

Day 10 Functions, part 2

Self-Study Guide · About 75 minutes · With answer key

By the end you will be able to

- Give parameters default values and call a function with keyword arguments, and avoid the mutable default trap.
- Write functions that accept any number of arguments with `*args` and `**kwargs`.
- Pass a function to another function, write short functions with `lambda`, and use `key` with `sorted()`, `min()`, and `max()`.
- Read any function signature in the Python documentation, including the `/` and `*` markers.

What to remember from earlier

- `def` makes a function; parameters are the names in its definition, arguments are the values in a call, and `return` sends a value back. ([Day 9: Functions, part 1](#))
- A function with no return statement gives back `None`. ([Day 9: Functions, part 1, 9.5 Returning Versus Printing](#))
- `b = a` makes a second name for the same list, so a change through one name shows through the other. ([Day 6: Lists, 6.9 Aliasing Versus Copying](#))
- `sorted()` returns a new list and leaves the original alone, while `sort()` changes the list in place. ([Day 6: Lists, 6.8 Sorting and Reversing](#))
- A tuple is a sequence that cannot change, `(72,)` is a tuple with one item, and `first, *rest = items` collects the leftovers into a list. ([Day 7: Tuples, sets and unpacking](#))
- `items()` gives each key and value of a dictionary together, and `get(key, default)` gives the default instead of a `KeyError`. ([Day 8: Dictionaries](#))
- `is None` checks whether a name refers to `None`, the value that means "no value yet". ([Day 4: Making decisions, 4.8 Equality Versus Identity](#))

Make a `day10` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days.

On Day 9 you wrote your own functions: `def`, parameters, `return`. Every function you wrote needed exactly as many arguments as it had parameters, given in exactly the right order. The built-in functions you have used since Day 1 are more flexible than that. `round(7.6)` and `round(7.6, 1)` both work. `enumerate(word)` counts from 0 and `enumerate(word, 1)` counts from 1. `max` takes two numbers, five numbers, or a whole list. Today you learn how to give your own functions that flexibility, and then something new: handing a function to another function, so you can tell `sorted()`, `min()`, and `max()` **how** to compare things. By the end of the day you will be able to read the first line of any function in the Python documentation and know how to call it.

10.1 Default Parameter Values

Day 9's functions always needed every argument. Suppose you write a greeting function and nearly every call says "Hello". Typing `"Hello"` in every call is repetitive, and leaving it out is an

error. What you want is an argument the caller can **leave out**, with a sensible value used in its place. `round` already works like this: with no second argument it rounds to a whole number.

Recall from Day 9 that a **parameter** is a name in the function's definition and an **argument** is the value you hand over in a call. A parameter can be given a **default value**: a value written in the `def` line with `=`, which Python uses whenever the call leaves that argument out.

```
1 def name(required, optional=default_value):
```

- `required` has no default, so every call must supply it. It is a **required parameter**.
- `optional=default_value` has a default, so a call may leave it out. It is an **optional parameter**.
- There are no spaces around the `=` in a parameter list. That is the PEP 8 style for defaults (and for keyword arguments in 10.2), and it helps you tell a default apart from an ordinary assignment.

WORKED EXAMPLE A greeting with a default

`greet_default.py`

```
1 def greet(name, greeting="Hello"):
2     """Return a greeting for name."""
3     return f"{greeting}, {name}!"
4
5
6 print(greet("Ana"))
7 print(greet("Ben", "Welcome"))
```

```
Hello, Ana!
Welcome, Ben!
```

- Line 6 passes one argument. Python matches `"Ana"` to `name`. Nothing is left for `greeting`, so Python uses its default, `"Hello"`.
- Line 7 passes two. `"Ben"` goes to `name` and `"Welcome"` goes to `greeting`, replacing the default for this call only. The next call that leaves `greeting` out gets `"Hello"` again.

Now change what the default is: a number rather than a string, used in a calculation.

`power_default.py`

```
1 def power(base, exponent=2):
2     """Return base raised to exponent. With no exponent, square it."""
3     return base ** exponent
4
5
6 print(power(5))
7 print(power(2, 10))
8 print(power(10, 0))
```

```
25
1024
1
```

`power(5)` uses the default exponent 2, giving 5 squared. `power(2, 10)` gives 2 to the power 10. `power(10, 0)` is worth a look: the caller passed 0, a value that is falsy (Day 4), but Python still uses

it. A default is used only when the argument is **missing**, never because the value passed looks empty.

KEY IDEA

`def f(a, b=2):` makes `b` optional: a call that leaves it out gets 2, and a call that passes it replaces 2 for that call only. Parameters without defaults are required.

WATCH OUT

A default before a required parameter. It seems natural to put the greeting first, since it comes first in the sentence:

default_first.py

```
1 def greet(greeting="Hello", name):
2     """Return a greeting for name."""
3     return f"{greeting}, {name}!"
4
5
6 print(greet("Ana"))
```

```
File ".../day10/default_first.py", line 1
    def greet(greeting="Hello", name):
                                ^^^^^
```

SyntaxError: parameter without a default follows parameter with a default

Python matches arguments to parameters from left to right. If `greet("Ana")` were allowed, Python could not tell whether "Ana" was meant for `greeting` (leaving `name` missing) or for `name` (skipping `greeting`). To make every call unambiguous, the rule is: **required parameters come first, then the ones with defaults**. Fix it by swapping them back:

`def greet(name, greeting="Hello"):`. Older Python versions word this message differently, but it is still a `SyntaxError` on the `def` line.

PAUSE AND TRY

Question 1 Tracing

What does this program print? For each call, say which values `seats` and `price` get and where they came from.

ticket.py

```
1 def ticket(event, seats=1, price=12.5):
2     """Return a line describing a ticket order."""
3     total = seats * price
4     return f"{event}: {seats} x {price} = {total}"
5
6
7 print(ticket("Concert"))
8 print(ticket("Play", 3))
9 print(ticket("Film", 2, 8))
```

10.2 Keyword Arguments

So far every argument has been matched to a parameter by its **position**: first argument to first parameter, second to second. That works for one or two parameters. With three similar ones it becomes easy to put them in the wrong order, and Python cannot notice, because a string is a string.

WORKED EXAMPLE Mixing up the order

badge.py

```
1 def make_badge(name, role, team):
2     """Return the text for a name badge."""
3     return f"{name} | {role} | {team}"
4
5
6 print(make_badge("Ana", "Captain", "Blue"))
7 print(make_badge("Ana", "Blue", "Captain"))
8 print(make_badge(team="Blue", name="Ana", role="Captain"))
```

```
Ana | Captain | Blue
Ana | Blue | Captain
Ana | Captain | Blue
```

- Line 6 gives the arguments in the order of the parameters, so the badge is right.
- Line 7 swaps the last two. Python happily puts "Blue" in `role` and "Captain" in `team`. There is no error, just a wrong badge.
- Line 8 names each parameter in the call: `team="Blue"` means "the parameter called `team` gets "Blue". The arguments are in a completely different order, and the badge is still right.

An argument matched by its place in the call is a **positional argument**. Day 6 met one other kind, `reverse=True` in `sorted`: a **keyword argument**, written as `parameter_name=value` in a call. A keyword argument is matched by **name**, so its position does not matter. Keyword arguments also make a call easier to read, because each value says what it is for.

Keyword arguments really pay off with defaults. In this coffee order, both `size` and `milk` are optional. With positional arguments alone, there would be no way to choose the milk without also typing the size, because `milk` comes second. A keyword argument can **skip over** a default:

coffee_order.py

```
1 def order(drink, size="medium", milk="whole"):
2     """Return a description of a coffee order."""
3     return f"A {size} {drink} with {milk} milk"
4
5
6 print(order("latte"))
7 print(order("latte", "large"))
8 print(order("latte", milk="oat"))
9 print(order("mocha", milk="oat", size="small"))
```

```
A medium latte with whole milk
A large latte with whole milk
A medium latte with oat milk
A small mocha with oat milk
```

- `order("latte")` uses both defaults.
- `order("latte", "large")` fills `size` by position.
- `order("latte", milk="oat")` fills `drink` by position, skips `size` (it keeps its default), and sets `milk` by name.
- The last call gives two keyword arguments in the "wrong" order, which is fine: names, not places, decide where they go.

You have been close to keyword arguments since Day 1. `print` has two optional parameters that change what it does: `sep`, the text put **between** the values (a space by default), and `end`, the text put **after** them (a newline by default). You can only set them by keyword:

`print_keywords.py`

```
1 print("2026", "10", "15", sep="-")
2 print("a", "b", "c", sep="")
3 print("Loading", end="")
4 print("...", end="")
5 print(" done")
```

```
2026-10-15
abc
Loading... done
```

Line 1 joins its three values with `-` instead of a space. Line 2 uses the empty string, so nothing goes between them. Lines 3 and 4 use `end=""`, so `print` does not move to a new line, and the next `print` carries on where it stopped. Line 5 ends normally, finishing the line.

KEY IDEA

A **keyword argument** (`name=value` in a call) is matched by parameter name, so order does not matter and it can skip over defaults. In a call, positional arguments come first, then keyword arguments.

WATCH OUT

Breaking the order rule in a call. Positional arguments must come before keyword arguments, for the same reason as in 10.1: Python fills positions from the left, so it must see them all first.

