

Day 16 Classes and objects, part 1

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

By the end you will be able to

- Explain why a class that bundles data with behavior beats a dictionary plus loose functions.
- Write a class with `__init__`, instance attributes, and methods, and create objects from it.
- Tell instance attributes from class attributes, and give a class readable `__str__` and `__repr__` methods.
- Predict what happens when two names refer to one object, and tell identity from equality.

What to remember from earlier

- Every value is an object in memory. `==` asks whether two values are equal, while `is` asks whether they are the very same object. ([Day 4: Making decisions, 4.8 Equality Versus Identity](#))
- A method is a function that belongs to a value, called with a dot, such as `name.upper()`. ([Day 3: Strings, 3.6 String Methods](#))
- An attribute is a value that belongs to an object, read with a dot and no parentheses, such as `file.closed`. ([Day 14: Files and data formats, 14.2 The with Statement](#))
- `b = a` makes a second name for the same list, so a change through one name shows through the other. ([Day 6: Lists, 6.9 Aliasing Versus Copying](#))
- A default value such as `items=[]` is made once and shared by every call that leaves it out. ([Day 10: Functions, part 2, 10.3 The Mutable Default Argument Trap](#))
- `raise ValueError("message")` stops a function and hands the problem to its caller. ([Day 15: Errors and exceptions, 15.7 Raising Your Own Errors with raise](#))
- `class SomethingError(Exception)` followed by `pass` makes a new exception type. ([Day 15: Errors and exceptions, 15.8 Defining a Custom Exception](#))

Make a `day16` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days.

Since Day 2 you have used values of many types: ints, strings, lists, dictionaries, files, paths. Each type came with its own methods, such as `upper()` for strings and `append()` for lists. Today you make **your own** types. A game can have a `Player` type, a library program a `Book` type, a bank program a `BankAccount` type, each holding its own data and knowing its own operations. This is the biggest new idea since functions, and it is how most large Python programs are organized. On Day 15 you copied a two-line pattern starting with the word `class` to make a custom exception; by the end of today you will know what every part of that pattern means.

16.1 Why Classes: Bundling Data with Behavior

Imagine a small game that keeps a name and a score for each player. With what you know, you would store each player as a dictionary (Day 8) and write functions that take a player dictionary as their argument (Day 9). Read this program and look for the mistake before you look at the output:

player_dicts.py

```

1 def add_points(player, points):
2     player["score"] += points
3
4
5 def describe(player):
6     name = player["name"]
7     score = player["score"]
8     return f"{name}: {score} points"
9
10
11 ana = {"name": "Ana", "score": 0}
12 ben = {"name": "Ben", "scroe": 0}
13 add_points(ana, 5)
14 print(describe(ana))
15 add_points(ben, 3)
16 print(describe(ben))

```

Ana: 5 points

Traceback (most recent call last):

File ".../day16/player_dicts.py", line 15, in <module>

add_points(ben, 3)

~~~~~^~~~~~

File ".../day16/player\_dicts.py", line 2, in add\_points

player["score"] += points

~~~~~^~~~~~

KeyError: 'score'

The mistake is on line 12: "scroe" instead of "score". Nothing complained when the dictionary was made, because a dictionary accepts any keys you give it. The crash came three lines later, inside a different function. In a real program the typo and the crash could be hundreds of lines apart. This design has two weaknesses:

1. **Nothing guarantees the shape of the data.** Every player dictionary is supposed to have a "name" and a "score", but that rule lives only in your head. Each place that makes a player can get it wrong.
2. **The data and the code that works on it are separate.** `add_points` and `describe` only make sense for player dictionaries, but nothing connects them. They sit loose in the file, and any dictionary at all can be passed to them.

Now think about a list. A list holds its items (data), and it also knows how to `append`, `sort`, and `pop` (behavior). You never pass a list to a separate `append_to_list(items, value)` function; the operation belongs to the list. Strings are the same:

```
>>> type("hello")
<class 'str'>
>>> type([3, 1, 2])
<class 'list'>
>>> word = "hello"
>>> word.upper()
'HELLO'
>>> scores = [3, 1, 2]
>>> scores.sort()
>>> scores
[1, 2, 3]
```

Day 2 said that "class" is Python's word for a type, which is why `type()` answers `<class 'str'>`. A **class** is a definition of a new type: it says what data each value of that type holds and which methods it has. Each value made from a class is an **instance** of that class, also called an **object** of that class. "hello" is an instance of `str`, and `[3, 1, 2]` is an instance of `list`.

A helpful picture is a blank form. A class is like a printed form design for a library card: it has a box for a name and a box for a card number. Each filled-in card is an instance. All cards share the same design, but each holds its own name and number. The picture stops working in one place: a paper form cannot do anything, but an object also carries **methods**, the actions that make sense for it.

Writing your own class fixes both weaknesses from the game:

- One place, the class, says what data every player has, so every player is made the same way.
- The functions that work on players move **inside** the class and become methods, so they travel with the data: `ana.add_points(5)`, just like `scores.append(5)`.

Bundling data together with the behavior that works on it is the whole point of a class. The sections that follow build a `Player` class step by step.

KEY IDEA

A class defines a new type: the data each value holds and the methods that work on it. Each value made from the class is an instance (an object). Built-in types such as `str` and `list` are classes too.

PAUSE AND TRY

Question 1 Multiple choice

Which statement best describes the advantage of a `Player` class over player dictionaries plus separate functions?

- A. A class runs faster, because dictionaries are slow.
- B. A class keeps each player's data and the operations on a player together, and makes every player with the same attributes.
- C. A class lets you store more than two values per player.
- D. A class means you no longer need to write functions at all.

16.2 Defining a Class and Making Objects

The smallest possible class has a name and nothing else:

```
1 class Player:
2     pass
```

- `class` is the keyword that starts a **class definition**, the way `def` starts a function definition.
- `Player` is the class's name. Class names are written in **CapWords** (each word starts with a capital, no underscores), as you saw for exception names on Day 15: `Player`, `BankAccount`, `TaskNumberError`. That is how a reader tells a class from a function or variable, which use `snake_case`.
- The colon and the indented block are the **class body**. Like any block, it cannot be empty, so `pass` (Day 15) fills it for now.

Running a class definition does not make any players. It only creates the class, the way running a `def` line creates a function without calling it. To make an instance, **call the class** like a function, with parentheses. Making an instance is called **instantiating** the class.

WORKED EXAMPLE Two players from one class

make_players.py

```
1 class Player:
2     pass
3
4
5 ana = Player()
6 ben = Player()
7 print(type(ana))
8 print(ana is ben)
9 ana.name = "Ana"
10 ana.score = 5
11 print(ana.name, ana.score)
```

```
<class '__main__.Player'>
False
Ana 5
```

- Lines 5 and 6 call `Player()` twice, so there are two separate `Player` objects, one named `ana` and one named `ben`.
- Line 7: `type(ana)` is `<class '__main__.Player'>`. The type is your class. `__main__` is the name Python gives the file you run directly (Day 13), so the full name means "the `Player` class defined in the main file".
- Line 8: `ana is ben` is `False`. Each call to `Player()` makes a **new** object, so these are two objects, not two names for one (Day 4's identity).
- Lines 9 and 10 attach two **attributes** to `ana`. Day 14 defined an attribute as a value that belongs to an object, read with a dot and no parentheses, like `file.closed`. Writing `ana.name = "Ana"` works like assigning a variable, except the name `name` lives **on the object** `ana`.
- Line 11 reads them back.

So an object is something like a dictionary that you reach with dots instead of square brackets: `ana.name` instead of `ana["name"]`. But this example still has the first weakness from 16.1. Nothing says every player must have a name. Only `ana` got attributes:

missing_attribute.py

```
1 class Player:
2     pass
3
4
5 ana = Player()
6 ana.name = "Ana"
7 ben = Player()
8 print(ana.name)
9 print(ben.name)
```

Ana

Traceback (most recent call last):

```
File ".../day16/missing_attribute.py", line 9, in <module>
    print(ben.name)
    ^^^^^^^^^
```

AttributeError: 'Player' object has no attribute 'name'

An `AttributeError` is the object version of a `KeyError`: you asked an object for an attribute it does not have. (You met it on Day 7, when a dictionary had no `add` method.) `ben` was made by the same class, but nobody gave it a name. The fix is to make the class itself set up every attribute the moment an object is created, which is 16.3's job.

KEY IDEA

`class Name:` defines a class; calling `Name()` makes a new, separate instance each time.
`obj.attribute = value` stores a value on one object, and reading an attribute the object does not have raises an `AttributeError`.

WATCH OUT

Leaving off the parentheses. `ana = Player` does not make a player. Without the parentheses, the class is not called, so `ana` becomes a second name for the class itself:

no_parens.py

```
1 class Player:
2     pass
3
4
5 ana = Player
6 print(ana)
7 print(ana is Player)
```

```
<class '__main__.Player'>
True
```

Printing `ana` shows the class, and `ana is Player` is `True`. It is the same mistake as writing `print` instead of `print()`: naming a function is not calling it. Write `ana = Player()`.

