

Day 19 Type hints and code style

Self-Study Guide · About 85 minutes · With answer key

By the end you will be able to

- Add type hints to parameters, return values, and variables, including `list[int]`, `dict[str, int]`, and `X | None`.
- Run a type checker such as `mypy` or `pyright` and fix what it reports.
- Follow the PEP 8 essentials for names, whitespace, line length, and imports, and let `ruff` or `black` do the routine work.
- Write docstrings and comments that tell another developer what they need without asking you.

What to remember from earlier

- In a data class, a field written as `name: type` has a type annotation, and Python does not check it. ([Day 17: Classes and objects, part 2, 17.8 Data Classes with @dataclass](#))
- A docstring is a triple-quoted string on the first line of a function's body, and `help()` shows it. ([Day 9: Functions, part 1, 9.7 Docstrings](#))
- A function that ends without `return` gives back `None`. ([Day 9: Functions, part 1, 9.5 Returning Versus Printing](#))
- A `#` outside quotes starts a comment, and Python ignores the rest of that line. ([Day 1: Setup and your first program, 1.6 Comments: Notes Python Ignores](#))
- Variables and functions use `snake_case`, and classes use `CapWords`. ([Day 2: Variables, numbers and basic types, 2.2 Naming Rules and snake_case](#))
- With a virtual environment active, `python -m pip install name` installs a package into it. ([Day 13: Modules, packages and environments, 13.10 Installing Packages with pip](#))

Make a `day19` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days. Later in the day you will install two tools into a virtual environment, so create one in `day19` now if you like (Day 13: `python -m venv .venv`, then activate it).

A program is read far more often than it is written: by teammates, by reviewers, and by you a month from now. Yesterday's tools were about what code can **do**. Today is about making code **easy to read and hard to misuse**. You will write down what type each value should be, so a reader knows what to pass in and a tool can catch mistakes before the program runs. You will learn the shared style rules that make every Python file look familiar, and let tools apply them for you. And you will write the docstrings and comments that answer a reader's questions before they ask them.

19.1 Annotating Parameters and Return Values

Here is a function you could have written on Day 9. What should you pass to it?

total_cost.py

```
1 def total_cost(price, quantity):
2     return price * quantity
3
4
5 print(total_cost(2.5, 4))
6 print(total_cost("2.5", 4))
```

```
10.0
2.52.52.52.5
```

The name suggests numbers, but nothing says so. The second call passes the **string** "2.5", perhaps straight from `input()` (Day 3), and Python does not complain: `*` with a string and an int repeats the string (Day 3), so the "total cost" is the text 2.52.52.52.5. A docstring (Day 9) could say "price is a number", but a sentence in English is easy to skip and impossible for a tool to check.

Day 17 showed a better way to say it. In a data class, `x: int` noted that the field `x` is meant to hold an int. That note was a **type annotation**. You can write the same kind of note on a function's parameters and on its result. When an annotation states a type, it is called a **type hint**: a hint to the reader (and to tools) about what kind of value belongs there.

The syntax has two parts:

- A **parameter hint** goes after the parameter's name: a colon, a space, then the type, as in `price: float`.
- A **return hint** goes after the closing parenthesis of the parameter list: a space, an arrow `->` (a minus sign and a greater-than sign), a space, then the type of the value the function returns, and then the usual colon that ends the `def` line.

WORKED EXAMPLE A receipt with type hints

receipt_hints.py

```
1 def line_total(price: float, quantity: int = 1) -> float:
2     """Return the cost of quantity items at price each."""
3     return price * quantity
4
5
6 def print_line(item: str, price: float, quantity: int = 1) -> None:
7     """Print one receipt line, such as '4 x pens: 10.00'."""
8     print(f"{quantity} x {item}: {line_total(price, quantity):.2f}")
9
10
11 print_line("pens", 2.5, 4)
12 print_line("ruler", 1.75)
13 help(line_total)
```

```

4 x pens: 10.00
1 x ruler: 1.75
Help on function line_total in module __main__:

line_total(price: float, quantity: int = 1) -> float
    Return the cost of quantity items at price each.

```

Read line 1 piece by piece:

Piece	Meaning
price: float	price should be a float
quantity: int	quantity should be an int
= 1	A default value (Day 10). With a hint, it goes after the type
-> float	The function returns a float
: at the end	The usual colon that starts the body

- Line 6: `print_line` prints its line but returns nothing, so its return hint is `-> None`. A function that never runs `return` with a value gives back `None` (Day 9), and `-> None` says so on purpose. Without it, a reader cannot tell "returns nothing" from "the author forgot to add a hint".
- Line 8 calls `line_total`, which returns a float, and formats it with Day 3's `:.2f`.
- Line 13: `help()` (Day 9) now shows the hints in the function's signature, so anyone looking up `line_total` sees what to pass without reading its body.

Notice the spaces in `quantity: int = 1`. On Day 10 you wrote defaults with no spaces, `quantity=1`. Once there is a hint, Python's style guide puts a space on each side of the `=`, so the hint and the default are easy to tell apart. There is no space before the colon and one after it, and one space on each side of `->`.

Now the most important fact about hints. Change the function to have hints, and call it with the wrong types again:

hints_ignored.py

```

1 def line_total(price: float, quantity: int = 1) -> float:
2     """Return the cost of quantity items at price each."""
3     return price * quantity
4
5
6 print(line_total(2.5, 4))
7 print(line_total("2.5", 4))
8 print(line_total([0], 3))

```

```

10.0
2.52.52.52.5
[0, 0, 0]

```

The hints did not stop anything. **Python does not check type hints when it runs your program.** They are stored with the function (that is how `help()` found them), and otherwise ignored, exactly like the data class fields on Day 17 that accepted `Point("two", "three")`. Python works this way

on purpose: hints were added to the language long after it was created, and checking every argument on every call would slow every program down. So who reads the hints? People, your editor, which can suggest and warn as you type, and a **type checker**, a separate tool that you will run in 19.4.

KEY IDEA

A type hint states what type a value should have: `name: type` after a parameter, and `-> type` before the colon of the `def` line for the return value, with `-> None` for a function that returns nothing. Python stores hints but does not check them when the program runs.

WATCH OUT

Writing = where you mean :. `price=float` does not say "price is a float". It makes the `class float` itself the default value of `price`. The function still runs when you pass every argument, which hides the mistake until a call leaves one out:

`equals_not_colon.py`

```
1 def line_total(price=float, quantity=int):
2     """Return the cost of quantity items at price each."""
3     return price * quantity
4
5
6 print(line_total(2.5, 4))
7 print(line_total(2.5))
```

10.0

Traceback (most recent call last):

File ".../day19>equals_not_colon.py", line 7, in <module>
 print(line_total(2.5))

~~~~~^~~~~~

File ".../day19>equals\_not\_colon.py", line 3, in line\_total  
 return price \* quantity

~~~~~^~~~~~

TypeError: unsupported operand type(s) for *: 'float' and 'type'

On line 7, `quantity` takes its default, which is the type `int` itself (shown as `'type'` in the message), and a number cannot be multiplied by a type. A hint uses a colon: `price: float, quantity: int`. If you also want a default, add it after the hint: `quantity: int = 1`.

PAUSE AND TRY

Question 1 Multiple choice

A function `make_tag` takes a person's name and their age in whole years, and returns a name tag as a string. Which `def` line has the right hints?

- A. `def make_tag(name: str, age: int) -> str:`
- B. `def make_tag(name=str, age=int) -> str:`

- C. `def make_tag(name: str, age: int): -> str`
 D. `def make_tag(str: name, int: age) -> str:`

Question 2 Tracing

What does this program print? Think about what Python does with hints when it runs.

hint_trace.py

```
1 def repeat(word: str, times: int) -> str:
2     return word * times
3
4
5 def half(number: int) -> int:
6     return number / 2
7
8
9 print(repeat("ha", 3))
10 print(repeat(3, 2))
11 print(half(7))
```

19.2 Hints for Lists, Dictionaries, and Tuples

A hint of `list` tells a reader that a parameter is a list, but a list of **what**? Scores, names, tasks? A function that averages scores needs numbers in its list, and a dictionary could map anything to anything. Hints can say what goes **inside** a collection too, by putting the inner types in square brackets after the collection's type:

| Hint | Means | A value that fits |
|-----------------------------------|---|---|
| <code>list[int]</code> | A list in which every item is an int | <code>[85, 92, 78]</code> |
| <code>set[str]</code> | A set in which every item is a string | <code>{"red", "blue"}</code> |
| <code>dict[str, int]</code> | A dictionary with string keys and int values | <code>{"Ana": 3, "Ben": 5}</code> |
| <code>tuple[str, int]</code> | A tuple of exactly two items: first a string, then an int | <code>("Ana", 92)</code> |
| <code>list[list[int]]</code> | A list whose items are lists of ints (a grid, Day 6) | <code>[[1, 2], [3, 4]]</code> |
| <code>dict[str, list[int]]</code> | A dictionary from strings to lists of ints | <code>{"Ana": [90, 85], "Ben": [72]}</code> |

A list or a set takes **one** inner type, because every item is meant to be the same kind of thing. A dictionary takes two: the key type, then the value type. A tuple is different: it lists a type **for each position**, because a tuple usually holds a fixed number of values with different meanings (Day 7), like a name and a score. Read a nested hint from the outside in: `dict[str, list[int]]` is "a dictionary whose keys are strings and whose values are lists of ints".

WORKED EXAMPLE Counting words, with hints

word_counts.py

```

1 def count_words(text: str) -> dict[str, int]:
2     """Return how many times each word appears in text."""
3     counts: dict[str, int] = {}
4     for word in text.lower().split():
5         counts[word] = counts.get(word, 0) + 1
6     return counts
7
8
9 def most_common(counts: dict[str, int]) -> tuple[str, int]:
10    """Return the most common word and its count."""
11    best_word = ""
12    best_count = 0
13    for word, count in counts.items():
14        if count > best_count:
15            best_word, best_count = word, count
16    return best_word, best_count
17
18
19 counts = count_words("The cat saw the dog and the dog saw the cat")
20 print(counts)
21 word, count = most_common(counts)
22 print(f"{word} appears {count} times")

```

```

{'the': 4, 'cat': 2, 'saw': 2, 'dog': 2, 'and': 1}
the appears 4 times

```

- Line 1: `count_words` takes a string and returns a dictionary from each word (a string) to its count (an int), which is `dict[str, int]`.
- Line 3 is new. A hint can go on an ordinary **variable** too, with the same `name: type` form as a data class field, followed by `=` and its first value. This is a **variable annotation**. `{}` on its own says nothing about what will go into the dictionary; the annotation records the plan, so a reader (and a type checker) knows that only string keys and int values belong in `counts`. Most variables do not need one, because their first value already shows the type: `total = 0` is clearly an int. Annotate a variable when it starts empty or could otherwise be unclear.
- Line 9: `most_common` returns two values. Day 9 showed that `return a, b` really returns one tuple, so the return hint is `tuple[str, int]`: a word, then its count. Line 21 unpacks it (Day 7).