`keyword_first.py`

```
1 def order(drink, size="medium", milk="whole"):
2     """Return a description of a coffee order."""
3     return f"A {size} {drink} with {milk} milk"
4
5
6 print(order(size="large", "latte"))
```

```
SyntaxError: positional argument follows keyword argument
```

Fix: `order("latte", size="large")`. A related mistake is giving a parameter a value twice, once by position and once by name:

`keyword_twice.py`

```
1 def order(drink, size="medium", milk="whole"):
2     """Return a description of a coffee order."""
3     return f"A {size} {drink} with {milk} milk"
4
5
6 print(order("latte", drink="tea"))
```

```
TypeError: order() got multiple values for argument 'drink'
```

"latte" already went to drink by position, so `drink="tea"` tries to fill it again. Each parameter gets exactly one value. (A misspelled name, such as `milks="oat"`, gives got an unexpected keyword argument 'milks'.)

PAUSE AND TRY

Question 2 Multiple choice

What does this program print?

`describe_animal.py`

```
1 def describe(animal, sound="no sound", legs=4):
2     """Return a sentence about an animal."""
3     return f"The {animal} has {legs} legs and makes {sound}."
4
5
6 print(describe("bird", legs=2))
```

- A. The bird has 2 legs and makes no sound.
- B. The bird has 4 legs and makes 2.
- C. The bird has 4 legs and makes no sound.
- D. An error, because sound was skipped

10.3 The Mutable Default Argument Trap

Defaults look simple, but one kind of default hides a famous bug. Imagine a shopping cart function: add an item to a cart and return the cart, and if no cart is given, start an empty one. An empty list looks like the obvious default.

Before you see the bug, recall Day 6: `b = a` does not copy a list. It makes a second name for the **same** list, so a change made through one name shows through the other. Keep that in mind.

WORKED EXAMPLE A default that remembers

cart_trap.py

```

1 def add_item(item, cart=[]):
2     """Add item to cart and return the cart."""
3     cart.append(item)
4     return cart
5
6
7 first = add_item("apple")
8 print(first)
9 second = add_item("bread")
10 print(second)
11 print(first is second)

```

```

['apple']
['apple', 'bread']
True

```

The second call should have returned `['bread']`: a new cart with one item. Instead `bread` was added to the first cart, and line 11 confirms with `is` (Day 4) that `first` and `second` are the very same list.

Here is why. A default value is worked out **once**, when Python runs the `def` line, and is stored with the function. It is not made fresh on each call. So there is exactly one default list, and every call that leaves out `cart` gets that same list:

Step	The one default list	first	second
def runs	<code>[]</code>		
<code>add_item("apple")</code>	<code>['apple']</code>	that list	
<code>add_item("bread")</code>	<code>['apple', 'bread']</code>	that list	that list

The `cart.append(item)` on line 3 changes that list in place, so the change stays for the next call. This is Day 6's aliasing, hidden inside a function.

The fix is to use a default that cannot be changed, `None` ("no cart given"), and make the new list **inside** the function, where the code runs on every call:

cart_fixed.py

```
1 def add_item(item, cart=None):
2     """Add item to cart and return the cart. With no cart, start a new one."""
3     if cart is None:
4         cart = []
5     cart.append(item)
6     return cart
7
8
9 first = add_item("apple")
10 second = add_item("bread")
11 print(first)
12 print(second)
13 print(first is second)
14 groceries = ["eggs"]
15 add_item("milk", groceries)
16 print(groceries)
```

```
['apple']
['bread']
False
['eggs', 'milk']
```

Lines 3 and 4 run on every call. When `cart` is `None`, line 4 makes a brand-new empty list for this call only, so `first` and `second` are now different lists. The last three lines show that passing your own list still works: `"milk"` is added to `groceries` itself.

Why are `"Hello"`, `2`, and `12.5` safe defaults when a list is not? Strings and numbers are **immutable**: nothing can change them in place, so sharing one is harmless. Lists, dictionaries, and sets are **mutable**: they can change in place, so a shared one carries changes from call to call. A **mutable default argument** is a default value that is a list, dictionary, or set.

KEY IDEA

A default is made once, when `def` runs, and shared by every call. Never use a list, dictionary, or set as a default. Use `None` and create the new value inside the function:

```
if cart is None: cart = []
```

WATCH OUT

Believing the bug is in the caller. When `second` shows `['apple', 'bread']`, it is tempting to look for code that added `apple` to `second`. There is none. The extra item comes from the shared default. Whenever a function "remembers" things between calls that it should not, look at its `def` line for `=[]`, `={}`, or `=set()`.

PAUSE AND TRY**Question 3** Tracing

The same trap, with a dictionary. What does this program print? Track the one shared default dictionary.

tally_trap.py

```
1 def tally(word, counts={}):
2     """Count one more use of word and return the counts."""
3     counts[word] = counts.get(word, 0) + 1
4     return counts
5
6
7 print(tally("yes"))
8 print(tally("no"))
9 print(tally("yes", {}))
10 print(tally("yes"))
```

10.4 Any Number of Arguments: *args

Day 9 asked you to write a function that takes a list of numbers. That works, but the caller has to build a list first. `max(3, 9, 4)` does not need a list: it takes as many numbers as you give it. How can your own function do that, when you cannot know how many parameters to write?

Recall Day 7's starred unpacking: in `first, *rest = items`, the star means "collect everything left over". A star in front of a parameter means the same thing for a call. A parameter written as `*args` collects **all the leftover positional arguments** into a tuple. The name after the star can be anything (`*scores`, `*prices`); `args`, short for arguments, is the name programmers use when there is no better one.

WORKED EXAMPLE An average of any length

average_args.py

```
1 def average(*scores):
2     """Return the average of any number of scores."""
3     print("scores is", scores)
4     return sum(scores) / len(scores)
5
6
7 print(average(85, 90, 78))
8 print(average(100, 60))
9 print(average(72))
```

```
scores is (85, 90, 78)
84.33333333333333
scores is (100, 60)
80.0
scores is (72,)
72.0
```

Line 3 prints `scores` so you can see what Python built:

- Three arguments arrive as the tuple `(85, 90, 78)`.
- Two arguments arrive as `(100, 60)`.
- One argument still arrives as a tuple, `(72,)`. The trailing comma is Day 7's single-item tuple.

Because `scores` is always a tuple, line 4 can use `sum` and `len` on it whatever the number of arguments.

A `*args` parameter can come after ordinary parameters. The ordinary ones are filled first, from the left, and the star collects whatever is left, which may be nothing at all:

`report_args.py`

```
1 def report(title, *scores):
2     """Print a title and a count, then each score on its own line."""
3     print(f"{title}: {len(scores)} scores")
4     for score in scores:
5         print("-", score)
6
7
8 report("Quiz 1", 8, 10, 7)
9 report("Quiz 2")
```

```
Quiz 1: 3 scores
- 8
- 10
- 7
Quiz 2: 0 scores
```

"Quiz 1" fills `title`, and the three numbers go into `scores`. `report("Quiz 2")` fills `title` and leaves nothing over, so `scores` is the empty tuple `()`: `len` gives 0 and the for loop runs zero times. An empty `*args` tuple can catch you out, though. Call `average` (without its `print` line) with no arguments, and it divides by `len()`, which is 0:

`average_empty.py`

```
1 def average(*scores):
2     """Return the average of any number of scores."""
3     return sum(scores) / len(scores)
4
5
6 print(average())
```

```
ZeroDivisionError: division by zero
```

This is Day 6's empty-list average again. A real version should check the length first, as `biggest_gap` does in the question below.

The star works the other way round in a **call**. If your numbers are already in a list, `*` in front of the list spreads its items out as separate positional arguments. This is called **unpacking** the list into the call:

star_call.py

```
1 def average(*scores):
2     """Return the average of any number of scores."""
3     return sum(scores) / len(scores)
4
5
6 marks = [70, 80, 90]
7 print(average(*marks))
8 print(*marks)
9 print(marks)
```

```
80.0
70 80 90
[70, 80, 90]
```

`average(*marks)` is the same as `average(70, 80, 90)`. `print(*marks)` is the same as `print(70, 80, 90)`, which prints the three values separated by spaces, while `print(marks)` prints one value, the list itself.

KEY IDEA

A `*args` parameter collects any number of extra positional arguments into a tuple, which may be empty. In a call, `*items` does the reverse: it spreads a list out into separate arguments.