PAUSE AND TRY

Question 2 Tracing

What does this program print?

lamp_trace.py

```
1 class Lamp:
2     pass
3
4
5 desk = Lamp()
6 hall = Lamp()
7 desk.on = True
8 hall.on = False
9 desk.on = not hall.on
10 hall.watts = 40
11 print(desk.on, hall.on, desk is hall)
12 print(hall.watts)
```

16.3 The Initializer and self

You want every `Player` to get a `name` and a `score` the moment it is made, without anyone remembering to add them. Python looks for a method with a special name, `__init__`, inside the class. If it exists, Python calls it automatically every time a new object is made, so its job is to set up the new object's starting attributes. `__init__` is called the **initializer** (many people also call it the constructor).

The name has two underscores on each side: `__init__`. Python uses names in this double-underscore form, called **dunder** names (short for "double underscore"), for methods it calls for you at special moments. You write them; Python decides when to run them.

```
1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
```

Take it apart:

- `def __init__(...)` sits **inside** the class body, indented under `class Player:`. A function defined inside a class is a **method** of that class.
- `self` is the first parameter. When Python calls `__init__`, it passes the **brand-new object** as `self`. Every method gets the object it is working on as its first parameter, and by convention that parameter is always named `self`.
- `name` and `score` are ordinary parameters (Day 9). They receive the arguments you write when you call the class.

- `self.name = name` creates an attribute called `name` **on the new object** and gives it the value of the parameter `name`. The two `name`s are different things: `self.name` is an attribute that lives on the object for as long as the object exists, while `name` is a local variable that disappears when `__init__` ends (Day 9). Using the same word for both is normal and makes the line easy to read: "this object's name is the name I was given".
- Attributes made this way, on one object through `self`, are **instance attributes**: each instance has its own.

WORKED EXAMPLE What happens when you call `Player("Ana", 5)`

player_init.py

```

1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6
7 ana = Player("Ana", 5)
8 ben = Player("Ben", 0)
9 print(ana.name, ana.score)
10 print(ben.name, ben.score)
```

```

Ana 5
Ben 0
```

Line 7, `Player("Ana", 5)`, does all of this:

| Step | What Python does |
|------|---|
| 1 | Makes a new, empty <code>Player</code> object |
| 2 | Calls <code>__init__</code> with <code>self</code> = that new object, <code>name</code> = "Ana", <code>score</code> = 5 |
| 3 | Line 3: <code>self.name = name</code> stores "Ana" on the new object |
| 4 | Line 4: <code>self.score = score</code> stores 5 on the new object |
| 5 | <code>__init__</code> ends; <code>Player(...)</code> gives back the new object, and line 7 names it <code>ana</code> |

Notice that you pass two arguments but `__init__` has three parameters. Python fills in `self` for you, then matches your arguments to the rest. Line 8 repeats the steps with a different new object, so `ben` gets its own `name` and `score`. Inside `__init__`, `self` was `ana` during the first call and `ben` during the second.

`__init__` is an ordinary function in every other way, so everything from Days 9 and 10 works in it. It can have default values (Day 10), so a new player can start at 0 points without saying so. It can also run code to clean up its arguments, and it can make attributes that do not come from a parameter at all:

player_defaults.py

```

1 class Player:
2     def __init__(self, name, score=0):
3         self.name = name.strip().title()
4         self.score = score
5         self.badges = []
6
7
8 ana = Player(" ana ")
9 ben = Player("Ben", 12)
10 ana.badges.append("first win")
11 print(ana.name, ana.score, ana.badges)
12 print(ben.name, ben.score, ben.badges)

```

```

Ana 0 ['first win']
Ben 12 []

```

- `Player(" ana ")` leaves out `score`, so the default 0 is used. Line 3 stores the name after `strip()` and `title()` (Day 3), so the messy text becomes "Ana".
- Line 5 gives each player a `badges` list. `__init__` runs once per object, so `[]` runs once per object, and each player gets its **own** list. Appending to `ana.badges` leaves `ben.badges` empty. (16.5 shows what goes wrong when a list is **not** made inside `__init__`.)

`__init__` must not return a value: its job is to set up `self`, and calling the class gives back the object automatically.

Because `__init__` has required parameters, calling the class with too few arguments fails at once, with a message that names what is missing:

missing_argument.py

```

1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6
7 ben = Player("Ben")

```

```

Traceback (most recent call last):
  File "../day16/missing_argument.py", line 7, in <module>
    ben = Player("Ben")
TypeError: Player.__init__() missing 1 required positional argument: 'score'

```

This is exactly what 16.1 wanted. A player without a score can no longer be made by accident, and the mistake is reported on the line that made it, not somewhere later.

KEY IDEA

`__init__(self, ...)` runs automatically every time the class is called. Python passes the new object as `self`, and lines such as `self.name = name` store instance attributes on it, so every object starts with the same attributes.

WATCH OUT

Forgetting `self`. Inside `__init__`, `name = name` without `self` just assigns the local variable to itself. Nothing is stored on the object:

forgot_self.py

```
1 class Player:
2     def __init__(self, name, score):
3         name = name
4         score = score
5
6
7 ana = Player("Ana", 5)
8 print("Made a player")
9 print(ana.name)
```

Made a player

Traceback (most recent call last):

```
File "../day16/forgot_self.py", line 9, in <module>
    print(ana.name)
    ^^^^^^^^^
```

AttributeError: 'Player' object has no attribute 'name'

Making the player works, so the error appears later, when something reads the attribute. When you see `has no attribute` for an attribute you are sure you set in `__init__`, look for a missing `self.` prefix. A misspelled `__init__` causes a different error, which the Common mistakes table below shows.

PAUSE AND TRY**Question 3** Tracing

What does this program print?

pet_trace.py

```
1 class Pet:
2     def __init__(self, name, kind="cat", age=1):
3         self.name = name
4         self.kind = kind
5         self.age_in_months = age * 12
6
7
8 rex = Pet("Rex", "dog", 3)
9 tom = Pet("Tom")
10 mia = Pet("Mia", age=2)
11 print(rex.name, rex.kind, rex.age_in_months)
12 print(tom.name, tom.kind, tom.age_in_months)
13 print(mia.name, mia.kind, mia.age_in_months)
```

Question 4 Debugging

This program should print the title and then 233. It prints `Holes` and then crashes. Find the bug and fix it.

book_pages.py

```
1 class Book:
2     def __init__(self, title, pages):
3         self.title = title
4         pages = pages
5
6
7 novel = Book("Holes", 233)
8 print(novel.title)
9 print(novel.pages)
```

16.4 Methods

Now for the behavior half of the bundle. The game needs to add points to a player and ask whether a player has reached a target. Instead of loose functions, define them inside the class, below `__init__`:

```
1     def add_points(self, points):
2         self.score += points
```

A **method** is a function defined inside a class. You have called methods since Day 3 (`name.upper()`); now you write them. Every method takes `self` as its first parameter, for the same reason as `__init__`: it needs to know **which** object to work on. Inside the method, `self.score` means "the score of the object this method was called on".

WORKED EXAMPLE Methods that change and methods that answer

player_methods.py

```

1 class Player:
2     def __init__(self, name, score=0):
3         self.name = name
4         self.score = score
5
6     def add_points(self, points):
7         self.score += points
8
9     def has_won(self, target):
10        return self.score >= target
11
12
13 ana = Player("Ana")
14 ben = Player("Ben")
15 ana.add_points(5)
16 ana.add_points(7)
17 ben.add_points(3)
18 print(ana.score, ana.has_won(10))
19 print(ben.score, ben.has_won(10))
20 Player.add_points(ben, 8)
21 print(ben.score, ben.has_won(10))

```

```

12 True
3 False
11 True

```

When you write `ana.add_points(5)` on line 15, Python finds `add_points` in `ana`'s class and calls it with `ana` as `self` and 5 as `points`. Line 20 shows the long form of the same thing:

`Player.add_points(ben, 8)` names the class, then passes the object and the argument explicitly. You will almost never write the long form, but it shows exactly what the dot does: **the object before the dot becomes `self`**.

| Line | Call | self | Effect |
|------|--|------------------|--|
| 15 | <code>ana.add_points(5)</code> | <code>ana</code> | <code>ana.score: 0 → 5</code> |
| 16 | <code>ana.add_points(7)</code> | <code>ana</code> | <code>ana.score: 5 → 12</code> |
| 17 | <code>ben.add_points(3)</code> | <code>ben</code> | <code>ben.score: 0 → 3</code> |
| 18 | <code>ana.has_won(10)</code> | <code>ana</code> | returns <code>12 >= 10</code> , <code>True</code> |
| 19 | <code>ben.has_won(10)</code> | <code>ben</code> | returns <code>3 >= 10</code> , <code>False</code> |
| 20 | <code>Player.add_points(ben, 8)</code> | <code>ben</code> | <code>ben.score: 3 → 11</code> |

There are two kinds of method here, as with list methods on Day 6. `add_points` **changes** the object and returns nothing (so it gives back `None`, like `append`). `has_won` **answers a question** about the object and changes nothing. One method's change is seen by every later call, because they all read the same attribute on the same object.