Hints can nest as deep as the data does. Here a gradebook maps each student to a list of scores:

class_averages.py

```

1 def averages(grades: dict[str, list[int]]) -> dict[str, float]:
2     """Return each student's average grade, rounded to 1 decimal place."""
3     result: dict[str, float] = {}
4     for name, scores in grades.items():
5         result[name] = round(sum(scores) / len(scores), 1)
6     return result
7
8
9 def top_students(results: dict[str, float], limit: float) -> list[str]:
10    """Return the names of students whose average is at least limit."""
11    return [name for name, average in results.items() if average >= limit]
12
13
14 grades = {"Ana": [90, 85, 88], "Ben": [72, 80], "Cy": [95, 91, 99]}
15 results = averages(grades)
16 print(results)
17 print(top_students(results, 85))

```

```

{'Ana': 87.7, 'Ben': 76.0, 'Cy': 95.0}
['Ana', 'Cy']

```

Each function's first line now tells you the shape of the data it expects and returns: `averages` turns `dict[str, list[int]]` (names to lists of scores) into `dict[str, float]` (names to one average each), and `top_students` returns `list[str]`, a list of names. Ana's average is $(90 + 85 + 88) / 3 = 87.67$, rounded to 87.7, and Cy's is 95.0; both reach the limit of 85, Ben's 76.0 does not.

Any class can be a type in a hint, including your own. If `Task` is the data class from Day 17, then `tasks: list[Task]` means "a list of `Task` objects", and `-> Task` means the function returns one. You will hint the whole to-do manager this way in today's practice.

You may also meet older code that writes `List[int]` and `Dict[str, int]`, with capital letters, imported from the standard library's `typing` module (from `typing import List, Dict`). Python versions before 3.9 needed those. They mean the same thing; write the lowercase built-in forms in new code.

KEY IDEA

Put inner types in square brackets: `list[int]` and `set[str]` take one type for every item, `dict[str, int]` takes the key type and then the value type, and `tuple[str, int]` gives one type per position. Hints nest, any class can be a type, and `name: type = value` annotates a variable that starts empty.

WATCH OUT

Listing one type per item for a list. A list of three ints is not `list[int, int, int]`: a list hint takes exactly one type, which applies to every item however many there are. Type checkers reject `list[int, int, int]` as having too many type arguments. If the positions really mean different things, such as a name and a score, the value should probably be a tuple, and then

`tuple[str, int]` is right. The reverse mistake also happens: `tuple[int]` means a tuple of **exactly one** int, not "a tuple of ints".

PAUSE AND TRY

Question 3 Multiple choice

A function returns a value like `{"Ana": [90, 85], "Ben": [72]}`. Which return hint describes it?

- A. `dict[str, int]`
- B. `dict[list[int], str]`
- C. `dict[str, list[int]]`
- D. `list[dict[str, int]]`

Question 4 Short answer

Write the `def` line, with hints, for each function: (a) `cheapest(prices)` takes a dictionary from item names to prices such as `2.25`, and returns the name of the cheapest item; (b) `split_name(full_name)` takes `"Ana Lee"` and returns the pair `("Ana", "Lee")`; (c) `grid(rows, columns)` takes two whole numbers and returns a list of rows, each a list of zeros.

19.3 Values That Might Be None

Many functions you have written return a value **or** `None`. Day 8's `get()` returns `None` for a missing key, and the to-do manager's `complete_task` (Day 17) returns the `Task` it changed, or `None` if the task number does not exist. What is the return hint for "a float, or `None` if the item is not found"? -> `float` would be a lie, and a caller who believed it would use the result as a number without checking.

Python writes "one type or another" with the `|` symbol, read as "or". It is the same symbol that joined two sets on Day 7, here joining two types. A hint such as `int | str` (an int **or** a string) is called a **union type**, and the most common union by far is `X | None`: "a value of type X, or `None`".

WORKED EXAMPLE A price that might be missing

`find_price.py`

```
1 def find_price(prices: dict[str, float], item: str) -> float | None:
2     """Return the price of item, or None if the shop does not sell it."""
3     return prices.get(item)
4
5
6 prices = {"apple": 0.5, "bread": 2.25}
7 for item in ["bread", "milk"]:
8     price = find_price(prices, item)
9     if price is None:
10        print(f"Sorry, we do not sell {item}.")
11    else:
12        print(f"Three {item}: {price * 3:.2f}")
```

```
Three bread: 6.75
Sorry, we do not sell milk.
```

- Line 1: the return hint `float | None` warns every caller that the result might be `None`.
- Line 3: `get()` returns the price, or `None` when the key is missing (Day 8). The hint describes exactly that.
- Line 9 is the check the hint asks for. `is None` (Day 4) splits the two cases. Inside the `else` branch, `price` cannot be `None`, so line 12 can safely multiply it: $2.25 \times 3 = 6.75$.

The same idea works for parameters. A parameter whose default is `None` (Day 10 used this to avoid the shared-list trap) has the hint `X | None`, with the default after it. And a union does not have to include `None`:

`greet_optional.py`

```
1 def greeting(name: str | None = None) -> str:
2     """Return a greeting for name, or for everyone if no name is given."""
3     if name is None:
4         return "Hello, everyone!"
5     return f"Hello, {name}!"
6
7
8 def seat_label(seat: int | str) -> str:
9     """Return a label for a seat given as a number (12) or a code ("12B")."""
10    return f"Seat {seat}"
11
12
13 print(greeting())
14 print(greeting("Ana"))
15 print(seat_label(12), seat_label("12B"))
```

```
Hello, everyone!
Hello, Ana!
Seat 12 Seat 12B
```

`name: str | None = None` says that `name` is a string, or `None`, and is `None` when left out; line 3 handles that case first. `seat: int | str` accepts a seat number like 12 or a code like "12B"; the f-string works for both. Unions of unrelated types are less common than `X | None`, because a function that accepts many types has to handle each one, but they are sometimes exactly right.

The `|` form needs Python 3.10 or newer. In older code you will see the same hints written with names from the `typing` module: `Optional[float]` means `float | None`, and `Union[int, str]` means `int | str`.

KEY IDEA

`X | None` means "a value of type `X`, or `None`". Use it whenever a function can return `None` or a parameter defaults to `None`, and check with `is None` before using the value. `|` joins any types, as in `int | str`.

WATCH OUT

Using a maybe-None result without checking. The hint warns you, but Python does not stop you:

none_crash.py

```
1 def find_price(prices: dict[str, float], item: str) -> float | None:
2     """Return the price of item, or None if the shop does not sell it."""
3     return prices.get(item)
4
5
6 prices = {"apple": 0.5, "bread": 2.25}
7 for item in ["bread", "milk"]:
8     price = find_price(prices, item)
9     print(f"Three {item}: {price * 3:.2f}")
```

Three bread: 6.75

Traceback (most recent call last):

File ".../day19/none_crash.py", line 9, in <module>

print(f"Three {item}: {price * 3:.2f}")

~~~~~^~

TypeError: unsupported operand type(s) for \*: 'NoneType' and 'int'

Bread works, so the bug hides until someone asks for milk. `NoneType` in the message is the type of `None` (Day 2): `price` was `None`. Whenever a hint says `| None`, handle the `None` case before using the value, as `find_price.py` does. In 19.4 a type checker finds this line for you without running the program.

**PAUSE AND TRY****Question 5** Tracing

What does this program print?

none\_trace.py

```
1 def first_long_word(words: list[str], length: int) -> str | None:
2     for word in words:
3         if len(word) >= length:
4             return word
5     return None
6
7
8 words = ["fig", "plum", "banana"]
9 for length in [4, 6, 7]:
10     found = first_long_word(words, length)
11     if found is None:
12         print(length, "none")
13     else:
14         print(length, found.upper())
```

**Question 6** Multiple choice

The to-do manager's `delete_task(tasks, number)` removes and returns the `Task` with that number, or returns `None` if there is no such task. Which return hint is right?

- A. `-> Task`
- B. `-> None`
- C. `-> Task | None`
- D. `-> list[Task]`

## 19.4 Checking Types with mypy or pyright

Hints are only notes until something reads them. Here is a short program whose hints are right but whose code is wrong in two places. It runs without a single error:

checkout.py

```
1 def label(item: str, quantity: int) -> str:
2     """Return a label such as '3 x pens'."""
3     return f"{quantity} x {item}"
4
5
6 def total_items(basket: dict[str, int]) -> int:
7     """Return how many items are in the basket altogether."""
8     total = 0
9     for quantity in basket.values():
10         total += quantity
11     return f"{total} items"
12
13
14 basket = {"pens": 3, "pads": 2}
15 print(label(3, "pens"))
16 print(total_items(basket))
```

```
pens x 3
5 items
```

Both lines are wrong. Line 15 passes the arguments in the wrong order, so the label is backwards. Line 11 returns a string, although the hint promises an int; the output looks fine today, but the first caller that does arithmetic with the result, such as `total_items(basket) + 1`, will crash. Running a program only tests the lines and values that happen to run. A bug in a rarely used branch can wait months for the day it runs.

A **type checker** is a program that reads your code **without running it** and reports every place where the values do not match the hints. Checking code by reading it, without running it, is called **static analysis** ("static" because the program stands still). The two most widely used type checkers for Python are **mypy** and **pyright**. Both are third-party tools: they are not part of Python, and you install them with pip (Day 13), ideally into the project's virtual environment. You only need one; they find the same kinds of mistakes.

```
python -m pip install mypy
mypy checkout.py
```

or

```
python -m pip install pyright
pyright checkout.py
```

Type the second line of each pair in the terminal, in the folder holding the file. Use `mypy .` (or `pyright .`) to check every `.py` file in the current folder and the folders inside it; the dot means "this folder". Your editor can show the same complaints as you type: VS Code's Python extension includes Pylance, which is built on `pyright`, and underlines mismatches once its type checking setting is turned on.

### WORKED EXAMPLE What mypy reports

Here is what `mypy` prints for `checkout.py`. This is **sample output**: the exact wording and layout vary between versions.