WATCH OUT

Passing a list where separate values are expected. The list goes into the tuple as **one** item:

average_list.py

```
1 def average(*scores):
2     """Return the average of any number of scores."""
3     return sum(scores) / len(scores)
4
5
6 marks = [70, 80, 90]
7 print(average(marks))
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Inside `average`, `scores` is `([70, 80, 90],)`: a tuple holding one list. `sum` starts at 0 and tries `0 + [70, 80, 90]`, and Python cannot add a number and a list. Either pass the numbers separately, `average(70, 80, 90)`, or unpack the list, `average(*marks)`.

PAUSE AND TRY**Question 4** Tracing

What does this program print? The last call is the tricky one.

biggest_gap.py

```

1 def biggest_gap(label, *values):
2     """Return the label and the gap between the largest and smallest value."""
3     if len(values) < 2:
4         return f"{label}: need two values"
5     return f"{label}: {max(values) - min(values)}"
6
7
8 print(biggest_gap("Temps", 12, 19, 7, 15))
9 print(biggest_gap("Single", 4))
10 print(biggest_gap(3, 9))

```

10.5 Keyword Arguments by the Handful: **kwargs

`*args` collects extra **positional** arguments. Sometimes you want extra **keyword** arguments instead: a profile function where one person has an age and a city, and another has a club and a favorite color, and you cannot list every possible detail as a parameter.

A parameter written as `**kwargs` (two stars) collects **all the leftover keyword arguments** into a dictionary (Day 8). Each keyword becomes a key, as a string, and the value after `=` becomes its value. `kwargs` is short for keyword arguments; again, any name works.

WORKED EXAMPLE A profile with any details

profile_kwargs.py

```

1 def make_profile(name, **details):
2     """Print a profile: the name, then each detail given by keyword."""
3     print("details is", details)
4     print(f"Profile: {name}")
5     for field, value in details.items():
6         print(f"    {field}: {value}")
7
8
9 make_profile("Ana", age=15, city="Lima", club="Chess")
10 make_profile("Ben")

```

```

details is {'age': 15, 'city': 'Lima', 'club': 'Chess'}
Profile: Ana
    age: 15
    city: Lima
    club: Chess
details is {}
Profile: Ben

```

- "Ana" fills name by position.
- `age=15`, `city="Lima"`, and `club="Chess"` match no parameter by name, so all three go into the dictionary `details`, in the order they were typed. The keys are the strings `'age'`, `'city'`, and `'club'`.
- Lines 5 and 6 loop over the dictionary with `items()` from Day 8.

- `make_profile("Ben")` passes no keyword arguments, so `details` is the empty dictionary and the loop runs zero times.

Two stars also work in a **call**: `**` in front of a dictionary unpacks it into keyword arguments, one per key. This is handy when the settings are stored in a dictionary:

`kwargs_call.py`

```
1 def order(drink, size="medium", milk="whole"):
2     """Return a description of a coffee order."""
3     return f"A {size} {drink} with {milk} milk"
4
5
6 usual = {"size": "large", "milk": "oat"}
7 print(order("latte", **usual))
8 print(order("mocha", **usual))
```

```
A large latte with oat milk
A large mocha with oat milk
```

`order("latte", **usual)` is the same as `order("latte", size="large", milk="oat")`. Each key must be the name of a parameter.

Keyword-only parameters

A `*args` parameter swallows **every** leftover positional argument. So what about a parameter written after it? It can never be filled by position: the star got there first. It can only be set by keyword. A parameter that can only be given by keyword is a **keyword-only parameter**. That sounds like a limitation, but it is often exactly what you want:

`keyword_only.py`

```
1 def total_price(*prices, discount=0):
2     """Add up any number of prices, then take off a discount (0.1 is 10%)."""
3     subtotal = sum(prices)
4     return subtotal - subtotal * discount
5
6
7 print(total_price(20, 30, 50))
8 print(total_price(20, 30, 50, discount=0.1))
9 print(total_price(20, 30, 50, 0.1))
```

```
100
90.0
100.1
```

- The first call has no discount, so `discount` is its default 0, and the total is 100.
- The second sets `discount=0.1` by keyword: 10% off 100 is 90.0.
- The third shows why the rule is useful. The caller meant 0.1 as the discount, but it was positional, so it went into `prices` as a fourth price: 100.1. Because `discount` is keyword-only, a reader of the call `total_price(20, 30, 50, discount=0.1)` can see at once which number is the discount.

All the kinds of parameter can appear in one `def`, but only in this order: ordinary parameters, then ones with defaults, then `*args`, then keyword-only ones, then `**kwargs` last. This program uses all five, with short names because only the mechanics matter here:

parameter_order.py

```
1 def show(a, b=2, *args, c=3, **kwargs):
2     print(a, b, args, c, kwargs)
3
4
5 show(1)
6 show(1, 5, 6, 7, c=8, d=9)
```

```
1 2 () 3 {}
1 5 (6, 7) 8 {'d': 9}
```

Parameter	show(1)	show(1, 5, 6, 7, c=8, d=9)
a	1	1
b=2	2 (default)	5
*args	()	(6, 7)
c=3	3 (default)	8 (by keyword only)
**kwargs	{}	{'d': 9}

Python fills positions from the left (a, then b), the star takes the rest (6 and 7), keywords that match a parameter go there (c=8), and keywords that match nothing go into the dictionary (d=9).

KEY IDEA

`**kwargs` collects extra keyword arguments into a dictionary of name to value, and `**settings` in a call spreads a dictionary out as keyword arguments. Parameters after `*args` are keyword-only.

WATCH OUT

Passing a positional argument where only keywords are collected. Two stars collect keyword arguments only:

kwargs_positional.py

```
1 def make_profile(name, **details):
2     """Print a profile: the name, then each detail given by keyword."""
3     print(f"Profile: {name}")
4     for field, value in details.items():
5         print(f" {field}: {value}")
6
7
8 make_profile("Ana", "Lima")
```

```
TypeError: make_profile() takes 1 positional argument but 2 were given
```

"Lima" has no name, so it cannot be a dictionary entry, and there is no `*args` to catch it. Give it a name: `make_profile("Ana", city="Lima")`.

PAUSE AND TRY

Question 5 Tracing

What does this program print? Decide where each argument goes before you look.

pizza.py

```
1 def describe_pizza(size, *toppings, crust="thin", **extras):
2     print(f"{size} pizza, {crust} crust")
3     print("Toppings:", toppings)
4     print("Extras:", extras)
5
6
7 describe_pizza("Large", "olives", "peppers", sauce="spicy", crust="thick")
```

10.6 Functions Are Values

Here is a new kind of problem. You have an `announce` function that prints a message. Sometimes the message should be shouted, sometimes whispered, and next week someone will want a third style. You could add a `style` parameter holding a string and a long if chain inside `announce`, and edit that chain every time a style is added. It would be better if the caller could hand over **the styling itself**: a piece of behavior, not a piece of data.

Python allows this because a function is a value, like a number or a list. `def shout(...)` does two things: it builds a **function object**, and it stores it in the name `shout`, just as `count = 0` stores 0 in `count`. The name without parentheses refers to the function itself. Adding parentheses **calls** it.

```
>>> def shout(text):
...     return text.upper() + "!"
...
>>> type(shout)
<class 'function'>
>>> yell = shout
>>> yell("fire drill")
'FIRE DRILL!'
>>> shout("fire drill")
'FIRE DRILL!'
>>> len
<built-in function len>
>>> measure = len
>>> measure("hello")
5
```

- `type(shout)` shows the function is a value of type `function`.

- `yell = shout` has no parentheses, so nothing is called. It makes `yell` a second name for the same function, exactly like `b = a` with a list. `yell("fire drill")` and `shout("fire drill")` call the same code.
- Built-in functions are values too. `len` without parentheses is the function itself, and `measure = len` gives it a second name.

Because a function is a value, you can pass it as an argument. The parameter then holds a function, and the code inside can call it with parentheses:

WORKED EXAMPLE Passing the style in

announce.py

```

1 def shout(text):
2     return text.upper() + "!"
3
4
5 def whisper(text):
6     return text.lower() + "..."
7
8
9 def announce(message, style):
10    """Print message after passing it through the style function."""
11    print(style(message))
12
13
14 announce("Class Starts Now", shout)
15 announce("Class Starts Now", whisper)

```

```

CLASS STARTS NOW!
class starts now...

```

Follow the first call, `announce("Class Starts Now", shout)`:

1. `message` gets the string "Class Starts Now", and `style` gets the function `shout`. Notice that the call writes `shout`, with no parentheses: the function is handed over, not run.
2. Line 11 works out `style(message)`. `style` refers to `shout`, so this calls `shout("Class Starts Now")`, which returns "CLASS STARTS NOW!".
3. `print` shows that string.

The second call is identical except that `style` refers to `whisper`. `announce` never changes; the behavior that was passed in does.

A function that takes another function as an argument (or returns one) is called a **higher-order function**. Here is a second one that varies something different: the passed-in function returns `True` or `False`, and the higher-order function uses it to decide what to count. This is the counter pattern from Day 5 with the test left blank for the caller to fill in:

count_if.py

```
1 def count_if(items, test):
2     """Return how many items make test(item) True."""
3     count = 0
4     for item in items:
5         if test(item):
6             count += 1
7     return count
8
9
10 def is_even(number):
11     return number % 2 == 0
12
13
14 def is_big(number):
15     return number > 50
16
17
18 numbers = [12, 75, 33, 90, 48]
19 print(count_if(numbers, is_even))
20 print(count_if(numbers, is_big))
```

```
3
2
```

Inside `count_if`, line 5 calls `test(item)` for each number. With `is_even`, that is `True` for 12, 90, and 48, so the count is 3. With `is_big`, it is `True` for 75 and 90, so the count is 2. One loop, written once, counts anything you can write a test for.

KEY IDEA

A function's name without parentheses is the function itself, a value you can store or pass. With parentheses, it is a call. To pass a function to another function, write its name **without** parentheses.

WATCH OUT

Calling the function instead of passing it. Parentheses after the name run the function right away:

announce_called.py

```
1 def shout(text):
2     return text.upper() + "!"
3
4
5 def announce(message, style):
6     """Print message after passing it through the style function."""
7     print(style(message))
8
9
10 announce("Class Starts Now", shout())
```

```
TypeError: shout() missing 1 required positional argument: 'text'
```

`shout()` tries to call `shout` immediately, with no argument, before `announce` even starts, and `shout` needs a `text`. Even with an argument, `shout("hi")` would hand `announce` the **string** `"HI!"` rather than the function, and `style(message)` would then fail with `'str' object is not callable` ("callable" means "can be called with parentheses"). Write `announce("Class Starts Now", shout)`.

PAUSE AND TRY

Question 6 Multiple choice

Using `shout` and `announce` from `announce.py`, which call prints `GO TEAM!`?

- A. `announce("Go team", shout())`
- B. `announce("Go team", shout)`
- C. `announce("Go team", shout("Go team"))`
- D. `announce(shout, "Go team")`

10.7 lambda Expressions

`count_if` is useful, but every test needs its own `def`: `is_even`, `is_big`, and `next_is_between_30_and_60`. Each is one line of real work, used once, with two blank lines around it. For tiny one-use functions, that is a lot of ceremony.

A **lambda expression** builds a small function right where you need it, without a `def` and without a name:

```
1 lambda parameters: expression
```

- `lambda` is a keyword. The parameters come next, separated by commas, then a colon.
- After the colon is **one expression**. When the function is called, Python works out that expression and returns its value automatically. There is no `return` keyword.
- The whole thing is an expression whose value is a function, so it can go anywhere a value can: as an argument, in a list, after `=`.

`lambda n: n * 2` means the same as:

```
1 def double(n):  
2     return n * 2
```

except that the `lambda` version has no name of its own.

