start with an empty list and fill it as they go (6.4).
- A list can hold values of different types, even `True` and `None`. Most real lists hold one kind of thing, such as all scores or all names.
- `split()` still gives back a list, and `list()` converts another sequence into a list, just as `int()` and `str()` converted values on Day 2. `list("abc")` makes a list of the string's characters.

WORKED EXAMPLE Printing a list

fruit_list.py

```
1 fruits = ["apple", "banana", "cherry"]
2 print(fruits)
3 print(type(fruits))
4 print(f"I have {len(fruits)} fruits.")
```

```
['apple', 'banana', 'cherry']
<class 'list'>
I have 3 fruits.
```

- Line 2 prints the whole list. `print` shows each string **with quotes** inside the brackets, the way you would type it, so you can tell the string `'15'` from the number `15`.

- Line 3 uses `type()` from Day 2: a list's type is `list`. Line 4 puts `len(fruits)` in an f-string (Day 3).

Lists also work with `+` and `*`, the same way strings did on Day 3: `+` joins two lists into a new, longer list, and `*` repeats a list's items.

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
>>> ["ha"] * 3
['ha', 'ha', 'ha']
>>> [0] * 5
[0, 0, 0, 0, 0]
```

`[0] * 5` is a quick way to make a list of five zeros, for example five empty score slots.

KEY IDEA

A list holds several values, in order, under one name: `scores = [88, 92, 79]`. `len()` counts its items, and `[]` is the empty list.

WATCH OUT

A missing comma between strings. Python joins two strings that are written side by side with nothing between them, so a missing comma does not cause an error. It quietly makes one item out of two:

```
>>> snacks = ["chips" "salsa", "nuts"]
>>> snacks
['chipssalsa', 'nuts']
>>> len(snacks)
2
```

You meant three snacks and got two, and nothing warned you. When a list's length surprises you, look for a missing comma. (With numbers, `[88 92]` is a `SyntaxError`, because two numbers side by side mean nothing; Python points at the gap and asks whether you forgot a comma.)

PAUSE AND TRY

Question 1 Multiple choice

What is `len(["a b", "c", ""])`?

- A. 2
- B. 3
- C. 4
- D. 5

6.2 Indexing and Slicing Lists

Storing the scores is only half the job; you also need to get one back. Day 3 taught you to read one character of a string with an **index** in square brackets, counting from 0, and a part of a string with a **slice**. Lists use exactly the same rules, because a list, like a string, is a **sequence**: a collection of items kept in order, each at a numbered position. The only difference is what you get back: indexing a string gives a character, and indexing a list gives an item.

Position from the left	0	1	2	3	4
Item	88	92	79	95	60
Position from the end	-5	-4	-3	-2	-1

```
>>> scores = [88, 92, 79, 95, 60]
>>> scores[0]
88
>>> scores[4]
60
>>> scores[-1]
60
>>> scores[-2]
95
>>> scores[1:3]
[92, 79]
>>> scores[:2]
[88, 92]
>>> scores[2:]
[79, 95, 60]
>>> scores[::2]
[88, 79, 60]
>>> scores[::-1]
[60, 95, 79, 92, 88]
```

- `scores[0]` is the **first** item. The last item of a 5-item list is at index 4, which is `len(scores) - 1`, or simply `-1`.
- A slice `scores[start:stop]` gives the items from `start` up to, but **not including**, `stop`, just like a string slice and just like `range` (Day 5). `scores[1:3]` is index 1 and index 2: two items.
- Leaving out `start` means "from the beginning", leaving out `stop` means "to the end", and a step works as on Day 3: `[::2]` takes every second item, and `[::-1]` gives the items in reverse order.

A slice of a list is a **new list**, even when it holds only one item (`scores[0:1]` is `[88]`, a list, while `scores[0]` is `88`, a number). Indexing gives you one item; slicing always gives you a list.

WORKED EXAMPLE A week of temperatures

A list of seven daily temperatures, Monday first. Look at what each line asks for and whether the answer is one number or a list.

week_temps.py

```

1 temps = [14, 16, 15, 19, 21, 23, 20]
2 print("Monday:", temps[0])
3 print("Sunday:", temps[-1])
4 print("Midweek:", temps[2:5])
5 print("Weekend:", temps[5:])
6 print("Days recorded:", len(temps))

```

```

Monday: 14
Sunday: 20
Midweek: [15, 19, 21]
Weekend: [23, 20]
Days recorded: 7

```

Line	Expression	Positions	Result
2	temps[0]	0	14 (an item)
3	temps[-1]	6	20 (an item)
4	temps[2:5]	2, 3, 4	[15, 19, 21]
5	temps[5:]	5, 6	[23, 20]
6	len(temps)	(all)	7

Wednesday is index 2, and the slice 2:5 stops before Saturday's index 5. temps[5:] takes Saturday and Sunday without needing to know the length.

When the items are strings, you can index twice: once to pick the item, then again to pick a character from that string. Python works left to right, so names[2][0] means "item 2, then its character 0":

```

>>> names = ["Ana", "Ben", "Chloe"]
>>> names[2]
'Chloe'
>>> names[2][0]
'C'
>>> names[-1][-1]
'e'

```

names[2] is 'Chloe', and 'Chloe'[0] is 'C'. names[-1][-1] is the last character of the last name. You will use exactly this double indexing for grids in 6.10.

KEY IDEA

Lists index and slice exactly like strings: items[0] is the first item, items[-1] the last, and items[start:stop] is a new list that stops before stop. An index gives one item; a slice gives a list.

WATCH OUT

Using the length as an index. The last item is at len(scores) - 1, not len(scores):

index_past_end.py

```

1 scores = [88, 92, 79]
2 print("Last score:", scores[len(scores) - 1])
3 print("Next score:", scores[len(scores)])

```

```

Last score: 79
Traceback (most recent call last):
  File ".../day06/index_past_end.py", line 3, in <module>
    print("Next score:", scores[len(scores)])
                                ~~~~~^~~~~~
IndexError: list index out of range

```

The list has 3 items, at indexes 0, 1, and 2, so there is no index 3. This is the same **IndexError** you met with strings on Day 3, now saying "list" instead of "string". Use `scores[-1]` for the last item. A slice past the end, such as `scores[1:100]`, never causes an error: like a string slice, it just stops at the end. (Older Python versions may word error messages a little differently.)

PAUSE AND TRY

Question 2 Tracing

Predict exactly what the REPL shows for each line after the first.

```

>>> colors = ["red", "orange", "yellow", "green", "blue"]
>>> colors[-2]
>>> colors[1:4]
>>> colors[:2]
>>> colors[3][0]

```

6.3 Lists Are Mutable

A playlist has a typo in one song title. With a string, Day 3 taught you that you cannot fix one character in place: you have to build a whole new string. Is a list the same?

No, and this is the biggest difference between a list and a string. Put the two side by side:

```

>>> word = "cat"
>>> word[0] = "b"
Traceback (most recent call last):
...
TypeError: 'str' object does not support item assignment
>>> letters = ["c", "a", "t"]
>>> letters[0] = "b"
>>> letters
['b', 'a', 't']

```

Assigning to one position of a string is a `TypeError`. Assigning to one position of a list replaces that item. A list is **mutable**: its contents can be changed after it is made. A string is **immutable**: it can never change (Day 3). Writing `items[index] = value` is called **item assignment**.

Think of a list as a row of numbered boxes. `letters[0] = "b"` takes out what was in box 0 and puts "b" there instead. It is still the same list object (Day 4); only one box's contents changed.

Why make lists changeable? Because lists are usually collections that grow and change as a program runs, such as a shopping list or the scores so far, and rebuilding the whole collection for every change would be slow and awkward. Day 3 said immutable strings are safe to share; 6.9 shows the price lists pay for being changeable.

WORKED EXAMPLE Fixing a playlist

playlist_fix.py

```
1 playlist = ["Intro", "Sunrise", "Rainfal", "Night Drive"]
2 print("Before:", playlist)
3 playlist[2] = "Rainfall"
4 playlist[-1] = "Night Drive (Live)"
5 print("After:", playlist)
```

```
Before: ['Intro', 'Sunrise', 'Rainfal', 'Night Drive']
After: ['Intro', 'Sunrise', 'Rainfall', 'Night Drive (Live)']
```

- Line 3 replaces the item at index 2, the misspelled 'Rainfal', with the corrected title. Items 0, 1, and 3 are untouched.
- Line 4 uses a negative index on the left side: `playlist[-1]` is the last box, so the last title is replaced.
- Item assignment replaces an item; it never adds or removes one, so the list still has four items.

The right side of an item assignment can use the item's old value, exactly like `count = count - 1` on Day 2: work out the right side first, then store the result in the box. Augmented assignment from Day 5 works on items too:

bonus_points.py