```
checkout.py:11: error: Incompatible return value type (got "str", expected "int")
[return-value]
checkout.py:15: error: Argument 1 to "label" has incompatible type "int"; expected
"str" [arg-type]
checkout.py:15: error: Argument 2 to "label" has incompatible type "str"; expected
"int" [arg-type]
Found 3 errors in 1 file (checked 1 source file)
```

Each line is one complaint. Read it left to right:

| Part                                        | Meaning                                                     |
|---------------------------------------------|-------------------------------------------------------------|
| <code>checkout.py:11:</code>                | The file and the line number                                |
| <code>error:</code>                         | How serious it is                                           |
| <code>Incompatible return value type</code> | What does not match                                         |
| <code>(got "str", expected "int")</code>    | The type it found, then the type the hint promised          |
| <code>[return-value]</code>                 | The short name of the rule, useful for looking the error up |

So `mypy` found both bugs: line 11 returns a `str` where the hint says `int`, and line 15's two arguments are swapped (one complaint for each). Fix the code, not the hints, because the hints describe what the functions are supposed to do:

## checkout\_fixed.py

```
1 def label(item: str, quantity: int) -> str:
2     """Return a label such as '3 x pens'."""
3     return f"{quantity} x {item}"
4
5
6 def total_items(basket: dict[str, int]) -> int:
7     """Return how many items are in the basket altogether."""
8     total = 0
9     for quantity in basket.values():
10         total += quantity
11     return total
12
13
14 basket = {"pens": 3, "pads": 2}
15 print(label("pens", 3))
16 print(f"{total_items(basket)} items")
```

```
3 x pens
5 items
```

Run the checker again, and it reports something like:

```
Success: no issues found in 1 source file
```

A few things to know about how type checkers behave:

- **They can only check what is hinted.** mypy, by default, does not check the inside of a function that has no hints at all, so an unhinted function is invisible to it. Hint every function's parameters and return value; that is also today's first practice exercise.
- **An int is accepted where a float is expected.** A whole number is a perfectly good price, so `line_total(2, 3)` passes a check even though the hint says `price: float`. The reverse is not true: a float where an int is expected is reported.
- **A complaint is a question, not a verdict.** Usually it is a real bug. Sometimes the code is right and the hint is wrong, for example `-> int` on a function that really should return a string. Then fix the hint. Either way, the hint and the code end up agreeing.
- **Checking does not replace running.** A type checker proves that the types fit, not that the answers are right: `return price + quantity` instead of `*` has perfectly good types. Day 20's tests check the answers.

## KEY IDEA

A type checker such as mypy or pyright reads your code without running it and reports every place where a value does not match a hint, with the file, the line, the type it found, and the type it expected. It finds bugs in lines that have not run yet, but only in code that has hints.

**WATCH OUT**

**Using an X | None value without checking it.** This program runs fine, because "Water plants" is in the list:

find\_task\_unchecked.py

```
1 from dataclasses import dataclass
2
3
4 @dataclass
5 class Task:
6     title: str
7     done: bool = False
8
9
10 def find_task(tasks: list[Task], title: str) -> Task | None:
11     """Return the task with this title, or None if there is none."""
12     for task in tasks:
13         if task.title == title:
14             return task
15     return None
16
17
18 tasks = [Task("Practice piano"), Task("Water plants")]
19 task = find_task(tasks, "Water plants")
20 print(task.done)
```

False

A type checker still complains about line 20 (sample mypy output; the wording varies by version):

```
find_task_unchecked.py:20: error: Item "None" of "Task | None" has no attribute "done" [union-attr]
```

"Item None of Task | None" means: one of the two possible types of `task` is `None`, and `None` has no attribute `done`. The checker does not know which title you will search for, so it considers every value the hint allows, including the one that crashed `none_crash.py` in 19.3. Add the check, and the complaint goes away:

## find\_task\_checked.py

```
1 from dataclasses import dataclass
2
3
4 @dataclass
5 class Task:
6     title: str
7     done: bool = False
8
9
10 def find_task(tasks: list[Task], title: str) -> Task | None:
11     """Return the task with this title, or None if there is none."""
12     for task in tasks:
13         if task.title == title:
14             return task
15     return None
16
17
18 tasks = [Task("Practice piano"), Task("Water plants")]
19 for title in ["Water plants", "Swim"]:
20     task = find_task(tasks, title)
21     if task is None:
22         print(f"No task called {title}")
23     else:
24         print(task.done)
```

```
False
No task called Swim
```

After `if task is None:`, the checker knows that `task` in the `else` branch is a `Task`, so `task.done` is safe.

## PAUSE AND TRY

## Question 7 Multiple choice

## area\_check.py

```
1 def area(width: float, height: float) -> float:
2     return width * height
3
4
5 def describe(name: str, size: float) -> str:
6     return f"{name}: {size} square meters"
7
8
9 print(describe("kitchen", area(3, 4)))
10 print(describe(area(2.5, 4), "hall"))
```

Which line does a type checker report?

- A. The first `print` line, because 3 and 4 are ints, not floats
- B. The second `print` line, because the arguments to `describe` are in the wrong order

- C. The `return f"..."` line, because an f-string is not a float
- D. No line, because the program runs without an error

### Question 8 Short answer

A classmate says: "My program ran and printed the right answer, so running a type checker on it is a waste of time." Give two reasons they are wrong.

## 19.5 PEP 8: Naming and Whitespace

Python accepts both of these programs, and both print the same line. Which would you rather read?

`messy_grades.py`

```
1 MaxScore=100
2 def Percent(Score,outOf = MaxScore):
3     return round(Score/outOf*100 ,1)
4 class student_record:
5     def __init__( self,name,score ):
6         self.name=name
7         self.score=score
8     def summary(self):
9         return f"{self.name}: {Percent(self.score)}%"
10 print(student_record ('Ana',87).summary())
```

`tidy_grades.py`

```
1 MAX_SCORE = 100
2
3
4 def percent(score, out_of=MAX_SCORE):
5     return round(score / out_of * 100, 1)
6
7
8 class StudentRecord:
9     def __init__(self, name, score):
10         self.name = name
11         self.score = score
12
13     def summary(self):
14         return f"{self.name}: {percent(self.score)}%"
15
16
17 print(StudentRecord("Ana", 87).summary())
```

Ana: 87.0%

The first one makes you work. Is `Percent` a class or a function? Is `student_record` a variable? Where does one definition end and the next begin? None of this matters to Python, but all of it matters to the people who read the code. A program is read many more times than it is written, so

the Python community agreed on one shared set of style rules. Once everyone follows them, any Python file looks familiar and the reader can concentrate on what the code does.

Changes to Python are proposed in numbered documents called **PEPs** (Python Enhancement Proposals). **PEP 8** is the official style guide for Python code; you met a few of its rules on Days 9 and 10. Its main naming and whitespace rules are below. Each one has the same reason behind it: code is easier to read when the shape of a line tells you what it is.

## Naming

| Kind of name       | Style                          | Examples                                                               |
|--------------------|--------------------------------|------------------------------------------------------------------------|
| Variable, function | snake_case                     | <code>total_cost</code> , <code>percent</code> , <code>add_task</code> |
| Method, attribute  | snake_case                     | <code>summary</code> , <code>self.score</code>                         |
| Class, exception   | CapWords                       | <code>StudentRecord</code> , <code>TaskNumberError</code>              |
| Constant (Day 13)  | UPPER_CASE with <code>_</code> | <code>MAX_SCORE</code> , <code>SAVE_FILE</code>                        |
| Module (file)      | short, all lowercase           | <code>tasks.py</code> , <code>reading_log.py</code>                    |

Because the styles differ, a name tells you what it is at a glance: `StudentRecord(...)` makes an object, `percent(...)` calls a function, and `MAX_SCORE` should never be reassigned. Two more rules: never use `l` (lowercase L), `O` (capital o), or `I` (capital i) as a one-letter name, because in many fonts they look like the digits 1 and 0; and choose names that say what a value **means** (score, not s).

## Whitespace

| Rule                                                                                           | Write                                 | Not                                |
|------------------------------------------------------------------------------------------------|---------------------------------------|------------------------------------|
| One space each side of <code>=</code> , <code>+=</code> , comparisons, and <code>and/or</code> | <code>total = price + tax</code>      | <code>total=price+tax</code>       |
| A space after a comma, none before                                                             | <code>percent(score, 100)</code>      | <code>percent(score ,100)</code>   |
| No space just inside brackets                                                                  | <code>summary(self), scores[0]</code> | <code>summary( self )</code>       |
| No space before the <code>(</code> of a call or the <code>[</code> of an index                 | <code>print(x), scores[0]</code>      | <code>print (x), scores [0]</code> |
| No spaces around <code>=</code> for a default or keyword argument with no hint                 | <code>out_of=MAX_SCORE</code>         | <code>out_of = MAX_SCORE</code>    |
| Spaces around <code>=</code> when the parameter has a hint (19.1)                              | <code>rate: float = 0.08</code>       | <code>rate: float=0.08</code>      |
| Indent with 4 spaces                                                                           | (as in every example)                 | tabs, or 2 spaces                  |
| Two blank lines around top-level functions and classes, one between methods                    | (see <code>tidy_grades.py</code> )    | none                               |

### WORKED EXAMPLE Tidying the grades program

Compare the two files line by line:

| Messy                     | Tidy                         | Rule                                                |
|---------------------------|------------------------------|-----------------------------------------------------|
| <code>MaxScore=100</code> | <code>MAX_SCORE = 100</code> | Constants in capitals; spaces around <code>=</code> |

|                                                 |                                                        |                                                                                           |
|-------------------------------------------------|--------------------------------------------------------|-------------------------------------------------------------------------------------------|
| <code>def Percent(Score,outOf<br/>= ...)</code> | <code>def percent(score,<br/>out_of=...)</code>        | Function and parameters in snake_case; space after commas; no spaces around a default's = |
| <code>Score/outOf*100 ,1</code>                 | <code>score / out_of * 100, 1</code>                   | Spaces around operators; none before a comma                                              |
| <code>class student_record:</code>              | <code>class StudentRecord:</code>                      | Classes in CapWords                                                                       |
| <code>__init__( self,name,score )</code>        | <code>__init__(self, name, score)</code>               | No spaces inside brackets                                                                 |
| no blank lines                                  | two between top-level definitions, one between methods | Blank lines show where each definition ends                                               |
| <code>student_record ( 'Ana',87)</code>         | <code>StudentRecord("Ana", 87)</code>                  | No space before ( ; double quotes, as everywhere in this course                           |

PEP 8 itself allows single or double quotes, as long as a file is consistent; most projects, and the formatters in 19.7, choose double quotes. Nothing in the tidy version changes what the program does. Python ran both identically; only the reader's experience changed.

PEP 8 also warns against following it blindly. Its first section is titled "A Foolish Consistency is the Hobgoblin of Little Minds": consistency within one project matters more than any single rule, and a rule can be broken when following it would make a particular line harder to read. When you join an existing project, follow the style it already uses.

#### KEY IDEA

PEP 8 is Python's official style guide. Name variables and functions in snake\_case, classes in CapWords, and constants in UPPER\_CASE; put single spaces around = and operators and after commas, none inside brackets; indent 4 spaces; and separate top-level definitions with two blank lines.

#### WATCH OUT

**Naming a variable after a built-in.** list, sum, max, str, and input are not keywords (Day 2), so Python lets you use them as variable names. The new name then hides the built-in for the rest of the file:

shadow\_list.py

```
1 list = ["milk", "eggs", "bread"]
2 print(len(list), "items")
3 letters = list("abc")
4 print(letters)
```

```
3 items
Traceback (most recent call last):
  File ".../day19/shadow_list.py", line 3, in <module>
    letters = list("abc")
TypeError: 'list' object is not callable
```

After line 1, `list` means the shopping list, not the built-in function, and a list cannot be called. This is the same name-hiding that Day 13 showed for a file called `random.py`. Use a name that says what the list holds, such as `shopping`. If a built-in's name really is the best word, programmers add an underscore at the end, `list_`, the habit PEP 8 recommends for names that clash with keywords, such as `class_`.

### PAUSE AND TRY

#### Question 9 Multiple choice

Which line follows PEP 8's naming rules?

- A. `class bank_account:`
- B. `MAX_RETRIES = 3`
- C. `def CalculateTotal(prices):`
- D. `TaxRate = 0.08` for a variable that changes as the program runs

#### Question 10 Short answer

List every PEP 8 problem in these lines, and write the tidy version.