Methods can do anything a function can: take several parameters, raise exceptions (Day 15), and call other methods of the same object through `self`. This shopping cart guards its data and builds a receipt from another method's answer:

cart.py

```
1 class Cart:
2     def __init__(self, owner):
3         self.owner = owner
4         self.prices = {}
5
6     def add(self, item, price):
7         if price <= 0:
8             raise ValueError(f"price must be more than 0, not {price}")
9         self.prices[item] = price
10
11    def total(self):
12        return sum(self.prices.values())
13
14    def receipt(self):
15        count = len(self.prices)
16        return f"{self.owner}: {count} items, {self.total():.2f}"
17
18
19 cart = Cart("Ana")
20 cart.add("bread", 2.25)
21 cart.add("milk", 1.5)
22 try:
23     cart.add("apples", -3)
24 except ValueError as error:
25     print(f"Not added: {error}")
26 print(cart.receipt())
```

```
Not added: price must be more than 0, not -3
Ana: 2 items, 3.75
```

- `__init__` gives each cart its own empty dictionary of prices (line 4).
- `add` refuses a price of 0 or less by raising a `ValueError` (line 8), so a bad price never gets into the dictionary. The caller decides what to do about it (lines 22 to 25), exactly as on Day 15.
- `receipt` calls `self.total()` on line 16. Inside a method, `self` is the object, so `self.total()` calls another method on the **same** cart. Writing just `total()` would fail with a `NameError`, because methods are not loose names in the file; they are reached through an object.
- The total is $2.25 + 1.5 = 3.75$.

Protecting data like this is one of the main reasons to use methods: if every change to the prices goes through `add`, a negative price can never sneak in.

KEY IDEA

A method is a function inside a class whose first parameter is `self`. `obj.method(args)` runs it with `obj` as `self`, so `self.attribute` reads or changes that object's data, and `self.other_method()` calls another method on it.

WATCH OUT

Forgetting `self` in the method's parameters. Python always passes the object as the first argument, whether or not you left room for it:

forgot_self_param.py

```
1 class Player:
2     def __init__(self, name):
3         self.name = name
4         self.score = 0
5
6     def add_points(points):
7         self.score += points
8
9
10 ana = Player("Ana")
11 ana.add_points(5)
```

Traceback (most recent call last):

```
File ".../day16/forgot_self_param.py", line 11, in <module>
  ana.add_points(5)
  ~~~~~^
```

TypeError: Player.add_points() takes 1 positional argument but 2 were given

You wrote one argument, 5, so "2 were given" looks wrong. It counts the hidden one: `ana` was passed as the first argument and 5 as the second, but the method only has one parameter. Whenever a method's error counts one more argument than you wrote, check that `self` is the first parameter. Write `def add_points(self, points):`.

PAUSE AND TRY**Question 5** Tracing

What does this program print?

counter_trace.py

```
1 class Counter:
2     def __init__(self, start=0):
3         self.value = start
4
5     def up(self, step=1):
6         self.value += step
7         return self.value
8
9     def reset(self):
10        self.value = 0
11
12
13 clicks = Counter(5)
14 print(clicks.up())
15 print(clicks.up(3))
16 clicks.reset()
17 clicks.up()
18 print(clicks.value)
```

Question 6 Multiple choice

ana is a Player. Which call does exactly the same thing as `ana.add_points(5)`?

- A. `add_points(ana, 5)`
- B. `Player.add_points(ana, 5)`
- C. `Player.add_points(5)`
- D. `ana.add_points(ana, 5)`

16.5 Instance Attributes Versus Class Attributes

Every player in the game starts with 3 lives. That number is not about any one player; it is a rule of the game, the same for all of them. Storing a copy on every player works, but then changing the rule means changing every object. A value that belongs to the whole class goes in the class body itself, outside any method:

```
1 class Player:
2     max_lives = 3
```

A variable assigned directly in the class body is a **class attribute**: there is **one** of it, stored on the class and shared by every instance. An attribute set through `self` in a method is an instance attribute: each object has its own.

WORKED EXAMPLE One shared value, many players

class_attribute.py

```

1 class Player:
2     max_lives = 3
3
4     def __init__(self, name):
5         self.name = name
6         self.lives = Player.max_lives
7
8
9 ana = Player("Ana")
10 ben = Player("Ben")
11 print(Player.max_lives, ana.max_lives, ben.max_lives)
12 Player.max_lives = 5
13 cy = Player("Cy")
14 print(ana.lives, cy.lives, ana.max_lives)
15 ana.max_lives = 9
16 print(ana.max_lives, ben.max_lives, Player.max_lives)

```

```

3 3 3
3 5 5
9 5 5

```

When you read `ana.max_lives`, Python looks for the attribute **on the object first**. `ana` has no `max_lives` of its own, so Python then looks **on the class** and finds 3 there. That is why line 11 prints 3 three times: it is the same value, reached three ways.

| Line | What happens |
|------|--|
| 11 | <code>Player.max_lives</code> is 3; <code>ana</code> and <code>ben</code> have none of their own, so both find the class's 3 |
| 12 | Changes the class attribute: from now on every lookup that reaches the class finds 5 |
| 13 | <code>Cy's</code> <code>__init__</code> copies the current class value into <code>cy.lives</code> , which is 5 |
| 14 | <code>ana.lives</code> is still 3: it was copied when <code>ana</code> was made. <code>ana.max_lives</code> now finds 5 on the class |
| 15 | Assigning through <code>ana</code> creates a new instance attribute on <code>ana</code> , which hides the class's value for <code>ana</code> only |
| 16 | <code>ana.max_lives</code> is 9 (found on the object); <code>ben.max_lives</code> and <code>Player.max_lives</code> are still 5 |

Line 15 is the surprising one. **Reading** through an object falls back to the class, but **assigning** through an object always stores on that object. To change a class attribute, assign through the class name, as line 12 does.

A class attribute is also the place for data that genuinely belongs to the class as a whole, such as how many players exist. Each `__init__` updates the shared count through the class name:

count_players.py

```

1 class Player:
2     count = 0
3
4     def __init__(self, name):
5         self.name = name
6         Player.count += 1
7         self.number = Player.count
8
9
10 ana = Player("Ana")
11 ben = Player("Ben")
12 cy = Player("Cy")
13 print(Player.count)
14 print(ana.number, ben.number, cy.number)

```

```

3
1 2 3

```

`Player.count += 1` reads the shared count, adds 1, and stores it back **on the class**. Then `self.number = Player.count` gives each player its own copy of the count at the moment it was made, so `ana` is player 1 and `cy` is player 3.

| Kind | Made by | Stored on | Use it for |
|--------------------|--|-------------|---|
| Instance attribute | <code>self.name = ...</code> in a method | Each object | Data that differs per object: name, score |
| Class attribute | <code>max_lives = 3</code> in the class body | The class | Rules and settings shared by all; class-wide counts |

KEY IDEA

Instance attributes (`self.x = ...`) belong to one object; class attributes (assigned in the class body) are shared by all instances. Reading `obj.x` looks on the object first, then on the class. Assigning `obj.x = ...` always stores on the object, so change a class attribute through the class name.

WATCH OUT

A list as a class attribute. This is Day 10's mutable default trap in a new place. The class body runs **once**, so a list there is made once and shared by every instance:

shared_list.py

```
1 class Team:
2     members = []
3
4     def __init__(self, name):
5         self.name = name
6
7     def add(self, player_name):
8         self.members.append(player_name)
9
10
11 red = Team("Red")
12 blue = Team("Blue")
13 red.add("Ana")
14 blue.add("Ben")
15 print(red.name, red.members)
16 print(blue.name, blue.members)
```

```
Red ['Ana', 'Ben']
Blue ['Ana', 'Ben']
```

`self.members.append(...)` is a read, not an assignment: neither team has its own `members`, so both find the class's list and append to that one list. The teams were supposed to be separate. Make the list in `__init__`, which runs once **per object**:

shared_list_fixed.py

```
1 class Team:
2     def __init__(self, name):
3         self.name = name
4         self.members = []
5
6     def add(self, player_name):
7         self.members.append(player_name)
8
9
10 red = Team("Red")
11 blue = Team("Blue")
12 red.add("Ana")
13 blue.add("Ben")
14 print(red.name, red.members)
15 print(blue.name, blue.members)
```

```
Red ['Ana']
Blue ['Ben']
```

Rule of thumb: anything one object can change for itself (lists, dictionaries, scores) is an instance attribute made in `__init__`. Keep class attributes for values that are shared on purpose.

PAUSE AND TRY**Question 7** Tracing

This class tries to count tickets sold. What does it print? Look carefully at line 6.

self_count.py

```
1 class Ticket:
2     sold = 0
3
4     def __init__(self, seat):
5         self.seat = seat
6         self.sold += 1
7
8
9 first = Ticket("A1")
10 second = Ticket("A2")
11 print(first.sold, second.sold, Ticket.sold)
```

Question 8 Short answer

A Bus class needs these three pieces of data: the bus's route number, the fare (the same for every bus in the city), and the list of passengers on board. Which should be class attributes and which instance attributes? Explain each choice.

16.6 Printing an Object Readably

Try printing a `Player` from 16.3. Python does not know what a player should look like as text, so it prints the only thing it knows: the class name and where the object lives in memory. A run printed `<__main__.Player object at 0x104f8a2d0>`. The number after `at` is a memory address written in hexadecimal; it is different on every run and every computer, which is why this program checks only the parts that stay the same:

default_str.py

```
1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6
7 ana = Player("Ana", 5)
8 text = str(ana)
9 print(text.startswith("<__main__.Player object at 0x"))
10 print(text.endswith(">"))
```

True
True

That text is useless to a user. You can tell Python how to turn your object into text with another dunder method, `__str__`. Whenever Python needs an object as readable text, it calls the object's

`__str__` method and uses the string it returns. Three everyday things need text: `print(obj)`, `str(obj)`, and an f-string such as `f"{obj}"`.