```
1 scores = [70, 85, 90]
2 scores[0] += 5
3 scores[2] = scores[2] - 10
4 print(scores)
```

```
[75, 85, 80]
```

Line 2 means `scores[0] = scores[0] + 5`: read item 0 (70), add 5, store 75 back in box 0. Line 3 reads item 2 (90), subtracts 10, and stores 80. Item 1 is never mentioned, so it stays 85.

KEY IDEA

Lists are mutable: `items[index] = value` replaces the item at that position, in the same list. Strings are immutable, so the same assignment on a string is a `TypeError`.

WATCH OUT

Assigning past the end to add an item. It seems natural that assigning to the next free position would add an item there:

assign_past_end.py

```
1 seats = ["Ana", "Ben", "Chloe"]
2 seats[3] = "Dev"
3 print(seats)
```

```
Traceback (most recent call last):
  File "../day06/assign_past_end.py", line 2, in <module>
    seats[3] = "Dev"
    ~~~~~^^^
IndexError: list assignment index out of range
```

Item assignment only replaces an item that already exists, and the three-item list has no box 3. To add an item, use `append`, which is next, in 6.4.

PAUSE AND TRY**Question 3** Tracing

What does this program print? Track the list after each line.

shelf.py

```
1 shelf = ["jam", "tea", "rice", "oats"]
2 shelf[1] = "coffee"
3 shelf[-1] = shelf[0]
4 shelf[0] = "honey"
5 print(shelf)
6 print(len(shelf))
```

6.4 Adding Items: `append`, `insert`, `extend`

Most lists are not typed out in full at the start: a program collects names as people sign up, or scores as a teacher types them. It needs to **grow** the list.

Day 3 introduced the **method**: a function that belongs to a value, called with a dot, like `name.upper()`. String methods always gave back a **new** string, because strings cannot change. List methods are different: most of them **change the list itself**. Three of them add items:

Method	What it does
<code>items.append(value)</code>	Adds one value to the end
<code>items.insert(i, value)</code>	Puts the value at index i , shifting later items right
<code>items.extend(other)</code>	Adds each item of another sequence to the end

```
>>> guests = ["Ana", "Ben"]
>>> guests.append("Chloe")
>>> guests
['Ana', 'Ben', 'Chloe']
>>> guests.insert(0, "Dev")
>>> guests
['Dev', 'Ana', 'Ben', 'Chloe']
>>> guests.insert(2, "Eli")
>>> guests
['Dev', 'Ana', 'Eli', 'Ben', 'Chloe']
>>> guests.extend(["Fay", "Gus"])
>>> guests
['Dev', 'Ana', 'Eli', 'Ben', 'Chloe', 'Fay', 'Gus']
```

The REPL shows nothing after each call, because these methods change the list and give back nothing to show. `insert(0, "Dev")` moves everyone one place right, so Ana is now at index 1. If the index you give `insert` is past the end, the item simply goes at the end; `insert` never causes an `IndexError`.

WORKED EXAMPLE Collecting names until a blank line

The most common way to fill a list is to start with an empty list and `append` inside a loop. This is the Day 5 **accumulator** pattern, with a list as the accumulator: set it before the loop, add to it inside, use it after. This run types Ana, Ben, Chloe, then presses Enter on an empty line:

collect_names.py

```
1 names = []
2 while True:
3     name = input("Name (blank to finish): ").strip()
4     if name == "":
5         break
6     names.append(name)
7 print(names)
8 print(f"{len(names)} names collected.")
```

```
Name (blank to finish): Ana
Name (blank to finish): Ben
Name (blank to finish): Chloe
Name (blank to finish):
['Ana', 'Ben', 'Chloe']
3 names collected.
```

`names = []` comes **before** the loop; inside it, each pass would throw away the names so far, like the reset bug from Day 5. Unlike the Day 5 programs, this one still has every name after the loop, in the order they were typed.

`append` and `extend` look similar but treat their argument differently. `append` always adds **one** item, even if that item is itself a list. `extend` goes through its argument and adds **each** item:

```
>>> a = [1, 2, 3]
>>> a.append([4, 5])
>>> a
[1, 2, 3, [4, 5]]
>>> len(a)
4
>>> b = [1, 2, 3]
>>> b.extend([4, 5])
>>> b
[1, 2, 3, 4, 5]
>>> len(b)
5
>>> c = [1, 2, 3]
>>> c.extend("hi")
>>> c
[1, 2, 3, 'h', 'i']
>>> d = [1, 2]
>>> d + [3]
[1, 2, 3]
>>> d
[1, 2]
```

- `a.append([4, 5])` adds the whole list `[4, 5]` as a single item, so `a` has 4 items and the last one is a list. A list inside a list is called a **nested list**; 6.10 puts that to good use.
- `b.extend([4, 5])` adds 4 and 5 separately, so `b` has 5 items.
- `extend` accepts any sequence. A string is a sequence of characters, so `c.extend("hi")` adds `'h'` and `'i'`. To add the word as one item, use `c.append("hi")`.
- `d + [3]` builds a **new** list and leaves `d` unchanged, just as `+` on strings did. `append` and `extend` change the list itself. That difference matters a great deal in 6.9.

KEY IDEA

`append(value)` adds one item to the end, `insert(i, value)` adds one at index `i`, and `extend(sequence)` adds each item of a sequence. All three change the list itself and give back nothing.

WATCH OUT

Saving the result of `append`. String methods give back a new string, so `name = name.upper()` is the right habit. Carry that habit over to lists and the list disappears:

`append_returns_none.py`

```
1 names = ["Ana", "Ben"]
2 names = names.append("Chloe")
3 print(names)
```

None

append changes the list and gives back `None` (Day 2's "no value"). Line 2 then stores that `None` in `names`, and the list is lost. A list method that changes the list is called on its own line, with nothing assigned: `names.append("Chloe")`.

PAUSE AND TRY

Question 4 Multiple choice

What does this program print?

nested_append.py

```
1 nums = [1, 2]
2 nums.append([3, 4])
3 nums.extend([5])
4 print(len(nums))
```

- A. 3
- B. 4
- C. 5
- D. [1, 2, [3, 4], 5]

Question 5 Tracing

What does this program print? Track the list after each line.

line_up.py

```
1 line = ["Kai", "Lea"]
2 line.insert(1, "Max")
3 line.append("Nia")
4 line.insert(0, "Oto")
5 line.insert(10, "Pia")
6 print(line)
```

6.5 Removing Items: pop, remove, clear

A queue at a deli counter grows at the back and shrinks at the front. A program that keeps the queue as a list needs to **take items out**, sometimes by position ("serve whoever is first") and sometimes by value ("Ben left the queue").

Method	Removes	Gives back
<code>items.pop()</code>	The last item	The removed item
<code>items.pop(i)</code>	The item at index i	The removed item
<code>items.remove(value)</code>	The first item equal to <code>value</code>	<code>None</code>
<code>items.clear()</code>	Every item, leaving <code>[]</code>	<code>None</code>

```
>>> tasks = ["wash", "dry", "fold", "iron"]
>>> tasks.pop()
'iron'
>>> tasks
['wash', 'dry', 'fold']
>>> tasks.pop(0)
'wash'
>>> tasks
['dry', 'fold']
>>> tasks.remove("fold")
>>> tasks
['dry']
>>> tasks.clear()
>>> tasks
[]
>>> tasks.pop()
Traceback (most recent call last):
...
IndexError: pop from empty list
```

- `pop()` takes the last item out **and gives it back**, so the REPL shows `'iron'`, and you can store it: `last = tasks.pop()`.
- After `pop(0)`, everything shifts one place left, so `'dry'` is now at index 0.
- `remove("fold")` finds the value for you; you do not need to know where it is.
- `pop()` on an empty list is an `IndexError`: there is no last item to take.

WORKED EXAMPLE Serving a queue

deli_queue.py

```
1 queue = ["Ana", "Ben", "Chloe"]
2 queue.append("Dev")
3 served = queue.pop(0)
4 print("Now serving:", served)
5 served = queue.pop(0)
6 print("Now serving:", served)
7 print("Still waiting:", queue)
```