```
1 def AddBonus( points,bonus = 10 ):
2     Total=points+bonus
3     return Total
```

## 19.6 PEP 8: Line Length and Imports

Two more groups of rules decide how a file looks from a distance: how long its lines are, and how its imports are arranged at the top.

### Line length

PEP 8 limits lines to **79 characters** (and comments and docstrings to 72). Short lines let you put two files side by side, read code in a narrow window or on a phone, and compare two versions of a file without scrolling sideways. Many teams relax the limit to 88 characters, which is what the formatters in 19.7 use by default. Whichever limit your project chooses, long lines need a way to break.

A line of Python normally ends where the line ends. But **inside parentheses, square brackets, or curly braces, a statement can continue onto the next line**, because Python knows the statement is not finished until the bracket closes. This is called **implicit line continuation**. You have already seen it in Day 17's `main.py`, where one `from tasks import (...)` statement ran over several lines.

**WORKED EXAMPLE** Breaking long lines inside brackets

long\_lines.py

```

1 def can_ship(weight_kg: float, length_cm: float, country: str) -> bool:
2     """Return True if a parcel meets every shipping rule."""
3     return (
4         weight_kg <= 20
5         and length_cm <= 120
6         and country in ["Canada", "Mexico", "United States"]
7     )
8
9
10 parcels = [
11     ("books", 4.2, 35, "Canada"),
12     ("bike", 14.0, 150, "Mexico"),
13 ]
14 for name, weight_kg, length_cm, country in parcels:
15     allowed = can_ship(weight_kg, length_cm, country)
16     print(
17         f"{name}: {weight_kg} kg, {length_cm} cm to {country}",
18         "- OK" if allowed else "- too big or too heavy",
19     )

```

```

books: 4.2 kg, 35 cm to Canada - OK
bike: 14.0 kg, 150 cm to Mexico - too big or too heavy

```

- Lines 3 to 7: a long condition is wrapped in parentheses, so it can be split into one test per line. PEP 8 recommends putting each `and` (or `+`, or any operator) at the **start** of a line, so you can scan down the left edge and see how the pieces combine. The closing `)` goes on its own line, lined up with `return`.
- Lines 10 to 13: a list with one item per line. The comma after the last item is allowed, and it means a new item can be added later by changing only one line.
- Lines 16 to 19: a long `print` call with one argument per line, indented 4 spaces more than the line that opened the parenthesis.

Without the brackets, none of these breaks would be allowed.

**Imports**

PEP 8's rules for `import` statements (Day 13) are:

1. Put all imports at the **top of the file**, after the module's docstring and before anything else, so a reader sees at once what the file depends on.
2. Put **one module per import line**: `import json` and `import os` on separate lines, not `import json, os`. Importing several names from one module, `from tasks import add_task, list_tasks`, is fine.
3. Arrange imports in **three groups**, with a blank line between groups: the standard library first, then third-party packages (installed with pip), then your own modules. Many teams also sort each group alphabetically.
4. Avoid `from module import *` (Day 13): it hides where names come from.

```
1 """Fetch the weather and save a daily report."""
2
3 import json
4 from datetime import date
5 from pathlib import Path
6
7 import requests
8
9 from report import format_report
```

`json`, `datetime`, and `pathlib` are in the standard library; `requests` is the third-party package from Day 13 (it is not installed here, so this block is for reading only); `report` is a module of your own. Seeing the groups, a reader knows at once that this program needs `requests` installed.

#### KEY IDEA

Keep lines to 79 characters (or your project's limit, often 88). Break a long line inside parentheses, brackets, or braces, with operators at the start of each new line. Put imports at the top, one module per line, grouped as standard library, third-party, then your own modules.

#### WATCH OUT

**Breaking a line outside brackets.** Without an open bracket, Python sees the end of the line as the end of the statement:

`broken_line.py`

```
1 price = 19.99
2 tax = 1.60
3 shipping = 4.50
4 total = price + tax +
5     shipping
6 print(f"{total:.2f}")
```

```
File ".../day19/broken_line.py", line 4
    total = price + tax +
                        ^
```

```
SyntaxError: invalid syntax
```

Line 4 ends with `+` and nothing after it, which is not a complete expression. Wrap the expression in parentheses so it can continue:

## broken\_line\_fixed.py

```
1 price = 19.99
2 tax = 1.60
3 shipping = 4.50
4 total = (
5     price
6     + tax
7     + shipping
8 )
9 print(f"{total:.2f}")
```

26.09

(This expression is short enough for one line, `total = price + tax + shipping`, which is better; the split only shows the form for a line that is genuinely too long.) Python also lets you end a line with a backslash `\` to continue it, but PEP 8 prefers parentheses: a space typed after the backslash breaks it, and you cannot see the mistake.

## PAUSE AND TRY

## Question 11 Multiple choice

A file needs `json` and `pathlib` from the standard library, the third-party package `requests`, and the function `load_tasks` from the file `tasks.py` in the same project. Which import block follows PEP 8?

- A. `import json, requests` on one line, then `from pathlib import Path`, then `from tasks import load_tasks`
- B. `import json`, `from pathlib import Path`, a blank line, `import requests`, a blank line, `from tasks import load_tasks`
- C. `from tasks import load_tasks`, a blank line, `import requests`, a blank line, `import json`, `from pathlib import Path`
- D. `import json` at the top, and `import requests` inside the function that uses it, just before the call

## Question 12 Debugging

The author split a long `return` line to keep it under 79 characters, and now the program will not run. Explain the error and fix it.

## honor\_roll.py

```
1 def on_honor_roll(average: float, attendance: float, late_days: int) -> bool:
2     """Return True if a student qualifies for the honor roll."""
3     return average >= 90 and attendance >= 0.95
4         and late_days <= 3
5
6
7 print(on_honor_roll(93.5, 0.97, 1))
```

## 19.7 Formatting and Linting with ruff or black

You could tidy every file by hand with the tables from 19.5 and 19.6, but nobody wants to count spaces, and people miss things. Programmers hand that job to tools, so the people can argue about what the code does instead of where the spaces go. There are two kinds:

- A **formatter** rewrites the layout of your code automatically: spaces, blank lines, quotes, and line breaks. It never changes what the code does. The best-known Python formatters are **black** and **ruff**'s formatter, which produce almost identical results.
- A **linter** reads your code and reports likely bugs and style problems, without changing anything unless you ask. (The name comes from an old tool called lint, which picked little bits of fluff out of programs.) **ruff** is a fast, popular linter; older ones include flake8 and pylint.

ruff does both jobs in one tool. Like the type checkers, these are third-party tools that you install with pip, ideally into the project's virtual environment, and run in the terminal:

```
python -m pip install ruff

ruff format .
ruff format --check .
ruff check .
ruff check --fix .
```

| Command                            | Does                                                                     |
|------------------------------------|--------------------------------------------------------------------------|
| <code>ruff format .</code>         | Reformats every <code>.py</code> file in this folder and below, in place |
| <code>ruff format --check .</code> | Changes nothing; lists the files that formatting would change            |
| <code>ruff check .</code>          | Lints every <code>.py</code> file and reports what it finds              |
| <code>ruff check --fix .</code>    | Lints, and fixes the problems that have a safe automatic fix             |

If you prefer black, `python -m pip install black` installs it and `black .` reformats every file (`black --check .` only reports). Black is only a formatter, so pair it with a linter.

### WORKED EXAMPLE A formatter at work

Here is what `ruff format` (or `black`) does to `messy_grades.py` from 19.5, saved under a new name so you can compare the two:

## formatted\_grades.py

```
1 MaxScore = 100
2
3
4 def Percent(Score, outOf=MaxScore):
5     return round(Score / outOf * 100, 1)
6
7
8 class student_record:
9     def __init__(self, name, score):
10         self.name = name
11         self.score = score
12
13     def summary(self):
14         return f"{self.name}: {Percent(self.score)}%"
15
16
17 print(student_record("Ana", 87).summary())
```

```
Ana: 87.0%
```

Every spacing problem is fixed: spaces around `=` and operators, after commas, none inside brackets, two blank lines between top-level definitions and one between methods, and double quotes. But the names are untouched: it still says `MaxScore`, `Percent`, and `student_record`. Renaming changes what the code means (every use of a name has to change with it), and a formatter only touches layout. The program prints exactly what it printed before. Most editors can run a formatter every time you save a file, so you never have to think about spacing again.

**WORKED EXAMPLE** A linter at work

This program runs and prints the right thing:

## lint\_me.py

```
1 import json
2
3
4 def find_task(tasks, title):
5     """Return the task with this title, or None if there is none."""
6     count = 0
7     for task in tasks:
8         if task["title"] == title:
9             return task
10    return None
11
12
13 tasks = [{"title": "Practice piano", "done": False}]
14 if find_task(tasks, "Swim") == None:
15     print("No task called Swim")
```

```
No task called Swim
```

`ruff check lint_me.py` reports two problems anyway. This is **sample output** in ruff's short layout (the one `ruff check --output-format concise lint_me.py` prints); by default ruff shows each problem with the lines of code around it, and the exact layout differs between versions.

```
lint_me.py:1:8: F401 [*] `json` imported but unused
lint_me.py:6:5: F841 Local variable `count` is assigned to but never used
Found 2 errors.
[*] 1 fixable with `--fix` (1 unsafe fix available with `--fix --unsafe-fixes`).
```

Each line gives the file, the line, the column (the character position along the line), a **rule code**, and a message; `[*]` marks a problem that `--fix` can fix. Codes starting with `F` are about likely bugs; codes starting with `E` are about PEP 8 style.

Line 14's `== None` breaks a PEP 8 rule too, but ruff's default rules leave out most `E` checks.

`ruff check --select E,F lint_me.py` asks for all the `E` and `F` rules, and adds a third problem:

```
lint_me.py:14:32: E711 Comparison to `None` should be `cond is None`
```

| Code | Problem                                                                                                                                    | Fix                                               |
|------|--------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------|
| F401 | <code>json</code> is imported but never used: the file depends on something it does not need                                               | Delete the import                                 |
| F841 | <code>count</code> is set but never used: often a sign of an unfinished idea or a misspelled name                                          | Delete it, or use it                              |
| E711 | <code>== None</code> can be fooled by a class whose <code>__eq__</code> (Day 17) says it equals <code>None</code> ; <code>is</code> cannot | <code>if find_task(tasks, "Swim") is None:</code> |

`ruff check --fix` deletes the unused import by itself. The other two need a person to decide, because the tool cannot know whether `count` was meant to be used. Out of the box, ruff turns on only rules that are almost always right; it can check many more, including PEP 8 naming and most `E` rules, when a project asks for them.

| Tool                | What it looks at                    | Changes your code?           | Example                                       |
|---------------------|-------------------------------------|------------------------------|-----------------------------------------------|
| Formatter           | Layout: spaces, blank lines, quotes | Yes, layout only             | <code>ruff format</code> , <code>black</code> |
| Linter              | Likely bugs and style rules         | Only with <code>--fix</code> | <code>ruff check</code>                       |
| Type checker (19.4) | Values against type hints           | No                           | <code>mypy</code> , <code>pyright</code>      |

### KEY IDEA

A formatter, such as `ruff format` or `black`, rewrites the layout of your code to the standard style without changing what it does. A linter, such as `ruff check`, reports likely bugs and style problems by rule code, and can fix some of them. Neither replaces the other, or a type checker.

### WATCH OUT

**Thinking a clean format means correct code.** A formatter fixed every space in `messy_grades.py` and left every bad name in place; it would also leave a wrong formula, a

swapped argument, or a missing `None` check exactly as they were. Formatting makes code readable; linting and type checking find some bugs; none of them can tell whether the program computes the right answer. Use all three, and still test your code (Day 20).

#### PAUSE AND TRY

##### Question 13 Multiple choice

You run `ruff format .` on a project. Which change can it make?

- A. Renaming the function `CalcTotal` to `calc_total` everywhere it is used
- B. Deleting an import that is never used
- C. Changing `total=price*qty` to `total = price * qty`
- D. Changing `-> int` to `-> str` on a function that returns a string

##### Question 14 Short answer

A teammate's project has no formatter or linter. Give two reasons to add both, and say why both are needed rather than just one.

## 19.8 Docstrings and Comments That Help

Type hints now say what **type** each value is. A reader can still have questions that no hint answers. Is the price in dollars or cents? Can `people` be 0? What happens if it is? Does the result include the tip? Docstrings (Day 9) and comments (Day 1) are where you answer those questions, and the best ones answer exactly what the code cannot say for itself.

### Docstrings

A docstring is for someone **using** the function. Day 9 gave it a one-line summary; a useful docstring follows a few more conventions from **PEP 257**, the PEP about docstrings:

- The first line is a one-sentence summary that ends with a period and is written as a command: "Return the share...", not "This function returns the share...".
- For a function whose parameters need explaining, a blank line follows, then the details: what each parameter means (units, allowed values), what is returned, including any `None` case, and which exceptions it raises (Day 15).
- It does not repeat the hints. `total: float` already says it is a float; the docstring says it is the whole bill, in dollars, with the tip included.

**WORKED EXAMPLE** A docstring that answers the caller's questions

split\_bill\_doc.py