```
1 def __str__(self):
2     return f"{self.name} ({self.score} points)"
```

`__str__` takes only `self`, and it must **return** a string. It does not print anything itself; whoever asked for the text decides what to do with it.

WORKED EXAMPLE One method, three ways to use it

player_str.py

```
1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6     def __str__(self):
7         return f"{self.name} ({self.score} points)"
8
9
10 ana = Player("Ana", 5)
11 print(ana)
12 label = str(ana)
13 print(label.upper())
14 print(f"Leader: {ana}!")
```

```
Ana (5 points)
ANA (5 POINTS)
Leader: Ana (5 points)!
```

- Line 11: `print(ana)` calls `ana.__str__()` and prints the string it returns.
- Line 12: `str(ana)` calls the same method and gives back the string, which is an ordinary `str`, so line 13 can call `upper()` on it.
- Line 14: the f-string needs `ana` as text, so it calls `__str__` too.

You never call `__str__` yourself. You write it once, and Python calls it whenever your object is turned into text.

A good `__str__` gives the information a **user** would want to see, in a short, natural form. It is the object's answer to "describe yourself to a person".

KEY IDEA

Define `__str__(self)` to return a readable string. `print(obj)`, `str(obj)`, and `f"{obj}"` all call it. Without it, Python shows the class name and a memory address.

WATCH OUT

Printing instead of returning. It is tempting to write `print(...)` inside `__str__`, because the goal is to print. But Python is asking the method for a string, and a method without `return` gives back `None`:

str_prints.py

```
1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6     def __str__(self):
7         print(f"{self.name} ({self.score} points)")
8
9
10 ana = Player("Ana", 5)
11 print(ana)
```

```
Ana (5 points)
Traceback (most recent call last):
  File ".../day16/str_prints.py", line 11, in <module>
    print(ana)
    ~~~~~^^^^^
TypeError: __str__ returned non-string (type NoneType)
```

The text did appear, from the `print` inside the method, and then Python got `None` back and stopped. This is Day 9's returning versus printing again. Write `return f"..."`.

PAUSE AND TRY**Question 9** Tracing

What does this program print?

score_str.py

```
1 class Score:
2     def __init__(self, team, points):
3         self.team = team
4         self.points = points
5
6     def __str__(self):
7         return f"{self.team}: {self.points}"
8
9
10 home = Score("Lions", 2)
11 print(home)
12 home.points += 1
13 message = f"Now {home}"
14 print(message)
15 print(len(str(home)))
```

16.7 Showing an Object to a Programmer

`__str__` fixes `print(ana)`. But put players in a list and print the list, and the addresses come back:

`str_only_list.py`

```
1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6     def __str__(self):
7         return f"{self.name} ({self.score} points)"
8
9
10 ana = Player("Ana", 5)
11 ben = Player("Ben", 0)
12 print(ana)
13 listed = str([ana, ben])
14 print(listed.startswith("<__main__.Player object at 0x"))
```

```
Ana (5 points)
True
```

The list printed as `[<__main__.Player object at 0x...>, <__main__.Player object at 0x...>]`, which lines 13 and 14 check. The reason goes back to Day 3. There, the REPL showed a string **with** quotes ('Ana') while `print` showed it without. Python has two text forms for every value:

- The **str** form, for a user: readable, no extra detail. `print` and `str()` use it.
- The **repr** form, for a programmer: as unambiguous as possible, ideally looking like the code that makes the value. The REPL uses it when it shows a result, and the built-in function `repr()` gives it to you directly.

```
>>> name = "Ana"
>>> print(name)
Ana
>>> print(repr(name))
'Ana'
>>> name
'Ana'
>>> print(f"Hi {name}! Hi {name!r}!")
Hi Ana! Hi 'Ana'!
```

`print(repr(name))` shows the quotes, because the `repr` form of a string is the way you would type it. In an f-string, adding `!r` after the expression inside the braces asks for the `repr` form instead of the `str` form.

When you print a list, the **list** is turned into text with `str`, but the list builds that text from the **repr** of each item. That is why Day 6's lists showed strings with quotes, `['milk', 'eggs']`: the quotes tell you '15' from 15. A player's `repr` is what `print` showed above, because you have only written `__str__`. To control the `repr` form, define `__repr__`:

WORKED EXAMPLE A player with both forms

player_repr.py

```

1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6     def __str__(self):
7         return f"{self.name} ({self.score} points)"
8
9     def __repr__(self):
10        return f"Player({self.name!r}, {self.score})"
11
12
13 ana = Player("Ana", 5)
14 ben = Player("Ben", 0)
15 print(ana)
16 print(repr(ana))
17 print([ana, ben])
18 print(f"{ana} and {ben!r}")

```

```

Ana (5 points)
Player('Ana', 5)
[Player('Ana', 5), Player('Ben', 0)]
Ana (5 points) and Player('Ben', 0)

```

- Line 15: `print` uses `__str__`.
- Line 16: `repr(ana)` calls `__repr__`.
- Line 17: the list uses `__repr__` for each item.
- Line 18: `{ana}` gives the str form and `{ben!r}` gives the repr form.

`__repr__` follows a convention: it returns text that looks like the call that would make an equal object, `Player('Ana', 5)`. Line 10 builds it with `{self.name!r}`, so the name appears with quotes, exactly as you would type it in code. A programmer reading `Player('Ana', 5)` in a list, a traceback, or the REPL knows at once what the object is.

If a class has `__repr__` but no `__str__`, Python uses `__repr__` for **both** forms. So when you write only one, write `__repr__`; it covers `print` as well. This REPL session defines a class with only `__repr__`:

```
>>> class Player:
...     def __init__(self, name, score):
...         self.name = name
...         self.score = score
...     def __repr__(self):
...         return f"Player({self.name!r}, {self.score})"
...
>>> ana = Player("Ana", 5)
>>> ana
Player('Ana', 5)
>>> print(ana)
Player('Ana', 5)
>>> Player("Ana", 5).score
5
```

Typing `ana` at the prompt shows the repr form, and `print(ana)` falls back to it as well.

| Who asks for text | Uses |
|---|---|
| <code>print(obj)</code> , <code>str(obj)</code> , <code>f"{obj}"</code> | <code>__str__</code> (or <code>__repr__</code> if no <code>__str__</code>) |
| <code>repr(obj)</code> , <code>f"{obj!r}"</code> , the REPL | <code>__repr__</code> |
| Printing a list, tuple, or dictionary | <code>__repr__</code> of each item |

KEY IDEA

`__str__` is for users and `__repr__` is for programmers. `__repr__` should look like the code that makes the object, such as `Player('Ana', 5)`. Lists show their items with `__repr__`, and `print` falls back to `__repr__` when there is no `__str__`.

PAUSE AND TRY

Question 10 Multiple choice

What is the **second** line this program prints?

song_repr.py

```
1 class Song:
2     def __init__(self, title, plays):
3         self.title = title
4         self.plays = plays
5
6     def __str__(self):
7         return f"{self.title} ({self.plays} plays)"
8
9     def __repr__(self):
10        return f"Song({self.title!r}, {self.plays})"
11
12
13 hits = [Song("Echo", 12), Song("Glow", 7)]
14 print(hits[0])
15 print(hits)
```

- A. [Echo (12 plays), Glow (7 plays)]
- B. [Song('Echo', 12), Song('Glow', 7)]
- C. Song('Echo', 12) Song('Glow', 7)
- D. A list of <__main__.Song object at ...> entries

16.8 Objects Are References: Identity Versus Equality

On Day 6 you learned that `b = a` does not copy a list; it gives the same list a second name. Your own objects follow exactly the same rule, because every value in Python is an object and every name refers to one (Day 4). A name never **contains** a player. It **refers to** one, the way a contact in your phone refers to a person. Two contacts can point to the same person.

WORKED EXAMPLE Two names, one player

shared_player.py