```
Now serving: Ana
Now serving: Ben
Still waiting: ['Chloe', 'Dev']
```

New customers join at the end with `append`, and `pop(0)` serves the person at the front, giving back the name so it can be printed. With `pop()` instead, the last person to arrive would be served first, like taking the top plate off a stack.

`remove` takes out only the **first** match, searching from the left:

```
>>> votes = ["yes", "no", "yes", "yes"]
>>> votes.remove("yes")
>>> votes
['no', 'yes', 'yes']
```

Only the first "yes" went. To remove every copy, keep removing while there is still one in the list. `in` asks whether a value is an item of a list, just as it asked whether one string was inside another on Day 3 (6.7 says more):

remove_all.py

```
1 votes = ["yes", "no", "yes", "maybe", "yes"]
2 while "yes" in votes:
3     votes.remove("yes")
4 print(votes)
```

```
['no', 'maybe']
```

Each pass removes one "yes". After the third, "yes" in votes is `False` and the loop ends. The `remove` call is the line that moves the condition toward `False`, which is what Day 5 said every while loop needs.

KEY IDEA

`pop()` removes and gives back the last item, and `pop(i)` the item at index `i`. `remove(value)` removes the first item equal to `value`. `clear()` removes everything.

WATCH OUT

Removing a value that is not there. `remove` must find the value, or it stops the program:

remove_missing.py

```
1 cart = ["bread", "milk"]
2 cart.remove("eggs")
3 print(cart)
```

```
Traceback (most recent call last):
  File "../day06/remove_missing.py", line 2, in <module>
    cart.remove("eggs")
    ~~~~~^~~~~~
ValueError: list.remove(x): x not in list
```

A `ValueError` (Day 2) means the type was fine but this particular value was not. The message says the value you asked for, called `x` in the message, is not in the list. When you cannot be sure the value is there, check with `in` first:

remove_missing_fixed.py

```
1 cart = ["bread", "milk"]
2 item = "eggs"
3 if item in cart:
4     cart.remove(item)
5 else:
6     print(f"{item} is not in the cart.")
7 print(cart)
```

```
eggs is not in the cart.  
['bread', 'milk']
```

PAUSE AND TRY

Question 6 Tracing

What does this program print? Track `plates`, `top`, and `second`.

plates.py

```
1 plates = ["red", "blue", "green", "white"]  
2 top = plates.pop()  
3 plates.remove("red")  
4 plates.append(top)  
5 second = plates.pop(0)  
6 print(top, second)  
7 print(plates)
```

Question 7 Debugging

This program should remove the song Sunrise from the playlist. It stops with an error. Quote the last line of the error, explain it, and fix the program so it can never crash on a missing song.

playlist_remove.py

```
1 playlist = ["Intro", "Sunrise", "Rainfall"]  
2 playlist.remove("sunrise")  
3 print(playlist)
```

6.6 Looping Over a List

Day 5's for loop went through a string one character at a time. Its real strength is lists: "do this for each guest", "check each score". A for loop over a list runs its body once for **each item**, with the loop variable holding the current item.

welcome.py

```
1 guests = ["Ana", "Ben", "Chloe"]  
2 for name in guests:  
3     print(f"Welcome, {name}!")  
4 print("Everyone is here.")
```

```
Welcome, Ana!  
Welcome, Ben!  
Welcome, Chloe!  
Everyone is here.
```

The loop variable `name` holds `'Ana'`, then `'Ben'`, then `'Chloe'`. Adding a guest to the list adds a welcome line without touching the loop.

WORKED EXAMPLE Building a list of passing scores

A loop can also **build** a new list while it goes through an old one. This picks out the scores of 60 or more:

passing.py

```
1 scores = [72, 45, 88, 59, 91]
2 passing = []
3 for score in scores:
4     if score >= 60:
5         passing.append(score)
6 print("Passing scores:", passing)
7 print(f"{len(passing)} of {len(scores)} passed.")
```

```
Passing scores: [72, 88, 91]
3 of 5 passed.
```

score	score >= 60	passing after
72	True	[72]
45	False	[72]
88	True	[72, 88]
59	False	[72, 88]
91	True	[72, 88, 91]

This is the accumulator pattern again, with the `if` deciding which passes add an item. The original `scores` list is never changed, so the program can report "3 of 5".

To number the items as you print them, use `enumerate()` from Day 5. It works on a list exactly as it did on a string:

numbered.py

```
1 chores = ["dishes", "laundry", "trash"]
2 for number, chore in enumerate(chores, 1):
3     print(f"{number}. {chore}")
```

```
1. dishes
2. laundry
3. trash
```

Changing every item. Day 5 warned that assigning to a loop variable does not change the string you are looping over. The same is true for a list:

double_attempt.py

```
1 prices = [4, 10, 7]
2 for price in prices:
3     price = price * 2
4 print(prices)
```

```
[4, 10, 7]
```

`price` is just a name for the current item. `price = price * 2` makes `price` refer to a new number; the box in the list still holds the old one. To change the boxes, you need their **indexes**, so loop over `range(len(prices))` (Day 5) and use item assignment (6.3):

`double_fixed.py`

```
1 prices = [4, 10, 7]
2 for i in range(len(prices)):
3     prices[i] = prices[i] * 2
4 print(prices)
```

`[8, 20, 14]`

`range(len(prices))` gives 0, 1, 2, the indexes of the three items, and `prices[i] = prices[i] * 2` replaces each box with double its value.

KEY IDEA

`for item in items:` runs once per item. To change items in place, loop over the indexes with `range(len(items))` and assign to `items[i]`. To keep only some items, build a new list.

WATCH OUT

Removing items from the list you are looping over. This should remove every failing score:

`remove_while_looping.py`

```
1 scores = [45, 50, 88, 30]
2 for score in scores:
3     if score < 60:
4         scores.remove(score)
5 print(scores)
```

`[50, 88]`

The 50 is still there. A for loop over a list keeps track of a **position**, moving one place right after each pass, and `remove` shifts every later item one place left:

Pass	Position	List at the start of the pass	score	Action
1	0	[45, 50, 88, 30]	45	Remove 45; 50 slides to position 0
2	1	[50, 88, 30]	88	Keep
3	2	[50, 88, 30]	30	Remove

When 45 was removed, 50 slid into position 0, which the loop had already visited, so 50 was never checked. Never add or remove items in a list while a for loop is going through it. Build a new list of the items to keep instead, as `passing.py` does.

PAUSE AND TRY**Question 8** Tracing

What does this program print?

long_words.py

```
1 words = ["sun", "moon", "star", "sky"]
2 long_words = []
3 for index, word in enumerate(words):
4     if len(word) > 3:
5         long_words.append(word.upper())
6     else:
7         words[index] = "-"
8 print(words)
9 print(long_words)
```

6.7 len, in, min, max, and sum

On Day 5 you wrote loops to count, to total, and to find the largest value. Once the values are in a list, Python has a **built-in function** (a function that is always available, like `print` and `len`) for each one.

```
>>> scores = [72, 45, 88, 59, 91]
>>> len(scores)
5
>>> min(scores)
45
>>> max(scores)
91
>>> sum(scores)
355
>>> sum(scores) / len(scores)
71.0
>>> 88 in scores
True
>>> 100 in scores
False
>>> 100 not in scores
True
>>> max(["pear", "Apple", "fig"])
'pear'
>>> min(["pear", "Apple", "fig"])
'Apple'
```

Call	Gives
<code>len(items)</code>	The number of items
<code>min(items)</code>	The smallest item
<code>max(items)</code>	The largest item

<code>sum(items)</code>	All the numbers added up
<code>value in items</code>	True if some item equals value
<code>value not in items</code>	True if no item equals value

- There is no built-in average, but `sum(scores) / len(scores)` is one.
- **not in** is the opposite of **in**: `100 not in scores` reads like English and means the same as `not (100 in scores)`.
- `min` and `max` compare items with the same rules as `<` and `>` from Day 4. For strings that means alphabetical order, with every capital before every small letter, so `max` of `"pear"`, `"Apple"`, and `"fig"` is `'pear'`, and `min` is `'Apple'`.

`in` on a list is not quite the `in` you used on strings. On a string, `in` looks for a piece of text anywhere inside. On a list, it asks whether one of the **items** is equal to the value, so a part of an item does not count:

```
>>> "app" in "apple"
True
>>> "app" in ["apple", "pear"]
False
>>> "apple" in ["apple", "pear"]
True
```

`"app"` is inside the string `"apple"`, but no item of the list is equal to `"app"`.

WORKED EXAMPLE Score statistics in four lines

Compare this with the Day 5 loops that counted, totaled, and tracked the highest value:

`score_stats.py`