```

1 def split_bill(total: float, people: int = 2) -> float:
2     """Return each person's share of a bill, rounded to cents.
3
4     Args:
5         total: The whole bill in dollars, tip included.
6         people: How many people share it. Must be at least 1.
7
8     Returns:
9         One person's share, rounded to 2 decimal places.
10
11     Raises:
12         ValueError: If people is less than 1.
13     """
14     if people < 1:
15         raise ValueError(f"people must be at least 1, not {people}")
16     return round(total / people, 2)
17
18
19 print(split_bill(85.0, 3))
20 help(split_bill)

```

28.33

Help on function split\_bill in module \_\_main\_\_:

```

split_bill(total: float, people: int = 2) -> float
    Return each person's share of a bill, rounded to cents.

    Args:
        total: The whole bill in dollars, tip included.
        people: How many people share it. Must be at least 1.

    Returns:
        One person's share, rounded to 2 decimal places.

    Raises:
        ValueError: If people is less than 1.

```

The layout with `Args:`, `Returns:`, and `Raises:` headings is a common convention (often called Google style); many tools and editors recognize it. Together with the hints in the signature, `help()` now tells a caller everything: what to pass, what each value means, what comes back, and what can go wrong, without reading lines 14 to 16.  $85.0 / 3$  is 28.333..., rounded to 28.33.

A one-line docstring is enough when the name, the hints, and one sentence say it all, as in most of today's examples. Write the longer form when a caller would otherwise have to read the body.

**Comments**

A comment is for someone **reading the code**, usually someone about to change it. The code already says **what** it does; a good comment says **why**: a reason, a rule from outside the program, or a trap the next person might fall into.

## good\_comments.py

```

1 TAX_RATE = 0.0825 # 6.25% state tax plus 2% city tax.
2
3
4 def price_with_tax(price: float) -> float:
5     """Return price with sales tax added, rounded to cents."""
6     # Round once, at the end: rounding the tax and the price separately
7     # can leave the total a cent out.
8     return round(price * (1 + TAX_RATE), 2)
9
10
11 def task_title(tasks: list[str], number: int) -> str:
12     """Return the title of task number, counting from 1."""
13     # People number tasks from 1, but list indexes start at 0.
14     return tasks[number - 1]
15
16
17 print(price_with_tax(19.99))
18 print(task_title(["Practice piano", "Water plants"], 2))

```

21.64

Water plants

Each comment here tells the reader something the code cannot. Line 1's comment explains where the odd number 0.0825 comes from. PEP 8 puts an **inline comment** (one on the same line as code) at least two spaces after the code, with one space after the `#`. Lines 6 and 7 are a **block comment**, on their own lines above the code they explain, indented to match it; they warn the next person not to "improve" the code by rounding twice. Line 13 explains the `- 1` that would otherwise look like a bug. Compare comments that only repeat the code:

```

1 count = count + 1 # add 1 to count
2 for task in tasks: # loop over the tasks

```

These cost reading time and tell the reader nothing new. Leave them out. If a line needs a comment to say **what** it does, first try a clearer name: `seconds_per_day = 86400` needs no comment, where `n = 86400 # seconds in a day` does.

**KEY IDEA**

A docstring tells a caller what the function does, what each parameter means beyond its type, what it returns, and what it raises, starting with a one-line command such as "Return ...". A comment tells a reader **why** the code is the way it is; never use it to repeat what the code already says.

**WATCH OUT**

**Leaving a comment that no longer matches the code.** Someone changes the code and forgets the comment:

```
1 TAX_RATE = 0.0825 # 6% state tax.
```

Now the reader has two answers and no way to know which is true. PEP 8 puts it plainly: "Comments that contradict the code are worse than no comments." Whenever you change a line, update or delete the comments and docstring that describe it. The same goes for **commented-out code** (Day 1) left in a file after it stopped being useful: delete it, because the next reader cannot tell whether it is supposed to come back.

### PAUSE AND TRY

#### Question 15 Multiple choice

Which comment is the most useful?

- A. `index = number - 1 # subtract 1 from number`
- B. `index = number - 1 # people count tasks from 1, lists from 0`
- C. `index = number - 1 # this line works out the index`
- D. `index = number - 1 # TODO: check this`

#### Question 16 Short answer

Improve this function's docstring. Say what the hints already tell the reader, and what the docstring still needs to add.

```
1 def pace(distance: float, minutes: float) -> float:
2     """This function takes a float and a float and returns a float."""
3     return minutes / distance
```

## Putting it together

A reader keeps a log of the books they finish, with an optional rating from 1 to 5. The program below uses every idea from today: hints on every function, a field and a return value that might be `None`, nested collection hints, PEP 8 names and layout, and docstrings and a comment that answer the reader's questions.

reading\_log.py

```
1 """Keep a log of finished books and report on a reader's progress."""
2
3 from dataclasses import dataclass
4
5 PAGE_GOAL = 1000
6
7
8 @dataclass
9 class Book:
10     """One finished book. rating is None until the reader rates it."""
11
12     title: str
```

```
13     pages: int
14     rating: int | None = None
15
16
17 def total_pages(books: list[Book]) -> int:
18     """Return the number of pages read across all books."""
19     return sum(book.pages for book in books)
20
21
22 def average_rating(books: list[Book]) -> float | None:
23     """Return the average rating of the rated books.
24
25     Returns None when no book has a rating yet, because an average of
26     no numbers has no sensible value (and would divide by zero).
27     """
28     ratings = [book.rating for book in books if book.rating is not None]
29     if not ratings:
30         return None
31     return sum(ratings) / len(ratings)
32
33
34 def titles_by_rating(books: list[Book]) -> dict[int, list[str]]:
35     """Group the titles of rated books by their rating."""
36     groups: dict[int, list[str]] = {}
37     for book in books:
38         if book.rating is None:
39             continue
40         if book.rating not in groups:
41             groups[book.rating] = []
42         groups[book.rating].append(book.title)
43     return groups
44
45
46 def report(books: list[Book]) -> None:
47     """Print a short progress report for a list of finished books."""
48     pages = total_pages(books)
49     print(f"Pages read: {pages} of {PAGE_GOAL}")
50     average = average_rating(books)
51     if average is None:
52         print("No ratings yet")
53     else:
54         print(f"Average rating: {average:.1f}")
55     groups = titles_by_rating(books)
56     for rating in sorted(groups, reverse=True):
57         titles = groups[rating]
58         print(f"{rating} stars: {' '.join(titles)}")
59
60
61 log = [
62     Book("The Hobbit", 310, 5),
63     Book("Holes", 233, 4),
64     Book("Wonder", 316),
65     Book("Matilda", 240, 5),
66 ]
67 report(log)
```

```
68 print()
69 report([Book("Coraline", 162)])
```

```
Pages read: 1099 of 1000
Average rating: 4.7
5 stars: The Hobbit, Matilda
4 stars: Holes
```