```

1 class Player:
2     def __init__(self, name, score=0):
3         self.name = name
4         self.score = score
5
6     def add_points(self, points):
7         self.score += points
8
9
10 def reward(player):
11     player.add_points(10)
12
13
14 ana = Player("Ana", 5)
15 leader = ana
16 leader.add_points(3)
17 print(ana.score, leader is ana)
18 reward(ana)
19 print(ana.score, leader.score)
```

```

8 True
18 18
```

| Line | Action | Objects |
|------|-----------------------------------|---|
| 14 | <code>Player("Ana", 5)</code> | One Player, score 5, named ana |
| 15 | <code>leader = ana</code> | Still one Player, now with two names |
| 16 | <code>leader.add_points(3)</code> | That one player's score: 5 → 8 |
| 18 | <code>reward(ana)</code> | Inside reward, the parameter <code>player</code> is a third name for it; score 8 → 18 |

Line 16 changes the player through `leader`, and line 17 sees the change through `ana`, because there is only one player. Line 18 passes `ana` to a function, and the function's change is visible

after it returns. This is what Day 9 showed for lists: a parameter refers to the **same** object as the argument, so changing the object (rather than assigning a new value to the parameter) changes it for the caller.

Now the other side of the question. Make a **second** player with the same name and score. Are the two equal?

identity_equality.py

```
1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6
7 ana = Player("Ana", 5)
8 twin = Player("Ana", 5)
9 leader = ana
10 print(leader is ana, leader == ana)
11 print(twin is ana, twin == ana)
12 print(twin.name == ana.name and twin.score == ana.score)
```

```
True True
False False
True
```

- Line 10: `leader` and `ana` are one object, so both `is` and `==` are `True`.
- Line 11: `twin` and `ana` hold the same data, yet `twin == ana` is `False`. For your own classes, `==` does not compare attributes. Unless you teach it otherwise, `==` on two objects of your class means the same as `is`: "are these the same object?" Python cannot guess which attributes matter for equality. (Do two players with the same name but different scores count as equal?) Day 17 shows how to define equality with another dunder method.
- Line 12: to compare the data today, compare the attributes yourself.

This is the difference from built-in types, where Day 4's rule held: `==` compares contents. For a class you write, `==` starts out as identity.

KEY IDEA

A name refers to an object; assignment and passing an argument never copy it, so a change through any name is seen through all of them. `is` asks "same object?" For your own classes, `==` asks the same question until you define equality yourself.

WATCH OUT

One object in many places. This loop is meant to make a team of three guests. It appends **the same** player three times:

same_object.py

```
1 class Player:
2     def __init__(self, name, score=0):
3         self.name = name
4         self.score = score
5
6
7 guest = Player("Guest")
8 team = []
9 for _ in range(3):
10     team.append(guest)
11 team[0].score += 5
12 for player in team:
13     print(player.name, player.score)
```

```
Guest 5
Guest 5
Guest 5
```

Only `team[0]` got the points, but all three show them, because the list holds three references to one object. Calling the class makes a new object, so call it **inside** the loop, once per player you need:

same_object_fixed.py

```
1 class Player:
2     def __init__(self, name, score=0):
3         self.name = name
4         self.score = score
5
6
7 team = []
8 for number in range(1, 4):
9     team.append(Player(f"Guest {number}"))
10 team[0].score += 5
11 for player in team:
12     print(player.name, player.score)
```

```
Guest 1 5
Guest 2 0
Guest 3 0
```

To get a separate object with the same data, make a new one: `Player(ana.name, ana.score)`. Assignment never copies.

PAUSE AND TRY

Question 11 Tracing

What does this program print? First decide how many `Box` objects there are.

box_trace.py

```
1 class Box:
2     def __init__(self, label):
3         self.label = label
4         self.items = []
5
6
7 def pack(box, item):
8     box.items.append(item)
9
10
11 a = Box("toys")
12 b = a
13 c = Box("toys")
14 pack(b, "ball")
15 pack(c, "kite")
16 print(a.items, c.items)
17 print(a is b, a == c, a.label == c.label)
```

Question 12 Short answer

A classmate writes `backup = account` before changing a `BankAccount` object's balance, so the old balance can be restored later. Explain why this does not work and how to keep the old data.

Putting it together

A school library tracks which books are out. Each book is an object that knows its title, its author, and who has borrowed it, and it refuses to be checked out twice. A list of a member's loans holds references to the same book objects that sit in the shelf list.

library.py

```
1 class LoanError(Exception):
2     pass
3
4
5 class Book:
6     loan_days = 14
7
8     def __init__(self, title, author):
9         self.title = title
10        self.author = author
11        self.borrower = None
12
13    def check_out(self, member):
14        if self.borrower is not None:
15            raise LoanError(f"{self.title} is already out with {self.borrower}")
16        self.borrower = member
17
18    def give_back(self):
```

```

19         self.borrower = None
20
21     def __str__(self):
22         if self.borrower is None:
23             status = "on the shelf"
24         else:
25             status = f"out with {self.borrower}"
26         return f"{self.title} by {self.author}: {status}"
27
28     def __repr__(self):
29         return f"Book({self.title!r}, {self.author!r})"
30
31
32 shelf = [
33     Book("Holes", "Louis Sachar"),
34     Book("Wonder", "R. J. Palacio"),
35     Book("Matilda", "Roald Dahl"),
36 ]
37 ana_loans = [shelf[0], shelf[2]]
38 for book in ana_loans:
39     book.check_out("Ana")
40
41 try:
42     shelf[0].check_out("Ben")
43 except LoanError as error:
44     print(f"Sorry: {error}")
45
46 for book in shelf:
47     print(book)
48 print(ana_loans)
49 print(f"Loans last {Book.loan_days} days.")
50 shelf[2].give_back()
51 print(ana_loans[1], ana_loans[1] is shelf[2])

```

```

Sorry: Holes is already out with Ana
Holes by Louis Sachar: out with Ana
Wonder by R. J. Palacio: on the shelf
Matilda by Roald Dahl: out with Ana
[Book('Holes', 'Louis Sachar'), Book('Matilda', 'Roald Dahl')]
Loans last 14 days.
Matilda by Roald Dahl: on the shelf True

```

| Lines | What happens | Idea |
|-------|---|------------|
| 1-2 | A custom exception for loan problems, the Day 15 pattern: a class whose body is just pass | 16.2 |
| 6 | loan_days is a class attribute: the same loan period for every book | 16.5 |
| 8-11 | __init__ gives every book a title, an author, and a borrower that starts as None (Day 4's "no value yet") | 16.3 |
| 13-19 | check_out raises LoanError if the book is already out; give_back clears the borrower | 16.4 |
| 21-29 | __str__ describes the book for a person; __repr__ looks like the code that makes it | 16.6, 16.7 |

| | | |
|-------|--|------------|
| 37 | ana_loans holds references to the first and third books, not copies | 16.8 |
| 38–39 | Checking out through ana_loans changes the books in shelf too, since they are the same objects | 16.8 |
| 41–44 | Ben asks for Holes; check_out raises, and the caller handles it | 16.4 |
| 46–48 | print(book) uses <code>__str__</code> ; printing the list uses <code>__repr__</code> for each item | 16.6, 16.7 |
| 50–51 | Returning shelf[2] is seen through ana_loans[1], and is confirms they are one object | 16.8 |

Look at what the class bought you. Every book is made the same way. The rule "a book cannot be out twice" lives in one method, so no part of the program can break it by accident. And the rest of the program reads like a description of a library: `book.check_out("Ana")`, `print(book)`.

Common mistakes

Errors in class code can be confusing, because Python counts the hidden `self` argument and reports some mistakes far from their cause. Use this table to go from a message to its usual cause.

| Message | Usual cause | Fix |
|--|--|---|
| AttributeError: 'Player' object has no attribute 'name' | name = name without self. in <code>__init__</code> , or a misspelled attribute | Store with <code>self.name = name</code> |
| TypeError: Player.add_points() takes 1 positional argument but 2 were given | The method has no self parameter | <code>def add_points(self, points):</code> |
| TypeError: Player.__init__() missing 1 required positional argument: 'score' | The call left out an argument that <code>__init__</code> needs | Pass it, or give the parameter a default |
| TypeError: Player() takes no arguments | The class has no <code>__init__</code> : usually a misspelling such as <code>__int__</code> or <code>_init_</code> | Spell it <code>__init__</code> , two underscores each side |
| TypeError: <code>__str__</code> returned non-string (type NoneType) | <code>__str__</code> prints instead of returning | return the string |
| NameError: name 'name' is not defined inside a method | Used an attribute without self. | Write <code>self.name</code> |
| Printed <code><__main__.Player object at 0x...></code> | No <code>__str__</code> or <code>__repr__</code> (or a list printed with no <code>__repr__</code>) | Define <code>__repr__</code> , and <code>__str__</code> if needed |

The fourth row is easy to fall into, because a misspelled `__init__` is not an error by itself: Python just sees an ordinary method with an odd name. The class then has no initializer, so it accepts no arguments:

init_typo.py