```
1 scores = [72, 45, 88, 59, 91]
2 average = sum(scores) / len(scores)
3 print("Highest:", max(scores))
4 print("Lowest:", min(scores))
5 print(f"Average: {average:.1f}")
6 print("Anyone perfect?", 100 in scores)
```

```
Highest: 91
Lowest: 45
Average: 71.0
Anyone perfect? False
```

`355 / 5` is `71.0`, and line 6 prints the `bool` result of `in` directly. Every one of these lines would have been a loop with an accumulator on Day 5; the built-ins do the looping for you.

The empty list. A program that collects scores might end up with none. The built-ins do not all agree about what to do then:

```
>>> scores = []
>>> sum(scores)
0
>>> len(scores)
0
>>> max(scores)
Traceback (most recent call last):
...
ValueError: max() iterable argument is empty
>>> sum(scores) / len(scores)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

`sum` and `len` of an empty list are 0. But there is no largest item of nothing, so `max` (and `min`) stop with a `ValueError`; "iterable" in the message means "something you can loop over", here the list. And the average divides by 0, the `ZeroDivisionError` from Day 2. Guard with the Day 4 rule that an empty list is falsy:

safe_average.py

```
1 scores = []
2 if scores:
3     print(f"Average: {sum(scores) / len(scores):.1f}")
4 else:
5     print("No scores yet.")
```

No scores yet.

`if scores:` is `True` only when the list has at least one item, so the division runs only when it is safe.

KEY IDEA

`len`, `min`, `max`, and `sum` summarize a list, and `in` / `not in` ask whether a value is one of its items. On an empty list, `min` and `max` fail, so check `if items:` first.

WATCH OUT

Naming a variable `sum`. `sum` is a natural name for a total, and Python lets you use it. But that name already refers to the built-in function, and your assignment replaces it:

sum_shadow.py

```
1 sum = 0
2 for n in [3, 4, 5]:
3     sum += n
4 print(sum)
5 scores = [1, 2]
6 print(sum(scores))
```

```
12
Traceback (most recent call last):
  File ".../day06/sum_shadow.py", line 6, in <module>
    print(sum(scores))
          ~~~^^^^^^^^^^
TypeError: 'int' object is not callable
```

After line 1, `sum` refers to an `int`, not to the function. "Not callable" means "you cannot call it with parentheses": `12(scores)` makes no sense. Use names such as `total`, `highest`, and `lowest` for your own variables, and never `sum`, `min`, `max`, `len`, or `list`.

PAUSE AND TRY

Question 9 Multiple choice

What does this program print?

temp_range.py

```
1 temps = [3, -2, 8, 0]
2 print(max(temps) - min(temps), sum(temps) / len(temps))
```

- A. 10 2.25
- B. 6 2.25
- C. 10 2
- D. 8 2.25

6.8 Sorting and Reversing

A leaderboard shows scores from highest to lowest; a class register lists names alphabetically. Putting items in order is called **sorting**, and Python gives you two ways to do it that look almost alike but behave very differently.

- `items.sort()` is a list method. It sorts the list **in place**: it rearranges the items inside that same list and gives back `None`, like `append`.
- `sorted(items)` is a built-in function. It builds a **new** sorted list and gives it back, leaving the original list exactly as it was.

```
>>> scores = [72, 45, 88, 59]
>>> ranked = sorted(scores)
>>> ranked
[45, 59, 72, 88]
>>> scores
[72, 45, 88, 59]
>>> scores.sort()
>>> scores
[45, 59, 72, 88]
>>> sorted(scores, reverse=True)
[88, 72, 59, 45]
>>> scores.sort(reverse=True)
>>> scores
[88, 72, 59, 45]
```

After `sorted(scores)`, `scores` is still in the order the scores were entered; after `scores.sort()`, it is not. Both sort from smallest to largest. To sort largest first, add `reverse=True` inside the parentheses. This is a **keyword argument**: an argument passed with a name and `=`, so Python knows which setting you mean. `reverse=True` means "sort the other way round".

	<code>items.sort()</code>	<code>sorted(items)</code>
Kind	List method	Built-in function
Original list	Rearranged in place	Unchanged
Gives back	None	A new sorted list
Use when	You no longer need the old order	You need both orders, or want to use the result directly

Reversing is not sorting. The method `items.reverse()` flips the list's order in place, last item first, without comparing anything. A slice with step `-1` (6.2) gives a reversed **copy** instead:

```
>>> stops = ["Park", "Main", "Bridge", "Zoo"]
>>> stops.reverse()
>>> stops
['Zoo', 'Bridge', 'Main', 'Park']
>>> stops[::-1]
['Park', 'Main', 'Bridge', 'Zoo']
```

`reverse()` changed `stops` itself. The slice `stops[::-1]` gives a new list, reversed back, and leaves `stops` as it was.

WORKED EXAMPLE A top-three leaderboard

leaderboard.py

```
1 scores = [72, 45, 88, 59, 91, 66]
2 top_three = sorted(scores, reverse=True)[:3]
3 print("In order entered:", scores)
4 print("Top three:", top_three)
```

```
In order entered: [72, 45, 88, 59, 91, 66]
Top three: [91, 88, 72]
```

Line 2 works from the inside out: `sorted(scores, reverse=True)` builds the new list `[91, 88, 72, 66, 59, 45]`, and `[:3]` slices its first three items. `scores` is untouched, so line 3 still shows the order the scores were entered in.

Sorting uses the same comparisons as `min` and `max`, which brings two surprises:

```
>>> names = ["bea", "Ana", "cal", "Ben"]
>>> sorted(names)
['Ana', 'Ben', 'bea', 'cal']
>>> sorted(["9", "10", "100"])
['10', '100', '9']
>>> sorted([9, 10, 100])
[9, 10, 100]
>>> sorted([3, "10", 7])
Traceback (most recent call last):
...
TypeError: '<' not supported between instances of 'str' and 'int'
```

- Capital letters come before all small letters (Day 4), so `'Ben'` sorts before `'bea'`.
- Numbers stored as **strings** sort as text, character by character: `"10"` comes before `"9"` because `"1"` comes before `"9"`. Numbers typed with `input()` are strings until you convert them with `int()` or `float()` (Day 3), so convert before you sort.
- A list that mixes numbers and strings cannot be sorted at all: `<` does not work between a `str` and an `int`, so Python stops with a `TypeError`.

KEY IDEA

`items.sort()` rearranges the list itself and gives back `None`; `sorted(items)` gives back a new sorted list and leaves the original alone. `reverse=True` sorts largest first, and `items.reverse()` flips the order without sorting.

WATCH OUT

Saving the result of `sort()`. It reads naturally, but like `append`, `sort()` gives back `None`:

`sort_returns_none.py`

```
1 scores = [72, 45, 88]
2 ranked = scores.sort()
3 print(ranked)
4 print(scores)
```

```
None
[45, 72, 88]
```

`scores` was sorted, but `ranked` is `None`. If you want a sorted result to store, use `ranked = sorted(scores)`. If you want the list itself sorted, write `scores.sort()` on its own line.

PAUSE AND TRY**Question 10** Multiple choice

What does this program print?

cheap_first.py

```
1 prices = [5, 2, 9]
2 cheap_first = sorted(prices)
3 prices.reverse()
4 print(prices, cheap_first)
```

- A. [9, 5, 2] [2, 5, 9]
- B. [9, 2, 5] [2, 5, 9]
- C. [9, 2, 5] [9, 5, 2]
- D. [2, 5, 9] [2, 5, 9]

6.9 Aliasing Versus Copying

Here is a puzzle. A program makes a weekend routine by starting from the weekday routine and changing it: no alarm, and pancakes at the end. It only changes `weekend`. Read it and predict what it prints for `weekday`:

alias_demo.py

```
1 weekday = ["alarm", "breakfast", "bus"]
2 weekend = weekday
3 weekend.remove("alarm")
4 weekend.append("pancakes")
5 print("Weekend:", weekend)
6 print("Weekday:", weekday)
7 print(weekend is weekday)
```

```
Weekend: ['breakfast', 'bus', 'pancakes']
Weekday: ['breakfast', 'bus', 'pancakes']
True
```

The weekday routine lost its alarm and gained pancakes, even though no line mentions `weekday` after line 1. This is the day's most important idea, and it explains why changing one list can change "another" one.