```
>>> double = lambda n: n * 2
>>> double(21)
42
>>> type(double)
<class 'function'>
>>> (lambda price, quantity: price * quantity)(2.5, 4)
10.0
```

The first line stores a lambda in a name so you can try it: `double(21)` is 42, and `type` shows it is an ordinary function. The last line builds a two-parameter lambda and calls it at once with `(2.5, 4)`, giving 10.0. Both are for seeing how lambdas work. In real code, PEP 8 says: if you want a name, use `def`. The place where lambdas belong is as an argument:

WORKED EXAMPLE Tests written in place

`count_if_lambda.py`

```
1 def count_if(items, test):
2     """Return how many items make test(item) True."""
3     count = 0
4     for item in items:
5         if test(item):
6             count += 1
7     return count
8
9
10 numbers = [12, 75, 33, 90, 48]
11 print(count_if(numbers, lambda n: n % 2 == 0))
12 print(count_if(numbers, lambda n: n > 50))
13 print(count_if(numbers, lambda n: 30 <= n <= 60))
```

```
3
2
2
```

Each call writes its test right where it is used. `lambda n: n % 2 == 0` is `is_even` without the `def`; `lambda n: n > 50` is `is_big`. The third test is new, and it took one line rather than four: a chained comparison from Day 4, `True` for 33 and 48, so the count is 2. `count_if` itself is the same function as in 10.6, and it cannot tell whether it was given a `def` function or a lambda.

The expression after the colon can be any expression, including a conditional expression from Day 4:

`lambda_shorten.py`

```
1 def announce(message, style):
2     """Print message after passing it through the style function."""
3     print(style(message))
4
5
6 announce("Lunch is served", lambda text: text[:8] + "...")
7 announce("Lunch is served", lambda text: f"** {text} **")
8 announce("Hi all", lambda text: text if len(text) <= 8 else text[:8] + "...")
```

```
Lunch is...  
** Lunch is served **  
Hi all
```

The first lambda keeps the first 8 characters and adds dots. The second wraps the text in stars with an f-string. The third chooses: if the text is at most 8 characters, return it unchanged; otherwise shorten it. "Hi all" has 6 characters, so it comes back as it was.

KEY IDEA

`lambda x: expression` is a small, unnamed function that returns the value of one expression. Use it for short functions you pass to another function; use `def` for anything you name or reuse.

WATCH OUT

Writing `return` (or any statement) in a lambda. The part after the colon must be an expression, and `return` is a statement:

lambda_return.py

```
1 def count_if(items, test):  
2     """Return how many items make test(item) True."""  
3     count = 0  
4     for item in items:  
5         if test(item):  
6             count += 1  
7     return count  
8  
9  
10 print(count_if([12, 75, 33], lambda n: return n > 50))
```

SyntaxError: invalid syntax

Remove `return: lambda n: n > 50`. The lambda returns the expression's value without being told. If you find yourself wanting an `if` statement, a loop, or several lines inside a lambda, write a `def` function instead.

PAUSE AND TRY

Question 7 Tracing

What does this program print? Work out what each lambda does to one word first.

apply_to_each.py

```

1 def apply_to_each(words, change):
2     """Return a new list holding change(word) for each word."""
3     result = []
4     for word in words:
5         result.append(change(word))
6     return result
7
8
9 words = ["tea", "coffee", "juice"]
10 print(apply_to_each(words, lambda w: w[0].upper() + w[1:]))
11 print(apply_to_each(words, lambda w: len(w)))
12 print(apply_to_each(words, lambda w: w if len(w) > 3 else w * 2))

```

10.8 Sorting with key

Day 6 sorted lists with `sorted()`, which returns a new sorted list, and `sort()`, which sorts a list in place. Both put items in their **natural** order: numbers from smallest to largest, and strings alphabetically, with capitals before small letters (Day 4). But often you want a different order: words by length, names ignoring capitals, students by score. You need a way to tell `sorted` **what to compare**.

Both have an optional keyword argument `key`, which takes a function. Python calls that function **once on each item** and sorts the items by the values it returns. Those returned values are called the **sort keys**. The items themselves are not changed: only their order is.

WORKED EXAMPLE Four ways to sort the same words

```

>>> words = ["pear", "Fig", "banana", "kiwi", "apple"]
>>> sorted(words)
['Fig', 'apple', 'banana', 'kiwi', 'pear']
>>> sorted(words, key=len)
['Fig', 'pear', 'kiwi', 'apple', 'banana']
>>> sorted(words, key=str.lower)
['apple', 'banana', 'Fig', 'kiwi', 'pear']
>>> sorted(words, key=len, reverse=True)
['banana', 'apple', 'pear', 'kiwi', 'Fig']
>>> words
['pear', 'Fig', 'banana', 'kiwi', 'apple']
>>> words.sort(key=len)
>>> words
['Fig', 'pear', 'kiwi', 'apple', 'banana']

```

`sorted(words)` uses natural order. `'Fig'` comes first because a capital `F` comes before every small letter.

`key=len` passes the built-in `len` function (no parentheses: 10.6). Python works out a key for each word and sorts by those numbers:

Item	Key: len(item)
"pear"	4
"Fig"	3
"banana"	6
"kiwi"	4
"apple"	5

Sorting by key gives 3, 4, 4, 5, 6, so the result is Fig, pear, kiwi, apple, banana.

`key=str.lower` sorts by each word's lowercase version, so capitals no longer jump ahead. `str.lower` is the `lower` method from Day 3, named through its type, `str`, without parentheses, so it is the function itself; Python calls it on each word. The words in the result keep their original capitals: 'Fig', not 'fig'.

`reverse=True`, the keyword argument from Day 6, still sorts from largest to smallest, now by key. Asking for `words` after the four calls shows it unchanged, because `sorted` makes a new list each time. The last two lines use the list method `sort`, which takes the same `key` and `reverse` keyword arguments but sorts the list in place and returns nothing (the REPL shows no result line).

Look again at `pear` and `kiwi`. Both have length 4, a tie. In the result, `pear` is before `kiwi`, just as it was in the original list. That is a guarantee, not luck: Python's sort is **stable**, meaning items with equal keys keep their original order. Even with `reverse=True`, the tied `pear` and `kiwi` stay in their original order.

The `key` function can be a lambda, which is how you sort a list of dictionaries (Day 8) by one field:

`sort_students.py`

```
1 students = [
2     {"name": "Maya", "grade": 10, "score": 88},
3     {"name": "Leo", "grade": 9, "score": 95},
4     {"name": "Ava", "grade": 10, "score": 95},
5     {"name": "Sam", "grade": 9, "score": 72},
6 ]
7
8 by_score = sorted(students, key=lambda student: student["score"], reverse=True)
9 for student in by_score:
10     print(student["name"], student["score"])
```

```
Leo 95
Ava 95
Maya 88
Sam 72
```

For each student dictionary, the lambda returns `student["score"]`, so the list is sorted by score, highest first because of `reverse=True`. Leo and Ava tie on 95, and Leo was earlier in the original list, so stability keeps him first.

Sorting by one field, then another

What if ties should be broken by a second field, say by grade and then, within each grade, by name? A key can be a **tuple** (Day 7), and Python compares tuples item by item: first items first, and only if those are equal, the second items.

```
>>> (9, "Sam") < (10, "Ava")
True
>>> (10, "Ava") < (10, "Maya")
True
>>> (9, "Leo") < (9, "Sam")
True
```

(9, "Sam") is less than (10, "Ava") because 9 is less than 10; the names are never looked at. (10, "Ava") and (10, "Maya") tie on 10, so the names decide. So a key that returns (grade, name) sorts by grade, and by name within a grade:

sort_two_fields.py

```
1 students = [
2     {"name": "Maya", "grade": 10, "score": 88},
3     {"name": "Leo", "grade": 9, "score": 95},
4     {"name": "Ava", "grade": 10, "score": 95},
5     {"name": "Sam", "grade": 9, "score": 72},
6 ]
7
8 by_grade_then_name = sorted(
9     students, key=lambda student: (student["grade"], student["name"]))
10 )
11 for student in by_grade_then_name:
12     print(student["grade"], student["name"])
```

```
9 Leo
9 Sam
10 Ava
10 Maya
```

KEY IDEA

sorted(items, key=function) calls the function on each item and sorts by what it returns, leaving the items themselves unchanged. Add reverse=True for largest first. Ties keep their original order, and a tuple key breaks ties with a second field.

WATCH OUT

Calling the key function. The key must be a function, so it gets no parentheses:

sort_key_called.py

```
1 words = ["pear", "fig", "banana"]
2 print(sorted(words, key=len))
```

```
TypeError: len() takes exactly one argument (0 given)
```

`len()` runs at once, with nothing to measure, before `sorted` starts. Write `key=len`, and let `sorted` call it on each item. The same goes for `key=str.lower`, not `key=str.lower()`.

PAUSE AND TRY

Question 8 Tracing

What does this program print? Each item is a tuple of a name and a score.

`sort_pairs.py`

```
1 scores = [("Ben", 95), ("Ana", 82), ("Dee", 70), ("Cal", 82)]
2 print(sorted(scores))
3 print(sorted(scores, key=lambda pair: pair[1]))
4 print(sorted(scores, key=lambda pair: pair[1], reverse=True))
```

10.9 `min()` and `max()` with key

Day 6 used `min` and `max` to find the smallest and largest number in a list. Like `sorted`, they compare items in their natural order, and like `sorted`, they accept a `key` function to compare something else. With a `key`, `max` returns the **item** whose key is largest, not the key itself.