```

1 class Player:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6
7 ana = Player("Ana", 5)

```

```

Traceback (most recent call last):
  File ".../day16/init_typo.py", line 7, in <module>
    ana = Player("Ana", 5)
TypeError: Player() takes no arguments

```

You saw this exact message on Day 15, when a custom exception class was missing (Exception). Now you know why: a class with no `__init__` of its own accepts no arguments. Older versions of Python may word these messages a little differently.

Vocabulary

| Term | Definition | Example |
|--|--|--|
| class | A definition of a new type: the data its values hold and the methods they have | <code>class Player:</code> |
| instance / object | One value made from a class | <code>ana = Player("Ana", 5)</code> |
| class definition / class body | The <code>class</code> statement / the indented block under it | <code>class Player:</code> then its block |
| CapWords | The naming style for classes: each word capitalized, no underscores | <code>BankAccount</code> |
| instantiate | Make an instance by calling the class | <code>Player()</code> |
| AttributeError | Raised when an object has no attribute with that name | <code>ben.name</code> when it was never set |
| <code>__init__</code> / initializer | The method Python calls automatically to set up each new object | <code>def __init__(self, name):</code> |
| dunder | "Double underscore": a special name Python calls for you | <code>__init__</code> , <code>__str__</code> |
| self | The first parameter of every method: the object the method works on | <code>self.score += points</code> |
| instance attribute | An attribute stored on one object, usually set through <code>self</code> in <code>__init__</code> | <code>self.name = name</code> |
| method | A function defined inside a class; called through an object | <code>ana.add_points(5)</code> |
| class attribute | A variable in the class body, shared by every instance | <code>max_lives = 3</code> |
| <code>__str__</code> | Returns the readable text for an object; used by <code>print</code> , <code>str()</code> , and f-strings | <code>"Ana (5 points)"</code> |

| | | |
|---|--|---|
| <code>__repr__</code> / <code>repr()</code> | Returns the programmer's text for an object; used by the REPL, <code>repr()</code> , and lists | <code>"Player('Ana', 5)"</code> |
| <code>!r</code> | In an f-string, use the repr form of the value | <code>f"{name!r}"</code> gives <code>'Ana'</code> |
| reference | What a name holds: a way to reach an object, not a copy of it | <code>leader = ana</code> |

Summary

- A **class** defines a new type that bundles data with the behavior that works on it. Calling the class makes a new **instance** (object) each time; built-in types such as `str` and `list` are classes too.
- `__init__(self, ...)` runs automatically for every new object and sets up its **instance attributes** with `self.name = name`, so every object starts with the same attributes.
- A **method** is a function inside a class whose first parameter is `self`; `obj.method(args)` passes `obj` as `self`. Methods read and change the object's attributes, raise exceptions to protect its data, and call each other through `self`.
- **Class attributes** in the class body are shared by every instance; reading through an object falls back to the class, but assigning through an object makes an instance attribute. Make changeable lists in `__init__`.
- `__str__` returns readable text for users (`print`, `str()`, f-strings); `__repr__` returns code-like text for programmers (the REPL, `repr()`, items in a list).
- Names hold **references**: assignment and arguments never copy an object. `is` asks "same object?", and for your own classes `==` asks the same until you define equality.

Practice

Do these in your `day16` folder. Type each program yourself, run it, and only then open the answer.

Question 13 Short answer

A bank account. Write a `BankAccount` class with an owner and a balance (starting at 0 unless given), a `deposit(amount)` method, and a `withdraw(amount)` method. The balance must never go negative: refuse a withdrawal larger than the balance, and refuse amounts of 0 or less, by raising `ValueError` with a clear message. Then make an account and test every path.

Question 14 Short answer

Tasks as objects. Write a `Task` class with a `title`, a `done` flag that starts as `False`, and a `complete()` method. Then convert the Day 15 to-do manager to keep a list of `Task` objects instead of dictionaries. The save file should stay the same JSON as before.

Question 15 Short answer

Readable printing. Add `__str__` to both classes, so that `print(account)` shows the owner and balance and `print(task)` shows a check box and the title. Use the `Task` version in `list_tasks`.

Review questions

Question 16 Tracing

What does this program print? Watch when each attribute is looked up.

review_tickets.py

```
1 class Ticket:
2     price = 8
3     sold = 0
4
5     def __init__(self, film, seats=1):
6         self.film = film
7         self.seats = seats
8         Ticket.sold += seats
9
10    def cost(self):
11        return self.seats * self.price
12
13    def __str__(self):
14        return f"{self.seats} x {self.film} = {self.cost()}"
15
16
17 evening = Ticket("Up", 3)
18 Ticket.price = 10
19 matinee = Ticket("Coco")
20 matinee.price = 6
21 print(evening)
22 print(matinee)
23 print(Ticket.sold, Ticket.price)
```

Question 17 Debugging

This program should print `Hello, Ana!`. It crashes instead. Explain the error and fix it.

greeter.py

```
1 class Greeter:
2     def __init__(self, name):
3         self.name = name
4
5     def greet(self):
6         return f"Hello, {name}!"
7
8
9 host = Greeter("Ana")
10 print(host.greet())
```

Question 18 Multiple choice

A class `Player` has an `__init__(self, name, score)` method. In what order do these happen when Python runs `ana = Player("Ana", 5)`?

- A. `__init__` runs → a new object is made → the name `ana` refers to it
- B. A new object is made → `__init__` runs with it as `self` → the name `ana` refers to it
- C. The name `ana` is made → `__init__` runs with `ana` as `self` → a new object is made
- D. A new object is made → the name `ana` refers to it → `__init__` returns the object

Question 19 Tracing

What does this program print? `sorted` with `key` is from Day 10.

runners.py

```

1 class Runner:
2     def __init__(self, name, time):
3         self.name = name
4         self.time = time
5
6     def __repr__(self):
7         return f"Runner({self.name!r}, {self.time})"
8
9
10 runners = [Runner("Ana", 31), Runner("Ben", 28), Runner("Cy", 35)]
11 ranked = sorted(runners, key=lambda runner: runner.time)
12 ranked[0].time -= 2
13 print(ranked)
14 print(runners)
15 print(ranked[0] is runners[1], ranked == runners)

```

Question 20 Short answer

Design a class from this description: "A library card belongs to one person and has a card number. A card can hold up to 5 borrowed books, and that limit is the same for every card. A member can borrow a book (refused if the card is full) and return one. Printing a card shows its number, its owner, and how many books are out." List the class's attributes (saying which are class attributes), its methods, and what `__str__` should return.

Question 21 Short answer

A class has `__str__` but no `__repr__`. Its objects print nicely one at a time, but a list of them prints as `<__main__.Point object at 0x...>` entries. Explain why, and say which method you would add first to a new class, and why.

Go further

The official Python Tutorial explains how classes work, including the namespaces and scopes that make `self.name` and plain `name` different, in [Classes](#), up to and including "A First Look at Classes". Tomorrow, Day 17, builds new classes on top of existing ones.

Reflection

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

- ☐ I can explain why a class that bundles data and behavior beats a dictionary plus loose functions. Reread: [16.1 Why Classes: Bundling Data with Behavior](#)
- ☐ I can define a class and make separate instances of it. Reread: [16.2 Defining a Class and Making Objects](#)
- ☐ I can write `__init__` with `self` to give every new object its starting attributes. Reread: [16.3 The Initializer and self](#)

- ☐ I can write methods and say which object becomes self in a call. Reread: 16.4 Methods
- ☐ I can tell instance attributes from class attributes and predict which one a lookup finds. Reread: 16.5 Instance Attributes Versus Class Attributes
- ☐ I can write `__str__` so print shows an object readably. Reread: 16.6 Printing an Object Readably
- ☐ I can write `__repr__` and explain when Python uses it instead of `__str__`. Reread: 16.7 Showing an Object to a Programmer
- ☐ I can predict what happens when two names refer to one object, and tell identity from equality. Reread: 16.8 Objects Are References: Identity Versus Equality

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. A class bundles the data with the behavior that works on it, and one definition decides what data every instance has.

- **A** is wrong: speed is not the reason. Dictionaries are fast, and classes are not chosen for speed.
- **C** is wrong: a dictionary can hold as many keys as you like.
- **D** is wrong: the functions move into the class as methods. You still write them, with `def`.

Reread 16.1 *Why Classes: Bundling Data with Behavior* if you chose differently.

Question 2

```
True False False
40
```

`desk` and `hall` are two separate `Lamp` objects. Line 9 works out `not hall.on`, which is not `False`, so `desk.on` becomes `True`. `desk` is `hall` is `False` because each `Lamp()` call made a new object. Line 10 gives only `hall` a `watts` attribute, which line 12 prints; `desk.watts` would raise an `AttributeError`.

Reread 16.2 *Defining a Class and Making Objects* if your output differed.

Question 3