```
Pages read: 162 of 1000
No ratings yet
```

| Lines | What to notice                                                                                                                                           | Idea       |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------|------------|
| 1     | A module docstring says what the whole file is for                                                                                                       | 19.8       |
| 3–5   | Imports at the top, then a constant in UPPER_CASE                                                                                                        | 19.5, 19.6 |
| 8–14  | A data class whose rating field is <code>int   None</code> , defaulting to <code>None</code> ; the class docstring explains what <code>None</code> means | 19.3       |
| 17    | <code>list[Book]</code> takes a list of objects of your own class                                                                                        | 19.2       |
| 22–31 | <code>float   None</code> : the docstring explains <b>why</b> the function can return <code>None</code>                                                  | 19.3, 19.8 |
| 34    | A nested return hint, <code>dict[int, list[str]]</code> : from a rating to a list of titles                                                              | 19.2       |
| 36    | A variable annotation for a dictionary that starts empty                                                                                                 | 19.2       |
| 50–54 | The caller checks <code>is None</code> before formatting the average with <code>:.1f</code>                                                              | 19.3, 19.4 |

Follow the first report. `total_pages` adds  $310 + 233 + 316 + 240 = 1099$ . `average_rating` collects only the ratings that are not `None`: 5, 4, and 5 (Wonder has no rating), so the average is  $14 / 3 = 4.67$ , shown as 4.7. `titles_by_rating` skips Wonder and builds `{5: ["The Hobbit", "Matilda"], 4: ["Holes"]}`, and `report` prints the ratings from highest to lowest. In the second report, Coraline has no rating, so `ratings` is empty and `average_rating` returns `None` instead of dividing by zero; `report` checks for it on line 51 and prints `No ratings yet`. A type checker run on this file finds nothing to report, and a formatter would leave it unchanged, because it already follows the style that formatters produce.

## Common mistakes

Today's tools report problems by message and code. Use this table to go from a report to its usual cause. Messages from the tools are samples; their wording varies by version.

| Report                                                                        | From   | Usual cause                                                   | Fix                                      |
|-------------------------------------------------------------------------------|--------|---------------------------------------------------------------|------------------------------------------|
| <code>TypeError: unsupported operand type(s) for *: 'float' and 'type'</code> | Python | <code>price=float</code> instead of <code>price: float</code> | Use a colon for a hint                   |
| <code>TypeError: ... 'NoneType' and 'int'</code>                              | Python | A maybe-None value used without a check                       | Check <code>is None</code> first         |
| <code>Incompatible return value type (got "str", expected "int")</code>       | mypy   | The function returns a different type from its hint           | Fix the code, or the hint if it is wrong |

|                                                       |            |                                                    |                                    |
|-------------------------------------------------------|------------|----------------------------------------------------|------------------------------------|
| Argument 1 to "label" has incompatible type ...       | mypy       | Wrong argument, often arguments in the wrong order | Pass the right value               |
| Item "None" of "Task   None" has no attribute ...     | mypy       | A maybe-None value used without a check            | Check is None first                |
| TypeError: 'list' object is not callable              | Python     | A variable named after a built-in                  | Rename the variable                |
| SyntaxError or IndentationError on a split line       | Python     | A line broken outside brackets                     | Wrap the expression in parentheses |
| F401 ... imported but unused                          | ruff check | An import nothing uses                             | Delete it (--fix can)              |
| F841 Local variable ... is assigned to but never used | ruff check | A leftover or misspelled variable                  | Delete it, or use it               |
| E711 Comparison to None ...                           | ruff check | == None (reported with --select E)                 | is None                            |

Older versions of Python may word the Python messages differently.

## Vocabulary

| Term                       | Definition                                                                                   | Example                          |
|----------------------------|----------------------------------------------------------------------------------------------|----------------------------------|
| <b>type annotation</b>     | A note on a parameter, return value, or variable, written after a colon or after ->          | price: float                     |
| <b>type hint</b>           | An annotation that states the type a value should have; Python does not check it at run time | def area(width: float) -> float: |
| -> None                    | The return hint for a function that returns nothing useful                                   | def show_menu() -> None:         |
| <b>variable annotation</b> | A type hint on a variable, written before its first value                                    | counts: dict[str, int] = {}      |
| <b>union type</b>          | A hint meaning "one of these types", written with                                            | int   str                        |
| X   None                   | A value of type X, or None (older code: Optional[X])                                         | -> Task   None                   |
| <b>typing</b>              | The standard library module of extra hint tools, such as Optional and List                   | from typing import Optional      |
| <b>type checker</b>        | A tool that reads code without running it and reports values that do not match the hints     | mypy checkout.py                 |

|                                   |                                                                                       |                                                   |
|-----------------------------------|---------------------------------------------------------------------------------------|---------------------------------------------------|
| <b>static analysis</b>            | Checking code by reading it, without running it                                       | type checking, linting                            |
| <b>PEP</b>                        | A Python Enhancement Proposal: a numbered document proposing a change or a convention | PEP 8, PEP 257                                    |
| <b>PEP 8</b>                      | Python's official style guide                                                         | <code>snake_case</code> functions                 |
| <b>implicit line continuation</b> | Continuing a statement on the next line inside an open bracket                        | <code>return ( ... )</code>                       |
| <b>formatter</b>                  | A tool that rewrites code's layout to a standard style without changing what it does  | <code>ruff format .</code> , <code>black .</code> |
| <b>linter</b>                     | A tool that reports likely bugs and style problems                                    | <code>ruff check .</code>                         |
| <b>rule code</b>                  | A linter's short name for one rule                                                    | F401                                              |
| <b>PEP 257</b>                    | The conventions for writing docstrings                                                | "Return ..." summaries                            |
| <b>inline comment</b>             | A comment on the same line as code, two spaces after it                               | <code>x = 1 # why</code>                          |
| <b>block comment</b>              | A comment on its own lines, indented like the code it explains                        | <code># Round once, at the end: ...</code>        |

## Summary

- A **type hint** says what type a value should have: `name: type` on parameters and variables, `-> type` for the return value, with `-> None` when nothing is returned. Python stores hints but never checks them when the program runs.
- Collection hints name the inner types: `list[int]`, `set[str]`, `dict[str, int]` (keys, then values), `tuple[str, int]` (one type per position), nested as deep as the data. `X | None` means "an X, or None", and the caller should check `is None` before using the value.
- A **type checker** such as mypy or pyright reads hinted code without running it and reports mismatches, with the line, the type found, and the type expected, so it finds bugs in lines that have not run yet.
- **PEP 8** is the shared style guide: `snake_case` for functions and variables, `CapWords` for classes, `UPPER_CASE` for constants, single spaces around operators, 79-character lines broken inside brackets, and grouped imports at the top.
- A **formatter** (`ruff format`, `black`) applies the layout rules for you; a **linter** (`ruff check`) reports likely bugs and style problems. A docstring tells a caller what a function does, means, returns, and raises; a comment tells a reader **why**.

## Practice

Do these in your `day19` folder. Type each program yourself, run it, and only then open the answer. Questions 18 and 19 use `ruff` and `mypy` (or `pyright`); install them first, as shown in 19.7 and 19.4.

**Question 17** Short answer

**A typed to-do manager.** Copy your to-do manager from Day 17 (`main.py`, `task.py`, `tasks.py`, and `tasks.json`) into a `todo_app` folder in `day19`. Add type hints to every function and method: every parameter except `self`, and every return value. Then run `main.py` and check that it behaves exactly as before.

**Question 18** Short answer

**Lint and format.** Run a linter and a formatter on your project and fix what they report. If your own code comes back clean, practice on this quick to-do script, written in a hurry: save it as `todo_messy.py`, run `ruff check todo_messy.py`, fix what it reports, then run `ruff format todo_messy.py`, and finally fix anything else PEP 8 asks for that neither tool changed.

`todo_messy.py`

```
1 import json, os
2 def addTask(tasks,title):
3     tasks.append({"title":title,"done":False})
4 def countDone(tasks):
5     total=0
6     done = 0
7     for t in tasks:
8         if t['done']==True: done+=1
9     return done
10 tasks=[]
11 addTask(tasks,'Practice piano')
12 addTask(tasks,'Water plants')
13 tasks[0]['done']=True
14 print(countDone(tasks),'of',len(tasks),'done')
```

**Question 19** Short answer

**Resolve type checker complaints.** Run a type checker on your project and resolve at least three complaints. If your typed `todo_app` passes cleanly, practice on this file, which has three type mistakes: save it as `todo_check.py`, run `mypy todo_check.py` (or `pyright todo_check.py`), and fix each complaint.

`todo_check.py`

```
1 from dataclasses import dataclass
2
3
4 @dataclass
5 class Task:
6     title: str
7     done: bool = False
8
9
10 def add_task(tasks: list[Task], title: str) -> None:
11     tasks.append(Task(title))
12
13
14 def complete_task(tasks: list[Task], number: int) -> Task:
15     if not 1 <= number <= len(tasks):
16         return None
```

```

17     task = tasks[number - 1]
18     task.done = True
19     return task
20
21
22 def find_task(tasks: list[Task], title: str) -> Task | None:
23     for task in tasks:
24         if task.title == title:
25             return task
26     return None
27
28
29 tasks: list[Task] = []
30 add_task(tasks, "Practice piano")
31 add_task(tasks, Task("Water plants"))
32 complete_task(tasks, 1)
33 print(find_task(tasks, "Practice piano").done)
34 print(tasks[1])

```

## Review questions

### Question 20 Multiple choice

A function `best_score(scores)` takes a dictionary from student names to lists of test scores (whole numbers), and returns the name and score of the highest single score, or `None` if the dictionary is empty. Which `def` line has the right hints?

- A. `def best_score(scores: dict[str, list[int]]) -> tuple[str, int] | None:`
- B. `def best_score(scores: dict[str, int]) -> tuple[str, int]:`
- C. `def best_score(scores: dict[str, list[int]]) -> tuple[str, int]:`
- D. `def best_score(scores: list[int]) -> str | int | None:`

### Question 21 Debugging

This program should print each class's average, or `no scores` for a class with none. It prints the first average and then crashes. Explain the error, say which tool could have warned you before running it, and fix it.

`review_average.py`

```

1 def average(scores: list[int]) -> float | None:
2     """Return the average of scores, or None if there are no scores."""
3     if not scores:
4         return None
5     return sum(scores) / len(scores)
6
7
8 for scores in [[80, 95, 90], []]:
9     print(round(average(scores), 1))

```

### Question 22 Tracing

What does this program print? One line breaks a hint; decide what Python does with it.

review\_trace.py

```

1  from dataclasses import dataclass
2
3
4  @dataclass
5  class Player:
6      name: str
7      points: int = 0
8      team: str | None = None
9
10     def label(self) -> str:
11         if self.team is None:
12             return f"{self.name} ({self.points})"
13         return f"{self.name} of {self.team} ({self.points})"
14
15
16 def leader(players: list[Player]) -> Player | None:
17     best: Player | None = None
18     for player in players:
19         if best is None or player.points > best.points:
20             best = player
21     return best
22
23
24 players = [Player("Ana", 12, "Owls"), Player("Ben", 15), Player("Cy", "9")]
25 print(players[2].points * 2)
26 top = leader(players[:2])
27 if top is not None:
28     print(top.label())
29 print(leader([]))

```

**Question 23** Short answer

For each problem, say which tool would find or fix it: a formatter, a linter, a type checker, or none of them. (a) `x=y+1` with no spaces; (b) `import os` in a file that never uses `os`; (c) calling `label(3, "pens")` when the hints are `label(item: str, quantity: int)`; (d) a function that adds a 5% discount when the shop's rule is 10%.

**Question 24** Multiple choice

Which `def` line follows PEP 8 and has correct hints?

- A. `def pay(hours:float, rate:float=15.0)->float:`
- B. `def pay(hours: float, rate: float = 15.0) -> float:`
- C. `def pay(hours: float, rate: float=15.0) -> float:`
- D. `def Pay(hours: float, rate: float = 15.0) -> float:`

**Question 25** Short answer

A new teammate will use your `load_tasks(path: Path) -> list[Task]` function without reading its body. Using today's ideas, list what the signature already tells them, and what the docstring must still tell them for them to use it without asking you any questions.

## Go further

PEP 8 itself is worth skimming: PEP 8: Style Guide for Python Code explains the reason behind each rule, with many short examples. The typing module documentation introduces the hint tools beyond today's, for when you meet them in other people's code.

## Reflection

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

- ☐ I can add type hints to parameters and return values, including `→ None`, and explain that Python does not check them. Reread: 19.1 Annotating Parameters and Return Values
- ☐ I can write hints such as `list[int]`, `dict[str, int]`, and `tuple[str, int]`, and annotate a variable that starts empty. Reread: 19.2 Hints for Lists, Dictionaries, and Tuples
- ☐ I can use `X | None` for a value that might be `None`, and check it with `is None` before using it. Reread: 19.3 Values That Might Be None
- ☐ I can run `mypy` or `pyright`, read what it reports, and fix the code or the hint. Reread: 19.4 Checking Types with `mypy` or `pyright`
- ☐ I can name variables, functions, classes, and constants the PEP 8 way, and space code the PEP 8 way. Reread: 19.5 PEP 8: Naming and Whitespace
- ☐ I can break a long line inside brackets, and arrange imports in PEP 8 order. Reread: 19.6 PEP 8: Line Length and Imports
- ☐ I can run a formatter and a linter, and explain what each one does and does not change. Reread: 19.7 Formatting and Linting with `ruff` or `black`
- ☐ I can write a docstring that tells a caller what they need, and comments that explain why. Reread: 19.8 Docstrings and Comments That Help

## Check Your Reasoning

Answers and explanations for every question, numbered to match.

### Question 1

**A.** Each parameter is followed by a colon and its type, and the return type follows `->` before the final colon.

- **B** is wrong: `=` sets default values, so `name` would default to the class `str` itself. Nothing says what type the arguments should be.
- **C** is wrong: the return hint must come before the colon that ends the `def` line. Placed after it, `-> str` is a syntax error.
- **D** is wrong: the order is backwards. The parameter's name comes first, then the colon, then the type.

Reread *19.1 Annotating Parameters and Return Values* if you chose differently.

### Question 2