Assignment never copies. Day 4 said that every value is an object in memory and that a variable name refers to an object. `weekend = weekday` does not make a second list. It makes the name `weekend` refer to the **same** list object that `weekday` refers to. There is one list with two names:

Name	Refers to
weekday	List object #1: ['alarm', 'breakfast', 'bus']
weekend	List object #1 (the same one)

Two names for the same object are **aliases**, and this situation is **aliasing**, like a person who answers to both "Robert" and "Bob". `weekend.remove("alarm")` changes object #1, so `weekday` sees the change. Line 7 confirms it with Day 4's `is`.

Why did this never bite you with numbers or strings? Because they are immutable (Day 3): nothing can change them in place, so it never matters how many names share one. Lists are mutable, so a change made through one name shows up through every alias.

Copying. To get a second, separate list, ask for a copy. The method `items.copy()` builds a new list holding the same items:

WORKED EXAMPLE A weekend routine that leaves the weekday alone

`copy_fixed.py`

```
1 weekday = ["alarm", "breakfast", "bus"]
2 weekend = weekday.copy()
3 weekend.remove("alarm")
4 weekend.append("pancakes")
5 print("Weekend:", weekend)
6 print("Weekday:", weekday)
7 print(weekend is weekday)
```

```
Weekend: ['breakfast', 'bus', 'pancakes']
Weekday: ['alarm', 'breakfast', 'bus']
False
```

Only line 2 changed. Now there are two list objects, so changes through `weekend` reach only the copy, and `weekday` keeps its alarm.

There are three common ways to copy a list, and they all do the same job:

```
>>> a = [1, 2, 3]
>>> b = a
>>> c = a.copy()
>>> b is a
True
>>> c is a
False
>>> c == a
True
>>> d = a[:]
>>> d is a
False
>>> e = list(a)
>>> e is a
False
```

- `b = a` makes an alias: `b is a` is `True`.
- `a.copy()`, the full slice `a[:]` (every slice is a new list, 6.2), and `list(a)` each build a new list: `is` gives `False`, while `==` gives `True` because the contents are equal. This is Day 4's equality versus identity, now with a reason to care.

Rebinding is not changing. Aliasing only shows when the object itself is changed in place. Assigning a new object to one of the names, called **rebinding** (Day 2), moves that one name and leaves the other alone:

```
>>> x = 5
>>> y = x
>>> x = x + 1
>>> y
5
>>> a = [5]
>>> b = a
>>> a = a + [6]
>>> a
[5, 6]
>>> b
[5]
```

`a = a + [6]` builds a **new** list (+ always does, 6.4) and points `a` at it. `b` still refers to the original `[5]`. Compare `a.append(6)`, which would change the shared list, so `b` would show `[5, 6]` too.

Changes the list in place (aliases see it)	Makes a new list (aliases do not see it)
<code>items[i] = value</code>	<code>items + other</code>
<code>append, insert, extend</code>	<code>items[start:stop]</code>
<code>pop, remove, clear</code>	<code>sorted(items)</code>
<code>sort, reverse</code>	<code>items.copy(), list(items)</code>

One more detail: a copy is a new list, but the **items** in it are the same objects as in the original. For numbers and strings this never matters, because nothing can change them. When the items are themselves lists, it does; 6.10 has the details.

KEY IDEA

`b = a` makes `b` another name for the same list, so changes through either name show in both. `b = a.copy()` (or `a[:]`, or `list(a)`) makes a separate list. Use `is` to check: aliases are the same object.

WATCH OUT

Making a "backup" with `=`. `backup = scores` looks like it saves the old scores before you change the list. It does not: it gives the same list a second name, so every change to `scores` also appears in `backup`, and the "backup" is useless. Use `backup = scores.copy()` whenever you want to keep the old contents.

PAUSE AND TRY

Question 11 Tracing

What does this program print? Before you trace, decide how many list objects there are.

backup.py

```

1 original = [3, 1, 2]
2 backup = original
3 sorted_copy = sorted(original)
4 original.sort()
5 backup.append(4)
6 print(original)
7 print(backup)
8 print(sorted_copy)

```

Question 12 Multiple choice

a refers to a list. After which line can b be changed without changing a?

- A. b = a
- B. b = a[:]
- C. b = a.sort()
- D. b = a.append(0)

6.10 Nested Lists as a Grid

A tic-tac-toe board, a seating chart, and a spreadsheet of test scores all have **rows and columns**. One list is a single row of boxes. A grid needs rows of rows.

6.4 showed that a list can hold another list as an item. A list whose items are all lists of the same length is a **grid** (also called a **2D list**, for two dimensions): the outer list holds the rows, and each row is a list of values. To reach one value, you index twice, as you did with `names[2][0]` in 6.2: first the row, then the column.

```

>>> grid = [[1, 2, 3], [4, 5, 6]]
>>> grid[0]
[1, 2, 3]
>>> grid[1][2]
6
>>> len(grid)
2
>>> len(grid[0])
3
>>> grid[1][0] = 40
>>> grid
[[1, 2, 3], [40, 5, 6]]

```

	column 0	column 1	column 2
row 0	1	2	3
row 1	4 (then 40)	5	6

- `grid[0]` is the whole first row, `[1, 2, 3]`.
- `grid[1][2]` is evaluated left to right: `grid[1]` is the row `[4, 5, 6]`, and its item 2 is 6. Read it as "row 1, column 2".

- `len(grid)` is the number of **rows**, and `len(grid[0])`, the length of one row, is the number of **columns**.
- `grid[1][0] = 40` picks row 1, then replaces its item 0. Rows are lists, so they are mutable.

Writing a grid with one row per line, as below, makes the rows and columns easy to see. Python allows a list to continue over several lines until its closing bracket, and a comma after the last item is allowed.

WORKED EXAMPLE A tic-tac-toe board

"." marks an empty square.

tic_tac_toe.py

```
1 board = [
2     ["X", "O", "."],
3     [".", "X", "."],
4     ["O", ".", "."],
5 ]
6 board[2][2] = "X"
7 for row in board:
8     print(" ".join(row))
9 print("Center:", board[1][1])
```

```
X O .
. X .
O . X
Center: X
```

- Line 6 puts an X in row 2 (the bottom row), column 2 (the right square).
- The loop on line 7 goes through the outer list, so `row` is one whole row on each pass: a list of three strings. `" ".join(row)` (Day 3) glues the three strings into one line with spaces between them.
- `board[1][1]` is the middle row's middle square.

When you need the row and column **numbers**, for example to label rows or to do arithmetic with them, use nested loops (Day 5) over `range(len(...))`: the outer loop picks each row index, and the inner loop goes through every column index of that row.

gradebook.py

```
1 scores = [
2     [90, 85, 77],
3     [60, 72, 81],
4 ]
5 for row in range(len(scores)):
6     total = 0
7     for col in range(len(scores[row])):
8         total += scores[row][col]
9     print(f"Student {row + 1}: total {total}")
```

```
Student 1: total 252
Student 2: total 213
```

`total = 0` is inside the outer loop, so it starts again for each student, and `print` is at the outer loop's level, so it runs once per row, after the inner loop: the same placement rules as Day 5's `dice_sums.py`. Each row is a list of numbers, so `sum(scores[row])` would give the same totals in one step; the nested loops show how `scores[row][col]` reaches every cell.

KEY IDEA

A grid is a list of rows, each row a list. `grid[row][col]` reads or changes one cell; `len(grid)` counts the rows and `len(grid[0])` the columns.

WATCH OUT

Repeating a row with `*`. `[0] * 5` made five zeros in 6.1, so `[["."] * 3] * 3` looks like a quick 3×3 board:

`grid_repeat_bug.py`

```
1 board = [["."] * 3] * 3
2 board[0][0] = "X"
3 for row in board:
4     print(" ".join(row))
```

```
X . .
X . .
X . .
```

One assignment put an `X` in all three rows. `* 3` repeats the **items** of the outer list, and its one item is a reference to one row list. The result is a list holding the same row three times: three aliases (6.9). Changing "row 0" changes that one shared row, and the other two "rows" are the same object. Build a separate row on each pass instead:

`grid_loop_fixed.py`

```
1 board = []
2 for row in range(3):
3     board.append(["."] * 3)
4 board[0][0] = "X"
5 for row in board:
6     print(" ".join(row))
```

```
X . .
. . .
. . .
```

`[["."] * 3]` runs once per pass, so each `append` adds a new, separate row. (`*` was fine inside each row, because its items are strings, which cannot change.)