```
Rex dog 36
Tom cat 12
Mia cat 24
```

| Call | name | kind | age | age_in_months |
|-----------------------------------|-------|-----------------|-------------|---------------|
| <code>Pet("Rex", "dog", 3)</code> | "Rex" | "dog" | 3 | 36 |
| <code>Pet("Tom")</code> | "Tom" | "cat" (default) | 1 (default) | 12 |
| <code>Pet("Mia", age=2)</code> | "Mia" | "cat" (default) | 2 (keyword) | 24 |

`age` is a parameter, not an attribute: only `age_in_months` is stored on the object. The keyword argument `age=2` (Day 10) skips `kind`, which keeps its default.

Reread 16.3 *The Initializer and self* if your output differed.

Question 4

The traceback ends with:

```
AttributeError: 'Book' object has no attribute 'pages'
```

Line 4 is `pages = pages`, which only assigns the local parameter to itself. It never stores anything on the object, so `novel` has a `title` but no `pages`. Add `self.:`

book_pages_fixed.py

```

1 class Book:
2     def __init__(self, title, pages):
3         self.title = title
4         self.pages = pages
5
6
7 novel = Book("Holes", 233)
8 print(novel.title)
9 print(novel.pages)

```

```

Holes
233

```

Reread 16.3 *The Initializer and self* if you missed it.

Question 5

```

6
9
1

```

| Line | Call | clicks.value after | Printed |
|------|----------------|--------------------|---------|
| 13 | Counter(5) | 5 | |
| 14 | clicks.up() | 6 (default step 1) | 6 |
| 15 | clicks.up(3) | 9 | 9 |
| 16 | clicks.reset() | 0 | |
| 17 | clicks.up() | 1 | nothing |
| 18 | clicks.value | 1 | 1 |

Line 17 returns 1, but nothing prints it, so the return value is thrown away. Only the change to `value` remains.

Reread 16.4 *Methods* if your output differed.

Question 6

B. Calling a method through an object passes that object as `self`. The long form names the class and passes the object yourself.

- **A** is wrong: `add_points` lives inside the class, not loose in the file, so this is a `NameError`.
- **C** is wrong: nothing is passed for `self`, so 5 becomes `self` and `points` is missing, which raises a `TypeError`.
- **D** is wrong: `ana` is passed automatically **and** by hand, so the method gets three arguments for two parameters, a `TypeError`.

Reread 16.4 *Methods* if you chose differently.

Question 7

```

1 1 0

```

`self.sold += 1` means `self.sold = self.sold + 1`. The right side **reads** `self.sold`: the new ticket has none, so Python finds the class's 0, and adds 1. The assignment then stores 1 **on the ticket**, as a new instance attribute. Each ticket ends up with its own `sold` of 1, and `Ticket.sold` on the class never changes from 0. The fix is `Ticket.sold += 1`, which reads and writes the class attribute, and would print 2 2 2.

Reread 16.5 *Instance Attributes Versus Class Attributes* if your output differed.

Question 8

Model answer:

- **Route number: instance attribute.** Each bus runs its own route, so each object needs its own value, set in `__init__` from a parameter.
- **Fare: class attribute.** It is a city-wide rule shared by every bus. Storing it once on the class means one change to `Bus.fare` updates it for every bus.
- **Passengers: instance attribute,** made with `self.passengers = []` in `__init__`. Each bus has different people on board, and a list in the class body would be one list shared by every bus.

A correct answer bases each choice on whether the value differs per object or is shared on purpose, and notes that a changeable list must be made per object.

Reread 16.5 *Instance Attributes Versus Class Attributes* if your reasoning differed.

Question 9

```
Lions: 2
Now Lions: 3
8
```

Line 11 prints the text `__str__` returns. Line 12 changes the attribute, and `__str__` reads `self.points` every time it is called, so the f-string on line 13 gets the new value. `str(home)` is "Lions: 3": 5 letters, a colon, a space, and a digit make 8 characters.

Reread 16.6 *Printing an Object Readably* if your output differed.

Question 10

B. The output is:

```
Echo (12 plays)
[Song('Echo', 12), Song('Glow', 7)]
```

Printing a list shows each item's repr form, and `Song` defines `__repr__`.

- **A** is wrong: that would use `__str__` for the items, but lists always use `__repr__`. `__str__` is what the **first** line uses.
- **C** is wrong: `hits` is a list, so the output keeps the brackets and commas.
- **D** is wrong: that is what you would see with no `__repr__` at all.

Reread 16.7 *Showing an Object to a Programmer* if you chose differently.

Question 11

```
['ball'] ['kite']
True False True
```

There are **two** `Box` objects. `a` and `b` name the first one; `c` is the second, made by its own `Box(...)` call. `pack(b, "ball")` appends to the first box's list, so `a.items` shows it. `pack(c, "kite")` reaches only the second box.

`a is b` is `True` (one object). `a == c` is `False`: `Box` does not define equality, so `==` asks whether they are the same object, and they are not, even though both labels are `"toys"`.

`a.label == c.label` compares two strings, so it is `True`.

Reread *16.8 Objects Are References: Identity Versus Equality* if your output differed.

Question 12

Model answer: `backup = account` does not copy the object. It makes `backup` a second name for the same account, so when the balance changes through `account`, `backup.balance` shows the new value too, and nothing old is kept. To keep the old data, make a new object from it, such as `backup = BankAccount(account.owner, account.balance)`, or simply save the number: `old_balance = account.balance`. A number cannot change in place, so the saved value stays as it was.

A correct answer says that assignment makes an alias of the same object, not a copy, and makes a new object or saves the value itself.

Reread *16.8 Objects Are References: Identity Versus Equality* if your reasoning differed.

Question 13

One possible solution:

bank_account.py

```

1 class BankAccount:
2     def __init__(self, owner, balance=0):
3         if balance < 0:
4             raise ValueError(f"opening balance cannot be negative, not {balance}")
5         self.owner = owner
6         self.balance = balance
7
8     def deposit(self, amount):
9         if amount <= 0:
10            raise ValueError(f"deposit must be more than 0, not {amount}")
11            self.balance += amount
12
13    def withdraw(self, amount):
14        if amount <= 0:
15            raise ValueError(f"withdrawal must be more than 0, not {amount}")
16        if amount > self.balance:
17            raise ValueError(f"only {self.balance:.2f} available, not {amount:.2f}")
18        self.balance -= amount
19
20
21 account = BankAccount("Ana", 50)
22 account.deposit(25.5)
23 account.withdraw(30)
24 print(f"Balance: {account.balance:.2f}")
25 for amount in [100, -5]:
26     try:
27         account.withdraw(amount)
28     except ValueError as error:
29         print(f"Refused: {error}")
30 print(f"Balance: {account.balance:.2f}")

```

```

Balance: 45.50
Refused: only 45.50 available, not 100.00
Refused: withdrawal must be more than 0, not -5
Balance: 45.50

```

How it works: `__init__` stores the owner and balance as instance attributes, and refuses a negative opening balance. Each method checks its amount **before** changing anything, so a refused call leaves the balance exactly as it was: $50 + 25.5 - 30 = 45.5$, and the two bad withdrawals change nothing. Raising `ValueError` (Day 15) lets the caller decide what to tell the user, here with `except ValueError as error`. One honest limit: code outside the class can still write `account.balance = -5` directly. The rule holds as long as every change goes through the methods, so treat the methods as the only way in.

Reread 16.4 *Methods* if your result differed.

Question 14

Put the class in its own module, `task.py`, beside the others. `to_dict()` turns a task into the dictionary that JSON can save, because `json.dump` does not know how to save your objects:

todo_app/task.py

```

1  """The Task class for the to-do manager."""
2
3
4  class Task:
5      """One thing to do: a title, and whether it is done yet."""
6
7      def __init__(self, title, done=False):
8          self.title = title
9          self.done = done
10
11     def complete(self):
12         """Mark this task as done."""
13         self.done = True
14
15     def to_dict(self):
16         """Return this task as a dictionary, ready to save as JSON."""
17         return {"title": self.title, "done": self.done}
18
19
20 if __name__ == "__main__":
21     task = Task("Test the class")
22     print(task.title, task.done)
23     task.complete()
24     print(task.to_dict())

```

Run directly, its test block prints:

```

Test the class False
{'title': 'Test the class', 'done': True}

```

In tasks.py, every place that made or read a dictionary now uses a Task. Saving turns each task into a dictionary with a list comprehension (Day 11); loading builds a Task from each saved dictionary:

todo_app/tasks.py

```

1  """Task functions for the to-do manager. main.py imports them."""
2
3  import json
4
5  from task import Task
6
7
8  def show_menu():
9      """Print the menu of choices."""
10     print()
11     print("1) Add  2) List  3) Complete  4) Delete  5) Quit")
12
13
14 def add_task(tasks, title):
15     """Add a new, unfinished Task to the end of tasks."""
16     tasks.append(Task(title))
17
18

```