```
>>> words = ["pear", "banana", "fig", "cherry"]
>>> max(words)
'pear'
>>> max(words, key=len)
'banana'
>>> min(words, key=len)
'fig'
>>> len(max(words, key=len))
6
```

- `max(words)` compares alphabetically: `'pear'` comes last.
- `max(words, key=len)` compares lengths. `banana` and `cherry` both have 6 letters. When there is a tie, `max` and `min` return the **first** one, so the result is `'banana'`.
- `min(words, key=len)` gives the shortest word, `'fig'`.
- `max` returns the word, not its length. If you want the length, measure the word it returns: `len(max(words, key=len))` is 6.

WORKED EXAMPLE The cheapest product

cheapest.py

```
1 products = [  
2     {"name": "pen", "price": 1.5},  
3     {"name": "notebook", "price": 3.25},  
4     {"name": "eraser", "price": 0.75},  
5 ]  
6 cheapest = min(products, key=lambda product: product["price"])  
7 print(chapest)  
8 print(f"Cheapest: {cheapest['name']} at ${cheapest['price']:.2f}")  
9  
10 prices = {"pen": 1.5, "notebook": 3.25, "eraser": 0.75}  
11 print(max(prices))  
12 print(max(prices, key=prices.get))
```

```
{'name': 'eraser', 'price': 0.75}  
Cheapest: eraser at $0.75  
pen  
notebook
```

- `min(products, key=lambda product: product["price"])` compares the three products by price and returns the whole cheapest **dictionary**, so line 8 can show both its name and its price.
- Line 8 uses single quotes inside the f-string's braces, because the f-string itself uses double quotes.
- `prices` is a dictionary. Looping over a dictionary goes through its keys (Day 8), so `max(prices)` compares the **names**: 'pen' is last alphabetically.
- `max(prices, key=prices.get)` uses the dictionary's own `get` method as the key function, without parentheses. For each name, Python calls `prices.get(name)`, which gives its price, so the result is the name with the highest price: 'notebook'.

KEY IDEA

`max(items, key=function)` and `min(items, key=function)` return the item whose key is largest or smallest; on a tie, the first one. For a dictionary, `max(d, key=d.get)` gives the key with the largest value.

WATCH OUT

max or min on an empty list. As on Day 6, with nothing to compare there is no answer:

max_empty.py

```
1 scores = []  
2 best = max(scores)  
3 print("Best score:", best)
```

```
ValueError: max() iterable argument is empty
```

Day 6 read "iterable" in this message as "something you can loop over". It is a word you will meet often from now on: an **iterable** is anything a for loop can go through, such as a list, a string, or a dictionary. If the list might be empty, give the keyword argument `default`, the value to return in that case:

max_empty_fixed.py

```
1 scores = []
2 best = max(scores, default=None)
3 print("Best score:", best)
```

Best score: None

PAUSE AND TRY

Question 9 Tracing

What does this program print?

longest_word.py

```
1 words = ["kiwi", "plum", "fig", "lime", "pear"]
2 print(max(words, key=len))
3 print(min(words, key=len))
4 print(max(words, key=lambda word: word[-1]))
```

10.10 Reading a Function Signature

You now know every kind of parameter, which means you can read the documentation for any function. The first line of a function's definition, its name and parameter list, is its **signature**. It tells you everything about how to call it: which arguments are required, which are optional and what their defaults are, and which must be given by position or by keyword. The Python documentation shows a signature at the top of each function's entry. Here are some you have used:

```
round(number, ndigits=None)
enumerate(iterable, start=0)
print(*objects, sep=' ', end='\n', file=None, flush=False)
sorted(iterable, /, *, key=None, reverse=False)
max(iterable, /, *, default, key=None)
```

Most of this you can already read. `round` has a required `number` and an optional `ndigits`. `enumerate` has an optional `start` that defaults to 0, which is how `enumerate(word, 1)` worked on Day 5. `print` has `*objects`, so it takes any number of values, followed by keyword-only parameters with defaults: `sep` is a space and `end` is a newline, exactly as you saw in 10.2. Two markers are new:

- **A bare *** with no name after it means: every parameter after this point is keyword-only. It works like `*args` without collecting anything. In `sorted`, `key` and `reverse` come after the `*`, so they must be given by keyword.

- **A /** means: every parameter **before** this point is **positional-only**, so it can never be given by keyword. In `sorted`, `iterable` is before the `/`, so you write `sorted(words)`, never `sorted(iterable=words)`.

You can use both markers in your own functions:

WORKED EXAMPLE Splitting a bill

`split_bill.py`

```
1 def split_bill(total, /, people=2, *, tip=0.15):
2     """Return each person's share of total, with a tip added."""
3     return round(total * (1 + tip) / people, 2)
4
5
6 print(split_bill(60))
7 print(split_bill(60, 3))
8 print(split_bill(60, people=4, tip=0.2))
```

```
34.5
23.0
18.0
```

Read the signature `split_bill(total, /, people=2, *, tip=0.15)` from left to right:

Parameter	Kind	Default
<code>total</code>	Required, positional-only (before <code>/</code>)	<code>none</code>
<code>people</code>	Optional, by position or by keyword (between <code>/</code> and <code>*</code>)	<code>2</code>
<code>tip</code>	Optional, keyword-only (after <code>*</code>)	<code>0.15</code>

- `split_bill(60)` uses both defaults: 60 plus a 15% tip is 69, split 2 ways is 34.5.
- `split_bill(60, 3)` sets `people` by position: 69 split 3 ways is 23.0.
- `split_bill(60, people=4, tip=0.2)` sets both by keyword: 60 plus 20% is 72, split 4 ways is 18.0.

The `round(..., 2)` hides the tiny float errors from Day 2 that a tip like 0.15 can cause.

The markers are enforced. Break them, and Python says which rule you broke:

`positional_only.py`

```
1 def split_bill(total, /, people=2, *, tip=0.15):
2     """Return each person's share of total, with a tip added."""
3     return round(total * (1 + tip) / people, 2)
4
5
6 print(split_bill(total=60))
```

```
TypeError: split_bill() got some positional-only arguments passed as keyword arguments:
'total'
```

keyword_only_error.py

```

1 def split_bill(total, /, people=2, *, tip=0.15):
2     """Return each person's share of total, with a tip added."""
3     return round(total * (1 + tip) / people, 2)
4
5
6 print(split_bill(60, 3, 0.2))

```

TypeError: split_bill() takes from 1 to 2 positional arguments but 3 were given

The second message says "1 to 2" because only `total` and `people` can be filled by position. The third value, meant as the tip, had nowhere to go. The same rules apply to the built-in functions:

```

>>> sorted([3, 1, 2], reverse=True)
[3, 2, 1]
>>> sorted([3, 1, 2], None, True)
Traceback (most recent call last):
...
TypeError: sorted expected 1 argument, got 3
>>> "a,b,c".split(",", maxsplit=1)
['a', 'b,c']
>>> round(3.14159, ndigits=2)
3.14
>>> len(obj="abc")
Traceback (most recent call last):
...
TypeError: len() takes no keyword arguments

```

- `sorted([3, 1, 2], reverse=True)` follows the signature: the list by position, reverse by keyword.
- `sorted([3, 1, 2], None, True)` tries to give `key` and `reverse` by position. They are after the `*`, so `sorted` accepts only 1 positional argument.
- `str.split(sep=None, maxsplit=-1)` has no markers, so `maxsplit=1` can be given by keyword. It stops after one split.
- `round(number, ndigits=None)` also has no markers, so `ndigits=2` works by keyword.
- `len` takes its one argument by position only, so `len(obj="abc")` is an error.

Why do these markers exist? Positional-only parameters let the function's author rename them later without breaking anyone's code, since no call can use the name. Keyword-only parameters protect calls like `total_price` from 10.5, where a value in the wrong place would be silently misused, and they make calls such as `sorted(scores, reverse=True)` say what they mean.

Older entries in the documentation sometimes use square brackets for optional arguments, as in `range(start, stop[, step])`: the part in brackets may be left out. The brackets are notation for the reader, not Python you type.

KEY IDEA

A signature lists a function's parameters in order. Names with `=` have defaults; `*name` collects extra positional arguments and `**name` extra keyword arguments; parameters before `/` are positional-only, and parameters after `*` are keyword-only.

WATCH OUT

Typing the markers or defaults into a call. The signature describes the call; it is not the call. `sorted(words, /, *, key=len)` is a syntax error. Read the signature, then write only the arguments you need: `sorted(words, key=len)`. Leave out every optional parameter whose default you are happy with.

PAUSE AND TRY**Question 10** Multiple choice

A library's documentation shows this signature:

```
resize(image, /, width, height=None, *, keep_ratio=True)
```

Which call is an **error**?

- A. `resize("cat.png", 300)`
- B. `resize("cat.png", width=300, keep_ratio=False)`
- C. `resize("cat.png", 300, 200)`
- D. `resize("cat.png", 300, 200, False)`

Putting it together

A game club keeps a leaderboard. Players are dictionaries with a name, points, and wins. The program adds players with any stats they have, ranks them by any field, and prints the top few.

leaderboard.py

```
1 def add_player(players, name, **stats):
2     """Add a player to players. Points and wins start at 0 unless given."""
3     player = {"name": name, "points": 0, "wins": 0}
4     for field, value in stats.items():
5         player[field] = value
6     players.append(player)
7
8
9 def rank(players, *, by="points", best_first=True):
10    """Return a new list of players ordered by one field."""
11    return sorted(players, key=lambda player: player[by], reverse=best_first)
12
13
14 def show(players, title="Leaderboard", limit=None):
15    """Print a numbered list of players, at most limit of them if given."""
```

```

16     print(title)
17     if limit is not None:
18         players = players[:limit]
19     for place, player in enumerate(players, 1):
20         points = player["points"]
21         wins = player["wins"]
22         print(f"{place}. {player['name']}: {points} points, {wins} wins")
23
24
25 players = []
26 add_player(players, "Ana", points=42, wins=3)
27 add_player(players, "Ben", points=55, wins=2)
28 add_player(players, "Cal", wins=4, points=42)
29 add_player(players, "Dee")
30
31 show(rank(players))
32 show(rank(players, by="wins"), title="Most wins", limit=2)
33 show(rank(players, by="name", best_first=False), "A to Z")
34 top = max(players, key=lambda player: player["points"])
35 print("Top scorer:", top["name"])

```

Leaderboard

```

1. Ben: 55 points, 2 wins
2. Ana: 42 points, 3 wins
3. Cal: 42 points, 4 wins
4. Dee: 0 points, 0 wins
Most wins
1. Cal: 42 points, 4 wins
2. Ana: 42 points, 3 wins
A to Z
1. Ana: 42 points, 3 wins
2. Ben: 55 points, 2 wins
3. Cal: 42 points, 4 wins
4. Dee: 0 points, 0 wins
Top scorer: Ben

```

Each function uses today's ideas:

Function or line	What it does	Idea
<code>add_player</code>	<code>**stats</code> collects any keyword stats; each overwrites the 0 it starts with	10.5
<code>add_player(players, "Cal", wins=4, points=42)</code>	Keyword order does not matter	10.2
<code>add_player(players, "Dee")</code>	No stats: <code>stats</code> is empty, so Dee keeps 0 points and 0 wins	10.5
<code>rank</code>	Keyword-only <code>by</code> and <code>best_first</code> with defaults; sorted with a <code>lambda</code> key and <code>reverse</code>	10.5, 10.8
<code>show</code>	Optional <code>title</code> and <code>limit</code> ; <code>None</code> means "no limit given"	10.1, 10.3

<code>max(players, key=...)</code>	Returns the whole player dictionary with the most points	10.9
------------------------------------	--	------

Follow the three `show` calls:

- `show(rank(players))` uses every default: ranked by points, best first, title `Leaderboard`, no limit. Ana and Cal tie on 42; Ana was added first, so the stable sort keeps her first.
- `show(rank(players, by="wins"), title="Most wins", limit=2)` ranks by wins and shows only two rows. `players[:limit]` (Day 6 slicing) makes a shorter list inside `show`; the caller's list is untouched.
- `show(rank(players, by="name", best_first=False), "A to Z")` sorts names alphabetically, smallest first, and passes the title by position.

Because `by` and `best_first` are keyword-only, a call reads `rank(players, by="wins")` rather than the puzzling `rank(players, "wins", True)`.

Common mistakes

Today's errors are almost all `TypeError`s about how arguments match parameters. When you see one, read the end of the message:

Message ends with	Cause	Fix
parameter without a default follows parameter with a default	A default before a required parameter in <code>def</code>	Put required parameters first
positional argument follows keyword argument	<code>f(a=1, 2)</code> in a call	Put positional arguments first
got multiple values for argument 'x'	<code>x</code> given by position and again by keyword	Give each parameter one value
got an unexpected keyword argument 'x'	No parameter named <code>x</code> , and no <code>**kwargs</code>	Check the spelling against the signature
takes 2 positional arguments but 3 were given	Too many positional arguments, or one meant for a keyword-only parameter	Use a keyword, or remove the extra value
missing 1 required positional argument: 'x'	A required argument left out, often from <code>func()</code> instead of <code>func</code>	Pass the function without parentheses, or supply <code>x</code>
'str' object is not callable	Calling something that is not a function	Pass the function, not its result
got some positional-only arguments passed as keyword arguments	A keyword for a parameter before <code>/</code>	Give it by position

Exact wording can differ in older Python versions; the error type and the idea stay the same. The mutable default trap gives **no** error message at all, which is what makes it dangerous: look for `=[]` or `={} in a def line.`

Vocabulary

Term	Definition	Example
default value	A value in the <code>def</code> line used when a call leaves that argument out	<code>greeting="Hello"</code>
required / optional parameter	A parameter with no default / with a default	<code>name / greeting="Hello"</code>
positional argument	An argument matched to a parameter by its place in the call	<code>"Ana" in greet("Ana")</code>
keyword argument	An argument matched by parameter name, written <code>name=value</code> in a call	<code>milk="oat"</code>
sep / end	<code>print</code> 's keyword arguments for the text between values / after them	<code>print(a, b, sep="-")</code>
mutable default argument	A list, dictionary, or set as a default; made once and shared by every call	<code>def f(items=[])</code>
*args	A parameter that collects extra positional arguments into a tuple	<code>def average(*scores)</code>
unpacking (in a call)	<code>*</code> or <code>**</code> spreading a list or dictionary out into separate arguments	<code>average(*marks)</code>
**kwargs	A parameter that collects extra keyword arguments into a dictionary	<code>def make_profile(name, **details)</code>
keyword-only parameter	A parameter that can only be given by keyword; it comes after <code>*</code> or <code>*args</code>	<code>discount</code> in <code>total_price</code>
positional-only parameter	A parameter that can only be given by position; it comes before <code>/</code>	<code>iterable</code> in <code>sorted</code>
function object	The value a <code>def</code> or <code>lambda</code> creates; calling it runs its code	<code>shout</code> without <code>()</code>
higher-order function	A function that takes a function as an argument (or returns one)	<code>count_if(items, test)</code>
lambda expression	A small unnamed function that returns the value of one expression	<code>lambda n: n * 2</code>
key	A keyword argument of <code>sorted</code> , <code>sort</code> , <code>min</code> , and <code>max</code> : a function that gives each item's sort key	<code>key=len</code>
sort key	The value the key function returns for an item, used for comparing	<code>len("pear")</code> is 4
stable sort	A sort that keeps items with equal keys in their original order	<code>pear</code> stays before <code>kiwi</code>

iterable	Anything a for loop can go through	a list, a string
signature	A function's name and parameter list, which shows how to call it	<code>sorted(iterable, /, *, key=None, reverse=False)</code>

Summary

- **Default values** make parameters optional; required parameters come first. **Keyword arguments** (`name=value`) are matched by name, so they can be in any order and can skip defaults; in a call, positional arguments come first.
- A default is made **once**, when `def` runs, so a list or dictionary default is shared by every call. Use `None` and create the new value inside the function.
- A `*args` parameter collects extra positional arguments into a tuple, and a `**kwargs` parameter collects extra keyword arguments into a dictionary; in a call, one or two stars spread a list or dictionary out. Parameters after `*args` or a bare `*` are keyword-only; parameters before `/` are positional-only.
- A function is a value: its name without parentheses can be stored or passed to a **higher-order function**, and a **lambda** writes a short one-expression function in place.
- `sorted`, `sort`, `min`, and `max` take a **key** function and compare by what it returns; `sorted` and `sort` also take `reverse=True`, and the sort is stable.

Practice

Do these in your `day10` folder. Type each program yourself, run it, and only then open the answer.

Question 11 Short answer

Defaults and keywords. Write a function with at least one default argument. Call it once using only positional arguments, and at least twice using keyword arguments, including one call that skips over a default.

Question 12 Short answer

Sorting records. Make a list of at least five dictionaries describing books (title, author, year). Print it sorted by one field, then by another. Then sort by author with ties broken by year.

Question 13 Short answer

apply_twice. Write `apply_twice(func, value)`, which calls `func` on `value` and then calls `func` again on the result, returning the final value. Test it with three different functions, at least one of them a lambda.

Review questions

Question 14 Tracing

What does this program print? For each call, say what `items`, `bag`, and `max_items` hold.

pack.py

```
1 def pack(owner, *items, bag="backpack", max_items=3):
2     """Return what fits in the bag, and how many items were left out."""
3     kept = items[:max_items]
4     left_out = len(items) - len(kept)
5     return f"{owner}'s {bag}: {'', '.join(kept)} ({left_out} left out)"
6
7
8 print(pack("Ana", "book", "pen"))
9 print(pack("Ben", "map", "torch", "rope", "tent", bag="duffel"))
10 print(pack("Cal", "snack", "water", "hat", "gloves", max_items=2))
```

Question 15 Debugging

Each call to `log_event` should start a new log unless one is passed in, so Monday's log should hold only `login`. Instead the program prints the output below. Explain the bug and fix it.

log_event.py