PAUSE AND TRY**Question 13** Tracing

What does this program print?

grid_trace.py

```
1 grid = [[1, 2], [3, 4], [5, 6]]
2 grid[2][0] = grid[0][1] + grid[1][1]
3 column_total = 0
4 for row in grid:
5     column_total += row[0]
6 print(grid[-1])
7 print(column_total)
```

Putting it together

A shopping list program keeps asking for a command until the user types `done`. `add` asks for an item and adds it unless it is already there, `remove` takes an item off if it is there, and `show` prints the list in alphabetical order with numbers. This run adds milk, bread, Milk (again, with a capital), and apples, shows the list, tries to remove eggs, removes milk, and finishes:

shopping_list.py

```
1 shopping = []
2 while True:
3     command = input("Command (add, remove, show, done): ").strip().lower()
4     if command == "done":
5         break
6     if command == "add":
7         item = input("Item: ").strip().lower()
8         if item in shopping:
9             print(f"{item} is already on the list.")
10        else:
11            shopping.append(item)
12    elif command == "remove":
13        item = input("Item: ").strip().lower()
14        if item in shopping:
15            shopping.remove(item)
16        else:
17            print(f"{item} is not on the list.")
18    elif command == "show":
19        for number, item in enumerate(sorted(shopping), 1):
20            print(f"{number}. {item}")
21    else:
22        print("Unknown command.")
23    print(f"{len(shopping)} items: {' '.join(sorted(shopping))}")
```

```

Command (add, remove, show, done): add
Item: milk
Command (add, remove, show, done): add
Item: bread
Command (add, remove, show, done): add
Item: Milk
milk is already on the list.
Command (add, remove, show, done): add
Item: apples
Command (add, remove, show, done): show
1. apples
2. bread
3. milk
Command (add, remove, show, done): remove
Item: eggs
eggs is not on the list.
Command (add, remove, show, done): remove
Item: milk
Command (add, remove, show, done): done
2 items: apples, bread

```

Typed	What happens	shopping after	Idea
add, milk	Not in the list: append	['milk']	6.4, 6.7
add, bread	Not in the list: append	['milk', 'bread']	6.4
add, Milk	lower() makes it milk, which is in the list	['milk', 'bread']	6.7
add, apples	append	['milk', 'bread', 'apples']	6.4
show	enumerate(sorted(shopping), 1) numbers a sorted copy	unchanged	6.6, 6.8
remove, eggs	Not in the list: message, no ValueError	unchanged	6.5
remove, milk	remove	['bread', 'apples']	6.5
done	break	unchanged	Day 5

- Every item is lowercased (Day 3) before it is checked or stored, so Milk and milk count as the same item, and in (which compares whole items exactly) catches the duplicate.
- show uses sorted, not sort, so the list keeps the order items were added in; only the display is alphabetical.

Day 12 builds a to-do list manager with the same shape: a list that a menu loop adds to, shows, and removes from.

Vocabulary

Term	Definition	Example
list	One value holding several values in order, in square brackets	[88, 92, 79]
item / element	One value inside a list	92

empty list	A list with no items	<code>[]</code>
sequence	A collection of items kept in order at numbered positions	a string or a list
mutable / immutable	Can / cannot be changed after it is made	lists / strings
item assignment	Replacing the item at one position	<code>scores[0] = 90</code>
append()	Adds one item to the end of a list	<code>names.append("Ana")</code>
insert()	Adds one item at an index, shifting later items right	<code>names.insert(0, "Dev")</code>
extend()	Adds each item of another sequence to the end	<code>a.extend([4, 5])</code>
pop()	Removes and gives back the last item, or the item at an index	<code>queue.pop(0)</code>
remove()	Removes the first item equal to a value	<code>cart.remove("milk")</code>
clear()	Removes every item	<code>tasks.clear()</code>
built-in function	A function that is always available	<code>len, min, max, sum</code>
not in	True when no item equals the value	<code>100 not in scores</code>
sorting	Putting items in order	<code>[45, 72, 88]</code>
sort()	Sorts a list in place and gives back <code>None</code>	<code>scores.sort()</code>
sorted()	Gives back a new sorted list, leaving the original alone	<code>sorted(scores)</code>
keyword argument	An argument passed with a name and <code>=</code>	<code>reverse=True</code>
reverse()	Flips a list's order in place, without sorting	<code>stops.reverse()</code>
alias / aliasing	Two names for the same object	<code>b = a</code>
copy	A new, separate list with the same items	<code>a.copy(), a[:], list(a)</code>
nested list	A list that is an item of another list	<code>[1, [2, 3]]</code>
grid / 2D list	A list of rows, each row a list	<code>[[1, 2], [3, 4]]</code>

Summary

- A **list** holds several items in order under one name. It indexes and slices exactly like a string: `items[0]` is the first item, `items[-1]` the last, and a slice is a new list.
- Lists are **mutable**: `items[i] = value` replaces an item, `append`, `insert`, and `extend` add items, and `pop`, `remove`, and `clear` take them out. These methods change the list and give back `None` (except `pop`, which gives back the removed item), so never assign their result.
- Loop over a list with `for item in items;` build a new list to keep some items, and loop over `range(len(items))` to change items in place. `len`, `in`, `min`, `max`, and `sum` summarize a list without writing a loop.
- `items.sort()` sorts the list itself; `sorted(items)` gives back a new sorted list and leaves the original alone.
- `b = a` makes an **alias**, a second name for the same list, so changing one changes "the other". Use `a.copy()` for a separate list. A **grid** is a list of lists, read with `grid[row][col]`.

Practice

Do these in your `day06` folder. Type each program yourself, run it, and only then open the answer.

Question 14 Short answer

Min, max, and average. Ask the user for five numbers, one at a time, and store them in a list. Then report the smallest, the largest, and the average with two decimal places.

Question 15 Short answer

Remove duplicates. Start with a list that has repeated items, such as `["apple", "pear", "apple", "fig", "pear", "kiwi"]`. Make a list with each item only once, keeping the order in which each item first appeared.

Question 16 Short answer

The aliasing bug. Write a program that keeps a list of regular prices and makes a list of sale prices at 20% off (multiply by 0.8). First write it with `sale_prices = prices` and show that the regular prices change too. Then fix it.

Review questions

Question 17 Tracing

What does this program print? Track the list after each line.

`line_shuffle.py`

```
1 line = ["Ana", "Ben", "Cy"]
2 line.extend(["Dot", "Eve"])
3 first = line.pop(0)
4 line.insert(1, first)
5 print(line[1:3])
6 print(line[-1], len(line))
```

Question 18 Debugging

This program should print the top score. It stops with an error. Quote the last line of the error, explain it, and write the fix.

`top_score_bug.py`

```
1 scores = [72, 91, 88]
2 ranked = scores.sort(reverse=True)
3 print("Top score:", ranked[0])
```

Question 19 Tracing

What does this program print? Decide, line by line, whether each line changes a list or makes a new one.

`rebind_or_change.py`

```
1 a = [1, 2]
2 b = a
3 a = a + [3]
4 b.append(4)
5 print(a)
6 print(b)
```

Question 20 Multiple choice

scores holds some test scores. Which code leaves passing holding exactly the scores of 60 or more, with none missed?

- A. `for score in scores: with if score < 60: scores.remove(score)` inside, then `passing = scores`
- B. `passing = []`, then `for score in scores: with if score >= 60: passing.append(score)` inside
- C. `passing = scores.remove(score < 60)`
- D. `passing = []`, then `for score in scores: with passing = [score]` inside when `score >= 60`

Question 21 Tracing

What does this program print? Think about what `copy()` copies when the items are lists.

grid_copy.py

```
1 grid = [[1, 2], [3, 4]]
2 backup = grid.copy()
3 grid[0][0] = 99
4 grid.append([5, 6])
5 print(backup)
6 print(len(grid), len(backup))
```

Question 22 Short answer

A program reads three ages with `input()` and appends each one to `ages` without converting it. The user types 9, 10, and 100. Then it prints `sorted(ages)` and gets `['10', '100', '9']`. Explain why the ages are in this order, and say what to change so they sort as numbers.

Go further

The official Python Tutorial introduces lists in [An Informal Introduction to Python: Lists and lists](#) every list method in [Data Structures: More on Lists](#). Its examples also use tuples and list comprehensions, which you meet on Day 7 and Day 11.