```
hahaha
6
3.5
```

Python ignores the hints at run time, so each line does whatever the values themselves do. Line 9 repeats "ha" 3 times. Line 10 passes the int 3 as `word`, so `3 * 2` is ordinary multiplication: 6, even though the hints say `word` is a string and the result is a string. Line 11: `/` always gives a float (Day 2), so `half(7)` returns 3.5, even though the return hint says `int`. If you expected an error, remember that only a type checker would report these mistakes (mypy points to line 10, and to line 6, where `half` returns a float); running the program does not.

Reread *19.1 Annotating Parameters and Return Values* if your output differed.

### Question 3

**C.** The keys are strings (names) and each value is a list of ints, and a dictionary hint lists the key type first, then the value type.

- **A** is wrong: the values are lists, not single ints.
- **B** is wrong: it has the key and value types the wrong way round. (A list could not even be a key, because keys must be hashable, Day 8.)
- **D** is wrong: the value is a single dictionary, not a list of dictionaries.

Reread *19.2 Hints for Lists, Dictionaries, and Tuples* if you chose differently.

### Question 4

Model answer:

```
1 def cheapest(prices: dict[str, float]) -> str:
2 def split_name(full_name: str) -> tuple[str, str]:
3 def grid(rows: int, columns: int) -> list[list[int]]:
```

(These are only the first lines; each function still needs a body.) For (a), the keys are names (strings) and the values are prices (floats), and the function returns one name. For (b), a pair of strings is a tuple with two positions, so it needs one type for each. For (c), the result is a list whose items are lists of ints, which is a nested hint. A correct answer has the parameter hints with colons, the return hint after `->`, and the inner types in square brackets.

Reread 19.2 *Hints for Lists, Dictionaries, and Tuples* if your reasoning differed.

### Question 5

```
4 PLUM
6 BANANA
7 none
```

| Length | Words checked                               | Returns  | Printed  |
|--------|---------------------------------------------|----------|----------|
| 4      | fig (3, too short), plum (4, long enough)   | "plum"   | 4 PLUM   |
| 6      | fig, plum, banana (6)                       | "banana" | 6 BANANA |
| 7      | all three are shorter than 7; the loop ends | None     | 7 none   |

The return hint `str | None` matches both endings: a word from line 4, or `None` from line 5. Line 11 checks for `None` before line 14 calls `upper()`, so the third case prints `none` instead of crashing.

Reread 19.3 *Values That Might Be None* if your output differed.

### Question 6

C. The function returns one of two things, a `Task` or `None`, and `Task | None` says exactly that.

- **A** is wrong: it promises a `Task` every time, so a caller would not know to check for `None`.
- **B** is wrong: `-> None` means the function never returns a useful value, but it does return the deleted task.
- **D** is wrong: the function returns one task, not the list.

Reread 19.3 *Values That Might Be None* if you chose differently.

### Question 7

B. `describe` expects a string, then a float. The second `print` passes a float (the result of `area`) as `name` and a string as `size`, so both arguments are reported.

- **A** is wrong: an int is accepted where a float is expected, because a whole number is a perfectly good width.
- **C** is wrong: that return is in `describe`, whose hint is `-> str`, and an f-string is a string.
- **D** is wrong: a type checker does not run the program. Python runs this program without an error, but its second line of output is `10.0: hall square meters`, which is nonsense, and that is exactly the kind of bug a checker catches.

Reread 19.4 *Checking Types with mypy or pyright* if you chose differently.

### Question 8

Model answer:

1. Running the program only tests the lines and values that ran this time. A type checker reads **every** line, including branches that rarely run (such as the "not found" case), so it finds bugs that are waiting for an unusual input.

2. A program can print the right thing today with a wrong type, as `checkout.py`'s `total_items` did, and then crash later when other code relies on the hint (for example, adding 1 to a result that is really a string).

Other good reasons: the checker finds mistakes in seconds, before anyone runs the program, and it keeps the hints honest, so readers can trust them. A correct answer mentions that the checker examines code that did not run, or catches wrong types that have not caused a visible problem yet.

Reread *19.4 Checking Types with mypy or pyright* if your reasoning differed.

### Question 9

**B.** A constant, a value meant never to change, is written in capitals with underscores.

- **A** is wrong: class names use CapWords, so it should be `class BankAccount:`.
- **C** is wrong: function names use snake\_case, so it should be `def calculate_total(prices):`. A CapWords name makes a reader think `CalculateTotal(...)` creates an object.
- **D** is wrong: a variable that changes is an ordinary variable, so it should be `tax_rate`. CapWords would make it look like a class.

Reread *19.5 PEP 8: Naming and Whitespace* if you chose differently.

### Question 10

Model answer. Problems:

- `AddBonus` is a function, so it should be snake\_case: `add_bonus`.
- There are spaces just inside the parentheses, and none after the comma.
- `bonus = 10` is a default with no hint, so it takes no spaces around `=`.
- `Total` is a variable, so it should be snake\_case: `total`.
- `Total=points+bonus` needs spaces around `=` and `+`.

```
1 def add_bonus(points, bonus=10):
2     total = points + bonus
3     return total
```

A correct answer finds the two naming problems (function and variable) and at least three of the whitespace problems.

Reread *19.5 PEP 8: Naming and Whitespace* if your answer differed.

### Question 11

**B.** Standard library first, then third-party, then your own modules, with a blank line between the groups and one module per line.

- **A** is wrong: it puts two modules on one line, and mixes a standard library module with a third-party one.
- **C** is wrong: the groups are in reverse order. The standard library comes first.
- **D** is wrong: imports go at the top of the file, where a reader can see everything the file needs.

Reread *19.6 PEP 8: Line Length and Imports* if you chose differently.

### Question 12

Python stops before running anything:

```
IndentationError: unexpected indent
```

Line 3 is a complete statement on its own (`return average >= 90 and attendance >= 0.95`), and there is no open bracket, so Python treats the end of line 3 as the end of the statement. Line 4 then looks like a new statement that is indented for no reason. (Older versions of Python may word the message differently.) Wrap the whole condition in parentheses, with one test per line and each and at the start:

`honor_roll_fixed.py`

```
1 def on_honor_roll(average: float, attendance: float, late_days: int) -> bool:
2     """Return True if a student qualifies for the honor roll."""
3     return (
4         average >= 90
5         and attendance >= 0.95
6         and late_days <= 3
7     )
8
9
10 print(on_honor_roll(93.5, 0.97, 1))
11 print(on_honor_roll(93.5, 0.97, 5))
```

```
True
False
```

The first student meets all three rules. The second has 5 late days, so `late_days <= 3` is `False` and so is the whole `and` chain. Notice that the broken version, had it run, would have ignored the late-days rule completely.

Reread *19.6 PEP 8: Line Length and Imports* if you missed it.

### Question 13

**C.** Adding spaces is a layout change, and layout is all a formatter changes.

- **A** is wrong: renaming changes what the names mean, and a formatter never touches names. PEP 8 naming problems need a person (or a linter with naming rules turned on, to report them).
- **B** is wrong: deleting an unused import is a linter's fix (`ruff check --fix`), not a formatter's.
- **D** is wrong: no tool here rewrites your hints. A type checker would **report** the mismatch, and you decide whether to fix the hint or the code.

Reread *19.7 Formatting and Linting with ruff or black* if you chose differently.

### Question 14

Model answer: a formatter means nobody spends time on spacing or argues about it in a review, and every file looks the same, so readers can concentrate on what the code does. A linter catches likely bugs that still run, such as an unused import or a variable that is set and never used, and style problems such as `== None`. Both are needed because they do different jobs: the formatter changes layout but never reports bugs or renames anything, and the linter reports problems but (apart from a few safe fixes) does not reformat code.

A correct answer gives one benefit of each tool and explains that the formatter handles layout while the linter finds problems.

Reread *19.7 Formatting and Linting with ruff or black* if your reasoning differed.

### Question 15

**B.** It explains **why** the code subtracts 1, which is the one thing the code cannot say for itself, and it stops a future reader from "fixing" what looks like a mistake.

- **A** is wrong: it repeats exactly what the code says.
- **C** is wrong: it also repeats the code, in vaguer words; the name `index` already says that.
- **D** is wrong: it raises a doubt without saying what to check or why, so the next reader knows less than before.

Reread *19.8 Docstrings and Comments That Help* if you chose differently.

### Question 16

Model answer: the hints already say that both parameters and the result are floats, so the docstring repeats them and says nothing else. It should say what the function computes, which units it uses, and what happens with a distance of 0. For example:

```
1 def pace(distance: float, minutes: float) -> float:
2     """Return a running pace in minutes per kilometer.
3
4     Args:
5         distance: How far the run was, in kilometers. Must not be 0.
6         minutes: How long the run took, in minutes.
7
8     Returns:
9         The minutes taken for each kilometer.
10    """
11    return minutes / distance
```

A correct answer starts with a one-line command ("Return ..."), names the units, and does not just repeat the types. Mentioning that a distance of 0 would cause a `ZeroDivisionError` (Day 15) is a strong addition.

Reread *19.8 Docstrings and Comments That Help* if your answer differed.

### Question 17

One possible solution. `task.py`:

`todo_app/task.py`

```
1 """The Task class for the to-do manager, written as a data class."""
2
3 from dataclasses import asdict, dataclass
4
5
6 @dataclass
7 class Task:
8     """One thing to do: a title, and whether it is done yet."""
9
10    title: str
11    done: bool = False
12
13    def complete(self) -> None:
14        """Mark this task as done."""
```

```

15         self.done = True
16
17     def to_dict(self) -> dict[str, str | bool]:
18         """Return this task as a dictionary, ready to save as JSON."""
19         return asdict(self)
20
21     def __str__(self) -> str:
22         mark = "x" if self.done else " "
23         return f"[{mark}] {self.title}"
24
25
26 if __name__ == "__main__":
27     task = Task("Test the class")
28     print(repr(task))
29     task.complete()
30     print(task)
31     print(task.to_dict())
32     print(task == Task("Test the class", True))

```

```

Task(title='Test the class', done=False)
[x] Test the class
{'title': 'Test the class', 'done': True}
True

```

tasks.py:

todo\_app/tasks.py

```

1  """Task functions for the to-do manager. main.py imports them."""
2
3  import json
4  from pathlib import Path
5
6  from task import Task
7
8
9  def show_menu() -> None:
10     """Print the menu of choices."""
11     print()
12     print("1) Add  2) List  3) Complete  4) Delete  5) Quit")
13
14
15  def add_task(tasks: list[Task], title: str) -> None:
16     """Add a new, unfinished Task to the end of tasks."""
17     tasks.append(Task(title))
18
19
20  def list_tasks(tasks: list[Task]) -> None:
21     """Print every task with its number, using Task.__str__."""
22     if not tasks:
23         print("No tasks yet.")
24         return
25     for number, task in enumerate(tasks, 1):
26         print(f"{number}. {task}")
27

```