```
1 def log_event(event, log=[]):
2     """Add event to a log and return the log."""
3     log.append(event)
4     return log
5
6
7 monday = log_event("login")
8 tuesday = log_event("logout")
9 print("Monday:", monday)
10 print("Tuesday:", tuesday)
```

```
Monday: ['login', 'logout']
Tuesday: ['login', 'logout']
```

Question 16 Multiple choice

`prices = {"tea": 2.5, "coffee": 3.0, "juice": 2.25}`. Which expression gives `'juice'`, the name of the cheapest drink?

- A. `min(prices)`
- B. `min(prices.values())`
- C. `min(prices, key=prices.get)`
- D. `max(prices, key=prices.get)`

Question 17 Tracing

A dictionary's values can be functions. What does this program print?

operations.py

```

1 operations = {
2     "double": lambda n: n * 2,
3     "square": lambda n: n * n,
4     "negate": lambda n: -n,
5 }
6 value = 3
7 for name, operation in operations.items():
8     value = operation(value)
9     print(name, value)

```

Question 18 Short answer

Instead of a tuple key, a classmate sorts the students from 10.8 by grade, with names in order inside each grade, using two `sorted` calls:

two_pass_sort.py

```

1 students = [
2     {"name": "Maya", "grade": 10, "score": 88},
3     {"name": "Leo", "grade": 9, "score": 95},
4     {"name": "Ava", "grade": 10, "score": 95},
5     {"name": "Sam", "grade": 9, "score": 72},
6 ]
7
8 by_name = sorted(students, key=lambda student: student["name"])
9 by_grade = sorted(by_name, key=lambda student: student["grade"])
10 for student in by_grade:
11     print(student["grade"], student["name"])

```

```

9 Leo
9 Sam
10 Ava
10 Maya

```

Explain why this works, and what would go wrong if the two `sorted` calls were done in the other order.

Question 19 Short answer

The documentation for `print` shows

`print(*objects, sep=' ', end='\n', file=None, flush=False)`. Using the signature, explain why `print("a", "b", "-")` does not put a `-` between `a` and `b`, and how to make it do so.

Go further

The official Python Tutorial covers today's ideas in [More Control Flow Tools: More on Defining Functions](#), including default values, keyword arguments, special parameters (`/` and `*`), arbitrary argument lists, and lambda expressions. The [Sorting Techniques](#) guide shows more ways to use `key`.

Reflection

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

- ☐ I can give a parameter a default value and say which parameters must come first. Reread: 10.1 Default Parameter Values
- ☐ I can call a function with keyword arguments, including skipping a default. Reread: 10.2 Keyword Arguments
- ☐ I can explain why a list default is shared between calls and fix it with `None`. Reread: 10.3 The Mutable Default Argument Trap
- ☐ I can write a function that takes any number of arguments with `*args`. Reread: 10.4 Any Number of Arguments: `*args`
- ☐ I can collect keyword arguments with `**kwargs` and explain keyword-only parameters. Reread: 10.5 Keyword Arguments by the Handful: `**kwargs`
- ☐ I can pass a function to another function without calling it. Reread: 10.6 Functions Are Values
- ☐ I can write a lambda and say when a `def` is better. Reread: 10.7 lambda Expressions
- ☐ I can sort a list of dictionaries by one field, and by two. Reread: 10.8 Sorting with `key`
- ☐ I can find the item with the largest or smallest key with `max` and `min`. Reread: 10.9 `min()` and `max()` with `key`
- ☐ I can read a signature with `/` and `*` and write a valid call. Reread: 10.10 Reading a Function Signature

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

Concert: $1 \times 12.5 = 12.5$
 Play: $3 \times 12.5 = 37.5$
 Film: $2 \times 8 = 16$

Call	seats	price	total
<code>ticket("Concert")</code>	1 (default)	12.5 (default)	12.5
<code>ticket("Play", 3)</code>	3 (passed)	12.5 (default)	37.5
<code>ticket("Film", 2, 8)</code>	2 (passed)	8 (passed)	16

Arguments fill parameters from the left, so the single number in `ticket("Play", 3)` goes to `seats`, not `price`. In the last call `price` is the int 8, so $2 * 8$ is the int 16, with no `.0`.

Reread *10.1 Default Parameter Values* if your output differed.

Question 2

A. "bird" goes to `animal` by position. `legs=2` is a keyword argument, so it goes to `legs` by name. `sound` gets nothing, so it uses its default, "no sound".

- **B** is what `describe("bird", 2)` prints: without the keyword, the 2 would go by position to the next parameter, `sound`.
- **C** ignores the keyword argument. The default 4 is used only when `legs` is missing, and here it was passed.
- **D** is wrong: skipping a parameter that has a default is exactly what keyword arguments allow.

Reread *10.2 Keyword Arguments* if you chose differently.

Question 3

```
{'yes': 1}
{'yes': 1, 'no': 1}
{'yes': 1}
{'yes': 2, 'no': 1}
```

Call	Which dictionary	After the call
<code>tally("yes")</code>	The shared default	<code>{'yes': 1}</code>
<code>tally("no")</code>	The shared default	<code>{'yes': 1, 'no': 1}</code>
<code>tally("yes", {})</code>	A new one from the call	<code>{'yes': 1}</code>
<code>tally("yes")</code>	The shared default again	<code>{'yes': 2, 'no': 1}</code>

The third call passes its own empty dictionary, `{}`, which is made fresh when that call runs, so it starts from nothing. The fourth call goes back to the shared default, which still holds the counts from the first two calls. Line 3 uses `get` with a default of 0 (Day 8), so a new word starts at 0 before adding 1.

Reread 10.3 *The Mutable Default Argument Trap* if your output differed.

Question 4

```
Temps: 12
Single: need two values
3: need two values
```

Call	label	values	Result
biggest_gap("Temps", 12, 19, 7, 15)	"Temps"	(12, 19, 7, 15)	19 - 7 is 12
biggest_gap("Single", 4)	"Single"	(4,)	Fewer than 2 values
biggest_gap(3, 9)	3	(9,)	Fewer than 2 values

The ordinary parameter `label` is always filled first. In the last call that takes the 3, even though it was meant as a value, and only 9 is left for `values`. An f-string turns the 3 into text, so there is no error, just a surprising label. `max` and `min` work on a tuple just as they do on a list.

Reread 10.4 *Any Number of Arguments: *args* if your output differed.

Question 5

```
Large pizza, thick crust
Toppings: ('olives', 'peppers')
Extras: {'sauce': 'spicy'}
```

Argument	Goes to	Why
"Large"	size	First positional argument
"olives", "peppers"	toppings	Leftover positional arguments, as a tuple
sauce="spicy"	extras	Keyword that matches no parameter
crust="thick"	crust	Keyword that matches a parameter

`crust` is keyword-only, because it comes after `*toppings`. The keyword `crust="thick"` matches it by name, so it does **not** go into `extras`, even though it was written last.

Reread 10.5 *Keyword Arguments by the Handful: **kwargs* if your output differed.

Question 6

B. `shout` without parentheses passes the function. `announce` stores it in `style` and calls `style("Go team")`, which returns `"GO TEAM!"`.

- **A** calls `shout` with no argument before `announce` runs:
`TypeError: shout() missing 1 required positional argument: 'text'.`
- **C** calls `shout` first and passes its result, the string `"GO TEAM!"`. Inside `announce`, `style(message)` tries to call a string: `TypeError: 'str' object is not callable.`
- **D** has the arguments in the wrong order: `message` is the function and `style` is the string `"Go team"`, so `style(message)` again tries to call a string and gives the same `TypeError`.

Reread 10.6 *Functions Are Values* if you chose differently.

Question 7

```
['Tea', 'Coffee', 'Juice']
[3, 6, 5]
['teatea', 'coffee', 'juice']
```

- The first lambda makes the first character a capital (`w[0].upper()`) and joins it to the rest (`w[1:]`, Day 3 slicing).
- The second returns each word's length, so the new list holds numbers, not strings.
- The third uses a conditional expression: only "tea", at 3 characters, is not longer than 3, so it is repeated with `w * 2`; the other two come back unchanged.

`apply_to_each` builds a **new** list each time, so `words` itself never changes between the calls.

Reread *10.7 lambda Expressions* if your output differed.

Question 8

```
[('Ana', 82), ('Ben', 95), ('Cal', 82), ('Dee', 70)]
[('Dee', 70), ('Ana', 82), ('Cal', 82), ('Ben', 95)]
[('Ben', 95), ('Ana', 82), ('Cal', 82), ('Dee', 70)]
```

- Line 2 has no key, so the tuples are compared as whole tuples: by their first items, the names, alphabetically.
- Line 3's key returns each score, so the list is sorted 70, 82, 82, 95. Ana and Cal tie on 82; Ana was before Cal in the original list, so stability keeps her first.
- Line 4 sorts by score from largest to smallest. The tie is still kept in its original order: Ana before Cal.

Reread *10.8 Sorting with key* if your output differed.

Question 9

```
kiwi
fig
pear
```

Word	len	Last letter
"kiwi"	4	i
"plum"	4	m
"fig"	3	g
"lime"	4	e
"pear"	4	r

- Four words tie for the longest, at 4 letters. `max` returns the first, `kiwi`.
- `fig` is the only 3-letter word.
- The last lambda compares last letters as strings. `r` comes latest in the alphabet, so the answer is `pear`, the word, not the letter.

Reread *10.9 min() and max() with key* if your output differed.

Question 10

D. `keep_ratio` is after the `*`, so it is keyword-only. Only `image`, `width`, and `height` can be filled by position, so a fourth positional argument has nowhere to go:

`resize_wrong.py`