Reflection

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

- ☐ I can make a list, including an empty one, and count its items with `len()`. [Reread: 6.1 Making a List](#)
- ☐ I can read any item or slice of a list, including with negative indexes. [Reread: 6.2 Indexing and Slicing Lists](#)
- ☐ I can change an item in place and explain why that fails for a string. [Reread: 6.3 Lists Are Mutable](#)
- ☐ I can add items with `append`, `insert`, and `extend`, and explain how `append` and `extend` differ. [Reread: 6.4 Adding Items: append, insert, extend](#)
- ☐ I can remove items with `pop`, `remove`, and `clear` without causing an error. [Reread: 6.5 Removing Items: pop, remove, clear](#)
- ☐ I can loop over a list, change its items by index, and build a new list of the items to keep. [Reread: 6.6 Looping Over a List](#)
- ☐ I can summarize a list with `len`, `in`, `min`, `max`, and `sum`. [Reread: 6.7 len, in, min, max, and sum](#)

- ☐ I can choose between `sort()` and `sorted()` and explain the difference. Reread: 6.8 Sorting and Reversing
- ☐ I can explain why changing one list can change "another" one, and make a real copy. Reread: 6.9 Aliasing Versus Copying
- ☐ I can read and change any cell of a grid with two indexes. Reread: 6.10 Nested Lists as a Grid

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. The commas decide how many items there are, and there are three items: the string "a b", the string "c", and the empty string "".

```
>>> len(["a b", "c", ""])
3
```

- **A** is wrong: the empty string "" is still an item. It is a string with no characters, but it occupies a place in the list.
- **C** is wrong: it splits "a b" at its space. `len()` of a list counts items, not words; the space is just a character inside one string.
- **D** is wrong: it counts characters (a, space, b, c), but `len()` of a list counts items, not the characters inside them.

Reread 6.1 *Making a List* if you chose differently.

Question 2

```
>>> colors = ["red", "orange", "yellow", "green", "blue"]
>>> colors[-2]
'green'
>>> colors[1:4]
['orange', 'yellow', 'green']
>>> colors[:2]
['red', 'yellow', 'blue']
>>> colors[3][0]
'g'
```

- `-2` is the second item from the end: 'green'. The REPL shows a single string item with quotes.
- `1:4` is indexes 1, 2, and 3: three items, as a list. Index 4, 'blue', is the stop and is left out.
- `::2` takes indexes 0, 2, and 4.
- `colors[3]` is 'green', and its character 0 is 'g'.

Reread 6.2 *Indexing and Slicing Lists* if any answer differed.

Question 3

```
['honey', 'coffee', 'rice', 'jam']
4
```

Line	Change	shelf after
1	Make the list	['jam', 'tea', 'rice', 'oats']
2	Box 1 becomes 'coffee'	['jam', 'coffee', 'rice', 'oats']
3	Last box gets a copy of shelf[0]	['jam', 'coffee', 'rice', 'jam']
4	Box 0 becomes 'honey'	['honey', 'coffee', 'rice', 'jam']

Line 3 reads `shelf[0]` **at that moment**, when it is still `'jam'`. Changing box 0 on line 4 does not go back and change box 3. The length stays 4, because item assignment replaces items and never adds any.

Reread 6.3 *Lists Are Mutable* if your output differed.

Question 4

B. `append([3, 4])` adds the list `[3, 4]` as **one** item, so `nums` is `[1, 2, [3, 4]]`, 3 items.
`extend([5])` adds each item of `[5]`, which is one item, 5. Now `nums` is `[1, 2, [3, 4], 5]`: 4 items.

- **A** is wrong: it forgets that `extend` adds 5.
- **C** is wrong: it treats `append([3, 4])` as adding 3 and 4 separately, which is what `extend` does.
- **D** is wrong: that is what `nums` holds, but the program prints `len(nums)`, which is a number: how many items the list has.

Reread 6.4 *Adding Items: append, insert, extend* if you chose differently.

Question 5

```
['Oto', 'Kai', 'Max', 'Lea', 'Nia', 'Pia']
```

Line	Call	line after
2	<code>insert(1, "Max")</code>	<code>['Kai', 'Max', 'Lea']</code>
3	<code>append("Nia")</code>	<code>['Kai', 'Max', 'Lea', 'Nia']</code>
4	<code>insert(0, "Oto")</code>	<code>['Oto', 'Kai', 'Max', 'Lea', 'Nia']</code>
5	<code>insert(10, "Pia")</code>	<code>['Oto', 'Kai', 'Max', 'Lea', 'Nia', 'Pia']</code>

Index 10 is past the end of a 5-item list, so `insert` puts Pia at the end instead of causing an error.

Reread 6.4 *Adding Items: append, insert, extend* if your output differed.

Question 6

```
white blue
['green', 'white']
```

Line	Call	Gives back	plates after
2	<code>pop()</code>	<code>'white'</code>	<code>['red', 'blue', 'green']</code>
3	<code>remove("red")</code>	<code>None</code>	<code>['blue', 'green']</code>
4	<code>append(top)</code>	<code>None</code>	<code>['blue', 'green', 'white']</code>
5	<code>pop(0)</code>	<code>'blue'</code>	<code>['green', 'white']</code>

`pop()` with no index takes the **last** plate. After `remove("red")`, blue is at index 0, so `pop(0)` gives `'blue'`.

Reread 6.5 *Removing Items: pop, remove, clear* if your output differed.

Question 7

```
ValueError: list.remove(x): x not in list
```

`remove` looks for an item **equal** to `"sunrise"`, and string comparison is case-sensitive (Day 4): `"sunrise" == "Sunrise"` is `False`. No item matches, so `remove` stops the program. Use the exact title, and check with `in` before removing so a missing song prints a message instead of crashing:

playlist_remove_fixed.py

```
1 playlist = ["Intro", "Sunrise", "Rainfall"]
2 song = "Sunrise"
3 if song in playlist:
4     playlist.remove(song)
5 else:
6     print(f"{song} is not in the playlist.")
7 print(playlist)
```

```
['Intro', 'Rainfall']
```

Reread 6.5 *Removing Items: pop, remove, clear* if you missed it.

Question 8

```
['-', 'moon', 'star', '-']
['MOON', 'STAR']
```

index	word	len(word) > 3	Action
0	'sun'	False	words[0] becomes '-'
1	'moon'	True	Append 'MOON' to long_words
2	'star'	True	Append 'STAR' to long_words
3	'sky'	False	words[3] becomes '-'

Replacing an item by its index is safe during a loop, because the list keeps the same length and nothing shifts. `upper()` gives a new string, which is what gets appended; the items in `words` stay lowercase.

Reread 6.6 *Looping Over a List* if your output differed.

Question 9

A. `max(temps)` is 8 and `min(temps)` is -2, so the range is $8 - (-2) = 10$. `sum(temps)` is $3 + (-2) + 8 + 0 = 9$, and $9 / 4$ is 2.25.

- **B** is wrong: it subtracts 2 instead of -2. Subtracting a negative number adds it.
- **C** is wrong: `/` always gives a float (Day 2), so the average is 2.25, not 2.
- **D** is wrong: it takes the smallest value as 0, but -2 is smaller than 0.

Reread 6.7 *len, in, min, max, and sum* if you chose differently.

Question 10

B. `sorted(prices)` builds a new list `[2, 5, 9]` and leaves `prices` as `[5, 2, 9]`. Then `prices.reverse()` flips `prices` in place, without sorting, giving `[9, 2, 5]`. `cheap_first` is a separate list, so the reverse does not touch it.

- **A** is wrong: `reverse()` does not sort largest first; it only flips the current order.
- **C** is wrong: `cheap_first` was made by `sorted` before the reverse, and it is a different list from `prices`.

- **D** is wrong: `sorted` does not change `prices`.

Reread 6.8 *Sorting and Reversing* if you chose differently.

Question 11

```
[1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3]
```

There are **two** list objects. `original` and `backup` are aliases of the first; `sorted` builds the second, `[1, 2, 3]`, which only `sorted_copy` refers to.

Line	Action	Shared list (<code>original</code> , <code>backup</code>)	<code>sorted_copy</code>
1–2	Make, alias	<code>[3, 1, 2]</code>	
3	<code>sorted(original)</code>	<code>[3, 1, 2]</code>	<code>[1, 2, 3]</code>
4	<code>original.sort()</code>	<code>[1, 2, 3]</code>	<code>[1, 2, 3]</code>
5	<code>backup.append(4)</code>	<code>[1, 2, 3, 4]</code>	<code>[1, 2, 3]</code>

`sort` and `append` change the shared list, so both names show `[1, 2, 3, 4]`. `sorted_copy` is a separate list, so the `append` does not reach it.