```
28
29 def complete_task(tasks: list[Task], number: int) -> Task | None:
30     """Mark task number (counting from 1) as done.
31
32     Return the Task, or None if no task has that number.
33     """
34     if not 1 <= number <= len(tasks):
35         return None
36     task = tasks[number - 1]
37     task.complete()
38     return task
39
40
41 def delete_task(tasks: list[Task], number: int) -> Task | None:
42     """Remove task number (counting from 1) from tasks.
43
44     Return the removed Task, or None if no task has that number.
45     """
46     if not 1 <= number <= len(tasks):
47         return None
48     return tasks.pop(number - 1)
49
50
51 def save_tasks(tasks: list[Task], path: Path) -> None:
52     """Write tasks to the JSON file at path, replacing what was there."""
53     data = [task.to_dict() for task in tasks]
54     with path.open("w", encoding="utf-8") as file:
55         json.dump(data, file, indent=2)
56
57
58 def load_tasks(path: Path) -> list[Task]:
59     """Return the Tasks saved at path, or [] if there are none to load.
60
61     A missing file means this is the first run. A file that is not valid
62     JSON gets a warning, and the program starts with no tasks.
63     """
64     try:
65         with path.open(encoding="utf-8") as file:
66             data = json.load(file)
67     except FileNotFoundError:
68         return []
69     except json.JSONDecodeError:
70         print(f"Warning: {path.name} is not valid JSON, so no tasks loaded.")
71         return []
72     return [Task(**item) for item in data]
73
74
75 if __name__ == "__main__":
76     test_tasks: list[Task] = []
77     add_task(test_tasks, "Test the module")
78     task = complete_task(test_tasks, 1)
79     print(task is not None and task.done)
80     print(complete_task(test_tasks, 2))
81     list_tasks(test_tasks)
```

```
True
None
1. [x] Test the module
```

In `main.py`, the only change is the first line of the function: `def main() -> None:`. The menu session is exactly Day 17's, because hints never change what a program does.

How it works:

- Functions that only print or change a list, such as `show_menu`, `add_task`, and `save_tasks`, return nothing, so their hint is `-> None`.
- `complete_task` and `delete_task` return a `Task` or `None` (19.3), and `main.py` already checks `if task is None:` before using the result.
- `tasks: list[Task]` appears on every function that takes the list (19.2). The `Task` class from `task.py` is used as a type like any built-in type.
- `save_tasks` and `load_tasks` receive a `pathlib.Path` from `main.py`, so `tasks.py` now imports `Path` (line 4) just to use it in the hints.
- `to_dict` returns a dictionary whose keys are strings and whose values are a string (the title) or a bool (done): `dict[str, str | bool]`.
- `self` gets no hint: it is always an object of the class itself.
- Line 76 annotates the empty list that the test code starts with. Lines 78 and 79 store `complete_task`'s result and check it before reading `.done`; Day 17's `complete_task(test_tasks, 1).done` would be reported by a type checker now that the hint says the result might be `None` (19.4).

Reread 19.1 *Annotating Parameters and Return Values*, 19.2 *Hints for Lists, Dictionaries, and Tuples*, and 19.3 *Values That Might Be None* if your hints differed.

### Question 18

The linter's report looks something like this (sample output in the short layout of 19.7; the layout and wording vary by version):

```
todo_messy.py:1:1: I001 [*] Import block is un-sorted or un-formatted
todo_messy.py:1:8: F401 [*] `json` imported but unused
todo_messy.py:1:14: F401 [*] `os` imported but unused
todo_messy.py:5:5: F841 Local variable `total` is assigned to but never used
Found 4 errors.
[*] 3 fixable with `--fix` (1 unsafe fix available with `--fix --unsafe-fixes`).
```

Fixes: delete the import line (nothing uses `json` or `os`, and I001 objects to two modules on one line, which also breaks 19.6's rule) and delete `total = 0`. Then `ruff format` fixes the spacing, quotes, and blank lines, and moves `done += 1` onto its own line. Neither tool touches `== True` or renames anything, so do those by hand: write `if task["done"]:`, because a bool is already true or false (Day 4), and rename `addTask` to `add_task`, `countDone` to `count_done`, and the one-letter `t` to `task`. Adding a docstring to each function finishes the job. One possible result:

todo\_tidy.py

```

1 def add_task(tasks, title):
2     """Add a new, unfinished task to the end of tasks."""
3     tasks.append({"title": title, "done": False})
4
5
6 def count_done(tasks):
7     """Return how many tasks are done."""
8     done = 0
9     for task in tasks:
10         if task["done"]:
11             done += 1
12     return done
13
14
15 tasks = []
16 add_task(tasks, "Practice piano")
17 add_task(tasks, "Water plants")
18 tasks[0]["done"] = True
19 print(count_done(tasks), "of", len(tasks), "done")

```

1 of 2 done

How it works: the output is the same before and after, because none of these fixes changes what the program does; they make it shorter, clearer, and free of the leftover imports and variable. Run `ruff check` and `ruff format --check` again at the end: both should report nothing. On your typed `todo_app` from Question 17, both tools usually find nothing, because it already follows the style they enforce.

Reread [19.5 PEP 8: Naming and Whitespace](#) and [19.7 Formatting and Linting with ruff or black](#) if your result differed.

### Question 19

The program runs, but its last line already shows something is wrong:

```

True
Task(title=Task(title='Water plants', done=False), done=False)

```

mypy reports three errors (sample output; wording varies by version, and pyright words them differently but points to the same lines):

```

todo_check.py:16: error: Incompatible return value type (got "None", expected "Task")
[return-value]
todo_check.py:31: error: Argument 2 to "add_task" has incompatible type "Task"; expected
"str" [arg-type]
todo_check.py:33: error: Item "None" of "Task | None" has no attribute "done" [union-
attr]
Found 3 errors in 1 file (checked 1 source file)

```

The fixes, one per complaint:

1. **Line 16:** `complete_task` really can return `None`, so the code is right and the **hint** is wrong. Change line 14's return hint to `-> Task | None`.
2. **Line 31:** `add_task` expects a title and makes the `Task` itself, so passing a `Task` wrapped one task inside another; that is the strange last line of output. Pass the title:  
`add_task(tasks, "Water plants")`.
3. **Line 33:** `find_task` might return `None`. Store the result and check it before reading `.done`.

`todo_check_fixed.py`

```

1  from dataclasses import dataclass
2
3
4  @dataclass
5  class Task:
6      title: str
7      done: bool = False
8
9
10 def add_task(tasks: list[Task], title: str) -> None:
11     tasks.append(Task(title))
12
13
14 def complete_task(tasks: list[Task], number: int) -> Task | None:
15     if not 1 <= number <= len(tasks):
16         return None
17     task = tasks[number - 1]
18     task.done = True
19     return task
20
21
22 def find_task(tasks: list[Task], title: str) -> Task | None:
23     for task in tasks:
24         if task.title == title:
25             return task
26     return None
27
28
29 tasks: list[Task] = []
30 add_task(tasks, "Practice piano")
31 add_task(tasks, "Water plants")
32 complete_task(tasks, 1)
33 task = find_task(tasks, "Practice piano")
34 if task is not None:
35     print(task.done)
36 print(tasks[1])

```

```

True
Task(title='Water plants', done=False)

```

How it works: running the checker again reports no issues. Each fix is a different kind: a wrong hint (1), a wrong argument (2), and a missing `None` check (3). Complaint 2 was a real bug that the output showed only if you looked closely; complaint 3 was a crash waiting for the day someone searches for a title that does not exist.

Reread *19.4 Checking Types with mypy or pyright* and *19.3 Values That Might Be None* if your result differed.

### Question 20

**A.** The parameter maps names (strings) to lists of ints, and the result is either a (name, score) pair, which is `tuple[str, int]`, or `None`, so the whole return type is `tuple[str, int] | None`.

- **B** is wrong twice: each value is a list of scores, not one int, and the hint leaves out the `None` case.
- **C** is wrong: it promises a pair every time, so callers would not know to check for `None` when the dictionary is empty.
- **D** is wrong: the parameter is a dictionary, not a list, and `str | int | None` means "a single string, int, or `None`", not a pair of a name and a score.

Reread *19.2 Hints for Lists, Dictionaries, and Tuples* and *19.3 Values That Might Be None* if you chose differently.

### Question 21

It prints `88.3` and then stops with:

```
TypeError: type NoneType doesn't define __round__ method
```

For the empty list, `average` returns `None`, exactly as its hint `float | None` says, and `round(None, 1)` fails because `None` is not a number. (Older versions of Python may word the message differently.) A type checker would have reported line 9 without running the program, because it passes a value that might be `None` to `round`, which needs a number. Check for `None` before rounding:

`review_average_fixed.py`

```
1 def average(scores: list[int]) -> float | None:
2     """Return the average of scores, or None if there are no scores."""
3     if not scores:
4         return None
5     return sum(scores) / len(scores)
6
7
8 for scores in [[80, 95, 90], []]:
9     result = average(scores)
10    if result is None:
11        print("no scores")
12    else:
13        print(round(result, 1))
```

```
88.3
no scores
```

$(80 + 95 + 90) / 3 = 88.33$ , rounded to `88.3`.

Reread *19.3 Values That Might Be None* and *19.4 Checking Types with mypy or pyright* if you missed it.

## Question 22

```
99
Ben (15)
None
```

- Line 24 makes `Player("Cy", "9")`. The field hint says `points: int`, but Python does not check hints, so `points` holds the **string** `"9"`. Line 25 doubles it with string repetition (Day 3): `"99"`, printed as 99. A type checker would report line 24.
- Line 26: `leader` looks at Ana (12) and Ben (15) only. `best` starts as `None`, the first player becomes `best`, and Ben's 15 beats 12. Ben has no team, so `label` takes the `is None` branch on line 11: `Ben (15)`.
- Line 29: with an empty list, the loop never runs, and `leader` returns `None`, which matches its hint `Player | None`.

Reread 19.1 *Annotating Parameters and Return Values* and 19.3 *Values That Might Be None* if your output differed.

## Question 23

Model answer:

- **(a) A formatter.** Spacing is layout, and `ruff format` or `black` rewrites it to `x = y + 1`.
- **(b) A linter.** `ruff check` reports F401 (imported but unused), and `--fix` can delete the line.
- **(c) A type checker.** `mypy` or `pyright` reports that the arguments do not match the hinted types.
- **(d) None of them.** The types are right and the layout is fine; only a person, or a test that checks the answer (Day 20), can know that the rule is 10%.

A correct answer matches each problem to the right tool, and recognizes that a wrong rule is invisible to all three.

Reread 19.7 *Formatting and Linting with ruff or black* and 19.4 *Checking Types with mypy or pyright* if your reasoning differed.

## Question 24

**B.** One space after each colon, spaces around `=` because `rate` has a hint, spaces around `->`, and a snake\_case function name.

- **A** is wrong: it has no spaces after the colons or around `=` and `->`. Python accepts it, but a formatter would rewrite it to B.
- **C** is wrong: without a hint, a default takes no spaces (`rate=15.0`), but **with** a hint it takes a space on each side.
- **D** is wrong: `Pay` is in CapWords, which is for classes; a function name is `pay`.

Reread 19.5 *PEP 8: Naming and Whitespace* and 19.1 *Annotating Parameters and Return Values* if you chose differently.

## Question 25

Model answer. The signature says: pass a `Path`, and you get back a list of `Task` objects. The hints say nothing about the cases a caller has to plan for, so the docstring must say:

- what the file should contain (a JSON list of tasks saved by `save_tasks`);
- what happens when the file does not exist (an empty list, because it is the first run);

- what happens when the file is not valid JSON (a warning is printed, and an empty list is returned), and that no exception escapes to the caller.

Day 17's docstring for `load_tasks` covers the second and third points, which is why `main.py` needs no `try` around the call. A correct answer separates what the hints say (types) from what only the docstring can say (meaning, special cases, and errors).

Reread *19.8 Docstrings and Comments That Help* if your reasoning differed.