```
1 def resize(image, /, width, height=None, *, keep_ratio=True):
2     """Pretend to resize an image and describe what would happen."""
3     return f"{image}: width {width}, height {height}, keep ratio {keep_ratio}"
4
5
6 print(resize("cat.png", 300, 200, False))
```

`TypeError: resize() takes from 2 to 3 positional arguments but 4 were given`

The other three calls all work:

`resize_calls.py`

```
1 def resize(image, /, width, height=None, *, keep_ratio=True):
2     """Pretend to resize an image and describe what would happen."""
3     return f"{image}: width {width}, height {height}, keep ratio {keep_ratio}"
4
5
6 print(resize("cat.png", 300))
7 print(resize("cat.png", width=300, keep_ratio=False))
8 print(resize("cat.png", 300, 200))
```

```
cat.png: width 300, height None, keep ratio True
cat.png: width 300, height None, keep ratio False
cat.png: width 300, height 200, keep ratio True
```

- **A** gives the two required arguments by position; `height` and `keep_ratio` use their defaults.
- **B** gives `image` by position (it must be: it is before the `/`), `width` by keyword (allowed: it is between `/` and `*`), and `keep_ratio` by keyword.
- **C** fills `image`, `width`, and `height` by position, which is allowed for all three.

The fix for D is `resize("cat.png", 300, 200, keep_ratio=False)`.

Reread *10.10 Reading a Function Signature* if you chose differently.

Question 11

One possible solution:

`trip_plan.py`

```
1 def describe_trip(city, days=3, transport="train"):
2     """Return a one-line trip plan."""
3     return f"{days} days in {city}, travelling by {transport}"
4
5
6 print(describe_trip("Lisbon"))
7 print(describe_trip("Lisbon", 5))
8 print(describe_trip("Lisbon", 5, "plane"))
9 print(describe_trip(city="Rome", transport="bus"))
10 print(describe_trip("Oslo", transport="ferry", days=2))
```

```

3 days in Lisbon, travelling by train
5 days in Lisbon, travelling by train
5 days in Lisbon, travelling by plane
3 days in Rome, travelling by bus
2 days in Oslo, travelling by ferry

```

The first three calls are positional: each extra argument replaces the next default from the left. `describe_trip(city="Rome", transport="bus")` is all keywords and skips `days`, which keeps its default 3; that cannot be done by position alone. The last call mixes the two: the city by position first, then two keyword arguments in an order different from the `def` line. Check that your `def` puts the required parameter before the ones with defaults.

Reread *10.1 Default Parameter Values* and *10.2 Keyword Arguments* if your result differed.

Question 12

One possible solution:

`sort_books.py`

```

1 def show(books, heading):
2     """Print a heading, then one line per book."""
3     print(heading)
4     for book in books:
5         print(f" {book['year']} {book['title']} ({book['author']})")
6
7
8 books = [
9     {"title": "Dune", "author": "Herbert", "year": 1965},
10    {"title": "Emma", "author": "Austen", "year": 1815},
11    {"title": "Beloved", "author": "Morrison", "year": 1987},
12    {"title": "Persuasion", "author": "Austen", "year": 1817},
13    {"title": "Kindred", "author": "Butler", "year": 1979},
14 ]
15
16 show(sorted(books, key=lambda book: book["title"]), "By title:")
17 show(sorted(books, key=lambda book: book["year"]), "By year:")
18 by_author = sorted(books, key=lambda book: (book["author"], book["year"]))
19 show(by_author, "By author, then year:")

```

```

By title:
1987 Beloved (Morrison)
1965 Dune (Herbert)
1815 Emma (Austen)
1979 Kindred (Butler)
1817 Persuasion (Austen)
By year:
1815 Emma (Austen)
1817 Persuasion (Austen)
1965 Dune (Herbert)
1979 Kindred (Butler)
1987 Beloved (Morrison)
By author, then year:
1815 Emma (Austen)
1817 Persuasion (Austen)
1979 Kindred (Butler)
1965 Dune (Herbert)
1987 Beloved (Morrison)

```

Each `sorted` call gets a lambda that picks one field out of a book dictionary. The last key returns a tuple, `(author, year)`, so the two Austen books, which tie on author, are put in year order. `books` itself is never changed, because `sorted` returns a new list each time. Adding `reverse=True` to the year sort would list the newest book first.

Reread *10.8 Sorting with key* if your result differed.

Question 13

One possible solution:

`apply_twice.py`

```

1 def apply_twice(func, value):
2     """Call func on value, then call func again on that result."""
3     return func(func(value))
4
5
6 def double(number):
7     return number * 2
8
9
10 def add_exclamation(text):
11     return text + "!"
12
13
14 print(apply_twice(double, 5))
15 print(apply_twice(add_exclamation, "Hello"))
16 print(apply_twice(lambda n: n ** 2, 3))

```

```

20
Hello!!
81

```

`func(func(value))` works from the inside out: the inner call's result becomes the outer call's argument. With `double`, 5 becomes 10, then 20. With `add_exclamation`, "Hello" becomes "Hello!!",

then `"Hello!!"`. The lambda squares 3 to get 9, then 9 to get 81. `apply_twice` works with numbers and strings alike because it only calls `func`; what happens to the value is up to the function you pass. Each test passes the function's name without parentheses.

Reread *10.6 Functions Are Values* and *10.7 Lambda Expressions* if your result differed.

Question 14

Ana's backpack: book, pen (0 left out)
Ben's duffel: map, torch, rope (1 left out)
Cal's backpack: snack, water (2 left out)

Call	items	bag	max_items	kept
Ana	("book", "pen")	"backpack"	3	both
Ben	("map", "torch", "rope", "tent")	"duffel"	3	first 3
Cal	("snack", "water", "hat", "gloves")	"backpack"	2	first 2

Every string after the owner goes into the `items` tuple. `bag` and `max_items` come after `*items`, so they can only be set by keyword, and otherwise keep their defaults. Slicing a tuple, `items[:max_items]`, works like slicing a list, and `", ".join(kept)` (Day 3) joins the kept items with commas.

Reread *10.4 Any Number of Arguments: *args* and *10.5 Keyword Arguments by the Handful: **kwargs* if your output differed.

Question 15

The default `log=[]` is made once, when `def` runs, and both calls share that one list. The first call appends `"login"` to it, the second appends `"logout"` to the same list, and `monday` and `tuesday` are two names for it, so both show both events. Use `None` as the default and make the list inside the function:

log_event_fixed.py

```
1 def log_event(event, log=None):
2     """Add event to a log and return the log. With no log, start a new one."""
3     if log is None:
4         log = []
5     log.append(event)
6     return log
7
8
9 monday = log_event("login")
10 tuesday = log_event("logout")
11 print("Monday:", monday)
12 print("Tuesday:", tuesday)
```

Monday: ['login']
Tuesday: ['login', 'logout']

`if log is None: log = []` runs on every call, so each call without a log gets its own new list.

Reread *10.3 The Mutable Default Argument Trap* if you missed it.

Question 16

C.

```
>>> prices = {"tea": 2.5, "coffee": 3.0, "juice": 2.25}
>>> min(prices)
'coffee'
>>> min(prices.values())
2.25
>>> min(prices, key=prices.get)
'juice'
>>> max(prices, key=prices.get)
'coffee'
```

Going through a dictionary gives its keys, the names. `key=prices.get` makes `min` compare each name by its price, and `min` returns the name, not the price.

- **A** compares the names themselves, alphabetically: `'coffee'` comes first.
- **B** finds the cheapest **price**, 2.25, but not which drink it belongs to.
- **D** uses the right key but finds the most expensive drink, `'coffee'`.

Reread *10.9 `min()` and `max()` with key* if you chose differently.

Question 17

```
double 6
square 36
negate -36
```

name	operation	value before	value after
"double"	<code>lambda n: n * 2</code>	3	6
"square"	<code>lambda n: n * n</code>	6	36
"negate"	<code>lambda n: -n</code>	36	-36

`items()` gives each name with its function, in the order the dictionary was written.

`operation(value)` calls the function, and the result is stored back in `value`, so each step starts from the last one's result, like an accumulator. The functions are values stored in a dictionary just as numbers or strings would be.

Reread *10.6 Functions Are Values* and *10.7 Lambda Expressions* if your output differed.

Question 18

Model answer: the first call puts every student in name order. The second sorts by grade, and because the sort is stable, students who tie on grade keep the order they already had, which is name order. So within each grade the names stay alphabetical. In the other order, sorting by grade first and by name second, the final sort would be by name alone: Ava, Leo, Maya, Sam, with the grades mixed up. The **last** sort decides the main order, and earlier sorts only break its ties.

A correct answer uses stability (equal keys keep their existing order) and says that the tie-breaking field must be sorted **first**.

Reread *10.8 Sorting with key* if your reasoning differed.

Question 19

```
>>> print("a", "b", "-")
a b -
>>> print("a", "b", sep="-")
a-b
>>> print("a", "b", sep="-", end="!\n")
a-b!
```

Model answer: `*objects` collects **every** positional argument into the values to print, so `"-"` is just a third value, printed after a space. `sep` comes after `*objects`, so it is keyword-only: the only way to set it is by name, `print("a", "b", sep="-")`. The same goes for `end`, which can be set together with `sep` in one call.

A correct answer says that the positional `"-"` is collected by `*objects`, and that `sep` must be given by keyword because it comes after the starred parameter.

Reread *10.10 Reading a Function Signature* and *10.2 Keyword Arguments* if your reasoning differed.