Reread 6.9 *Aliasing Versus Copying* if your output differed.

Question 12

B. A full slice builds a new list with the same items, so `b is a` is `False`, and changes through `b` affect only the new list.

- **A** is wrong: `b = a` is an alias; both names refer to one list.
- **C** is wrong: `sort()` sorts `a` in place and gives back `None`, so `b` is `None`, not a list at all.
- **D** is wrong: `append` changes `a` and gives back `None`, so `b` is `None`.

Reread 6.9 *Aliasing Versus Copying* if you chose differently.

Question 13

```
[6, 6]
10
```

Line 2 works out the right side first: `grid[0][1]` is row 0, column 1, which is 2, and `grid[1][1]` is 4, so it stores 6 in row 2, column 0. The grid is now `[[1, 2], [3, 4], [6, 6]]`.

The loop goes through the rows, and `row[0]` is each row's first item: 1, 3, then 6. `column_total` is `1 + 3 + 6 = 10`. `grid[-1]` is the last row, `[6, 6]`.

Reread 6.10 *Nested Lists as a Grid* if your output differed.

Question 14

One possible solution, with a run that types 12, 7.5, 20, 3, and 9:

five_numbers.py

```
1 numbers = []
2 for count in range(1, 6):
3     number = float(input(f"Number {count}: "))
4     numbers.append(number)
5 average = sum(numbers) / len(numbers)
6 print("Min:", min(numbers))
7 print("Max:", max(numbers))
8 print(f"Average: {average:.2f}")
```

```
Number 1: 12
Number 2: 7.5
Number 3: 20
Number 4: 3
Number 5: 9
Min: 3.0
Max: 20.0
Average: 10.30
```

The list starts empty before the loop, and each pass converts the typed string with `float()` (so 7.5 works) and appends it. After the loop, `min`, `max`, and `sum(numbers) / len(numbers)` do the work. Converting **before** appending matters: a list of strings would compare as text, and `sum` cannot add strings. The total is 51.5, and 51.5 / 5 is 10.3, shown as 10.30 by `:.2f`.

Reread 6.4 *Adding Items: append, insert, extend* and 6.7 *len, in, min, max, and sum* if your result differed.

Question 15

One possible solution:

remove_duplicates.py

```
1 items = ["apple", "pear", "apple", "fig", "pear", "kiwi"]
2 unique = []
3 for item in items:
4     if item not in unique:
5         unique.append(item)
6 print(f"Before: {items}")
7 print(f"After: {unique}")
```

```
Before: ['apple', 'pear', 'apple', 'fig', 'pear', 'kiwi']
After:  ['apple', 'pear', 'fig', 'kiwi']
```

The loop goes through the items in order and appends an item to `unique` only if it is `not in unique` yet. The first `apple` and the first `pear` are added; their second copies are already in `unique`, so they are skipped. Because items are added in the order they are first met, the original order is kept. Building a new list, rather than removing duplicates from `items` while looping over it, avoids the skipping problem from 6.6. Sorting the list first would also group duplicates, but would lose the original order.

Reread 6.6 *Looping Over a List* if your result differed.

Question 16

The bug, on purpose:

sale_bug.py

```

1 prices = [4.0, 10.0, 7.5]
2 sale_prices = prices
3 for i in range(len(sale_prices)):
4     sale_prices[i] = sale_prices[i] * 0.8
5 print("Regular:", prices)
6 print("Sale:", sale_prices)

```

```

Regular: [3.2, 8.0, 6.0]
Sale: [3.2, 8.0, 6.0]

```

sale_prices = prices makes an alias: one list with two names. Changing sale_prices[i] changes that one list, so the regular prices are gone. The fix copies the list first:

sale_fixed.py

```

1 prices = [4.0, 10.0, 7.5]
2 sale_prices = prices.copy()
3 for i in range(len(sale_prices)):
4     sale_prices[i] = sale_prices[i] * 0.8
5 print("Regular:", prices)
6 print("Sale:", sale_prices)

```

```

Regular: [4.0, 10.0, 7.5]
Sale: [3.2, 8.0, 6.0]

```

Only line 2 changed. Now sale_prices is a separate list, so the loop's item assignments change only the copy. prices[:] or list(prices) would work just as well.

Reread 6.9 *Aliasing Versus Copying* if your result differed.

Question 17

```

['Ana', 'Cy']
Eve 5

```

Line	Action	line after
2	extend two names	['Ana', 'Ben', 'Cy', 'Dot', 'Eve']
3	pop(0) gives 'Ana'	['Ben', 'Cy', 'Dot', 'Eve']
4	insert(1, 'Ana')	['Ben', 'Ana', 'Cy', 'Dot', 'Eve']

line[1:3] is indexes 1 and 2: ['Ana', 'Cy']. The last item is 'Eve', and popping one and inserting one leaves 5 items.

Reread 6.5 *Removing Items: pop, remove, clear* and 6.2 *Indexing and Slicing Lists* if your output differed.

Question 18

```
TypeError: 'NoneType' object is not subscriptable
```

`scores.sort(reverse=True)` sorts `scores` in place and gives back `None`, so `ranked` is `None`. `NoneType` is the type of `None`, and **subscriptable** means "can be indexed with square brackets". `None` holds no items, so `ranked[0]` is a `TypeError`. Use `sorted`, which gives back a new list:

`top_score_fixed.py`

```
1 scores = [72, 91, 88]
2 ranked = sorted(scores, reverse=True)
3 print("Top score:", ranked[0])
```

```
Top score: 91
```

`scores.sort(reverse=True)` on its own line followed by `scores[0]` also works, but changes the order of `scores`. And `max(scores)` gives the top score with no sorting at all.

Reread the Watch Out in 6.8 *Sorting and Reversing* if you missed it.

Question 19

```
[1, 2, 3]
[1, 2, 4]
```

Line	What happens	a	b
2	b becomes an alias of the list [1, 2]	[1, 2]	[1, 2] (same list)
3	a + [3] builds a new list; a is rebound	[1, 2, 3]	[1, 2]
4	append changes the list b refers to	[1, 2, 3]	[1, 2, 4]

After line 3, `a` and `b` refer to different lists, so the `append` on line 4 reaches only `b`'s list. If line 3 had been `a.append(3)`, both names would still share one list, and both would print `[1, 2, 3, 4]`.

Reread 6.9 *Aliasing Versus Copying* if your output differed.

Question 20

B. It builds a new list, appending each score that passes, and never changes the list it is looping over.

- **A** is wrong: removing items from the list the for loop is going through shifts the later items left, so the item after each removed one is skipped, and a failing score can survive (6.6). It also changes `scores`, and `passing` is only an alias of it.
- **C** is wrong: `score < 60` is a `bool`, so `remove` looks for an item equal to `True` or `False`, not for low scores. And `remove` gives back `None`, so `passing` would be `None`.
- **D** is wrong: `passing = [score]` replaces the whole list with a one-item list on every passing score, so only the last passing score is left. Adding to a list is `append`.

Reread 6.6 *Looping Over a List* if you chose differently.

Question 21

```
[[99, 2], [3, 4]]  
3 2
```

`grid.copy()` builds a new **outer** list, so `backup is grid` is `False`. But a copy holds the same items as the original (6.9), and here the items are the two row lists. Both outer lists refer to the same rows:

Change	Reaches backup?	Why
<code>grid[0][0] = 99</code>	Yes	It changes row 0, a list that both outer lists share
<code>grid.append([5, 6])</code>	No	It changes <code>grid</code> 's outer list, and <code>backup</code> has its own

So `backup` shows the 99, but still has only 2 rows, while `grid` has 3. To copy a grid completely, copy each row too: start with an empty list and `append(row.copy())` for every row.

Reread 6.9 *Aliasing Versus Copying* and 6.10 *Nested Lists as a Grid* if your output differed.

Question 22

Model answer: `input()` always gives back a string (Day 3), so the list holds the strings `'9'`, `'10'`, and `'100'`, not numbers. Strings are sorted as text, comparing character by character from the left. `'1'` comes before `'9'`, so both strings that start with 1 come first, and `'10'` comes before `'100'` because it is shorter and otherwise the same. Convert each age before appending it, with `ages.append(int(input("Age: ")))`; then `sorted(ages)` gives `[9, 10, 100]`.

A correct answer says that the items are strings, that strings sort character by character, and that converting with `int()` before appending fixes it.

Reread 6.8 *Sorting and Reversing* if your reasoning differed.