```

19 def list_tasks(tasks):
20     """Print every task with its number and a check box."""
21     if not tasks:
22         print("No tasks yet.")
23         return
24     for number, task in enumerate(tasks, 1):
25         mark = "x" if task.done else " "
26         print(f"{number}. [{mark}] {task.title}")
27
28
29 def complete_task(tasks, number):
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, number):
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, path):
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):
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["title"], item["done"]) for item in data]
73

```

```

74
75 if __name__ == "__main__":
76     test_tasks = []
77     add_task(test_tasks, "Test the module")
78     print(complete_task(test_tasks, 1).done)
79     print(complete_task(test_tasks, 2))
80     list_tasks(test_tasks)

```

In main.py, only one line changes: the message after completing or deleting reads `task.title` instead of `task["title"]`.

todo_app/main.py

```

1  """The to-do manager's menu loop. Run this file to start the program."""
2
3  from pathlib import Path
4
5  from tasks import (
6      add_task,
7      complete_task,
8      delete_task,
9      list_tasks,
10     load_tasks,
11     save_tasks,
12     show_menu,
13 )
14
15 SAVE_FILE = Path(__file__).parent / "tasks.json"
16
17
18 def main():
19     """Run the menu loop until the user chooses Quit."""
20     tasks = load_tasks(SAVE_FILE)
21     print("To-do list manager")
22     print(f"Loaded {len(tasks)} tasks from {SAVE_FILE.name}")
23     while True:
24         show_menu()
25         choice = input("Choose 1-5: ").strip()
26         if choice == "1":
27             title = input("Task title: ").strip()
28             if title:
29                 add_task(tasks, title)
30                 save_tasks(tasks, SAVE_FILE)
31                 print(f"Added: {title}")
32             else:
33                 print("A task needs a title.")
34         elif choice == "2":
35             list_tasks(tasks)
36         elif choice in ("3", "4"):
37             try:
38                 number = int(input("Task number: "))
39             except ValueError:
40                 print("Please type a task number, such as 1.")
41                 continue
42             if choice == "3":

```

```

43         task = complete_task(tasks, number)
44         action = "Completed"
45     else:
46         task = delete_task(tasks, number)
47         action = "Deleted"
48     if task is None:
49         print(f"There is no task {number}.")
50     else:
51         save_tasks(tasks, SAVE_FILE)
52         print(f"{action}: {task.title}")
53     elif choice == "5":
54         print("Goodbye!")
55         break
56     else:
57         print("Please choose a number from 1 to 5.")
58
59
60 if __name__ == "__main__":
61     main()

```

A run with the same typed answers as on Day 15 gives the same session, which is the point: the program behaves the same, but its data is now made of objects.

```

To-do list manager
Loaded 2 tasks from tasks.json

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 2
1. [x] Return library books
2. [ ] Practice piano

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 3
Task number: two
Please type a task number, such as 1.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 3
Task number: 2
Completed: Practice piano

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 4
Task number: 9
There is no task 9.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 5
Goodbye!

```

How it works: `complete_task` now calls `task.complete()` instead of setting a key, so "how a task becomes done" lives in one place, the class. A typo such as `task.dne` fails with an `AttributeError` at the line that has it. The JSON file is unchanged because `save_tasks` converts on the way out

and `load_tasks` converts on the way in; the rest of the program only ever sees `Task` objects. Notice that `load_tasks` still catches the same two exceptions as on Day 15, and the list comprehension that builds the tasks runs only after the try statement succeeds.

Reread 16.3 *The Initializer and self* and 16.4 *Methods* if your result differed.

Question 15

For the account, add this method to the class from question 13 (the rest of the class is unchanged):

bank_account_str.py

```
1 class BankAccount:
2     def __init__(self, owner, balance=0):
3         if balance < 0:
4             raise ValueError(f"opening balance cannot be negative, not {balance}")
5         self.owner = owner
6         self.balance = balance
7
8     def deposit(self, amount):
9         if amount <= 0:
10            raise ValueError(f"deposit must be more than 0, not {amount}")
11            self.balance += amount
12
13    def withdraw(self, amount):
14        if amount <= 0:
15            raise ValueError(f"withdrawal must be more than 0, not {amount}")
16        if amount > self.balance:
17            raise ValueError(f"only {self.balance:.2f} available, not {amount:.2f}")
18        self.balance -= amount
19
20    def __str__(self):
21        return f"{self.owner}'s account: {self.balance:.2f}"
22
23
24 accounts = [BankAccount("Ana", 50), BankAccount("Ben")]
25 accounts[0].deposit(25.5)
26 accounts[1].deposit(10)
27 accounts[1].withdraw(2.5)
28 for account in accounts:
29     print(account)
30 print(f"Most money: {max(accounts, key=lambda item: item.balance)}")
```

```
Ana's account: 75.50
Ben's account: 7.50
Most money: Ana's account: 75.50
```

For the task, `__str__` builds the same `[x]` or `[ ]` box that `list_tasks` used to build:

task_str.py

```

1 class Task:
2     def __init__(self, title, done=False):
3         self.title = title
4         self.done = done
5
6     def complete(self):
7         self.done = True
8
9     def __str__(self):
10        mark = "x" if self.done else " "
11        return f"[{mark}] {self.title}"
12
13
14 tasks = [Task("Return library books"), Task("Practice piano")]
15 tasks[0].complete()
16 for number, task in enumerate(tasks, 1):
17     print(f"{number}. {task}")

```

1. [x] Return library books
2. [] Practice piano

Copy the `__str__` method into `todo_app/task.py`. Then the loop in `list_tasks` shrinks to one line in its body, `print(f"{number}. {task}")`, exactly as in `task_str.py`, because the f-string calls `__str__`.

How it works: both methods **return** a string and print nothing. The last line of `bank_account_str.py` uses `max` with a key (Day 10) to pick the account with the largest balance, and the f-string then turns that account into text with `__str__`. Moving the check-box code into `Task` means any part of the program that prints a task shows it the same way.

Reread 16.6 *Printing an Object Readably* if your result differed.

Question 16

```

3 x Up = 30
1 x Coco = 6
4 10

```

- Line 17 makes evening with 3 seats, so `Ticket.sold` becomes 3.
- Line 18 changes the class attribute `price` to 10.
- Line 19 makes `matinee` with the default 1 seat: `Ticket.sold` becomes 4.
- Line 20 assigns through `matinee`, which creates an instance attribute `price` of 6 on `matinee` only.
- Line 21: `cost()` reads `self.price` **when it is called**, not when the ticket was made. `evening` has no price of its own, so it finds the class's current 10: $3 \times 10 = 30$.
- Line 22: `matinee.price` finds its own 6: $1 \times 6 = 6$.
- Line 23: `sold` is 4, and the class's `price` is still 10.

Reread 16.5 *Instance Attributes Versus Class Attributes* if your output differed.

Question 17

The traceback's last line is:

```
NameError: name 'name' is not defined. Did you mean: 'self.name'?
```

Inside `greet`, `name` is a plain variable `name`, and `greet` has no local variable or parameter called `name`. The parameter `name` belonged to `__init__` and disappeared when `__init__` finished; what survives is the attribute stored on the object. Methods reach the object's data through `self`:

`greeter_fixed.py`

```
1 class Greeter:
2     def __init__(self, name):
3         self.name = name
4
5     def greet(self):
6         return f"Hello, {self.name}!"
7
8
9 host = Greeter("Ana")
10 print(host.greet())
```

```
Hello, Ana!
```

Python 3.14 even suggests the fix; older versions may not show the "Did you mean" part.

Reread [16.3 The Initializer and self](#) and [16.4 Methods](#) if you missed it.

Question 18

B. Python first makes the empty object, then calls `__init__` with that object as `self` so it can store attributes, and only when the whole right side is done does the assignment attach the name `ana`.

- **A** is wrong: `__init__` needs the object as `self`, so the object must exist first.
- **C** is wrong: the right side of `=` is always worked out before the name is assigned (Day 2).
- **D** is wrong: `__init__` returns nothing; calling the class is what gives back the object. And `__init__` runs before the assignment.

Reread [16.3 The Initializer and self](#) if you chose differently.

Question 19

```
[Runner('Ben', 26), Runner('Ana', 31), Runner('Cy', 35)]
[Runner('Ana', 31), Runner('Ben', 26), Runner('Cy', 35)]
True False
```

`sorted` makes a **new list**, ordered by `time`, but the items in it are the **same three objects** (Day 6: copying a list never copies its items). `ranked[0]` is Ben's object, so `ranked[0].time -= 2` changes Ben's time to 26, and `runners` shows it too, in its original order. `ranked[0]` is `runners[1]` is `True`: both are Ben. `ranked == runners` compares the lists item by item; the first items are Ben and Ana, two different objects, and `Runner` has no equality of its own, so the lists are not equal. The lists print with `__repr__` for each item.

Reread [16.8 Objects Are References: Identity Versus Equality](#) if your output differed.

Question 20

Model plan:

- **Class attribute:** `limit = 5`, the same rule for every card.
- **Instance attributes** set in `__init__`: `owner` and `number` from parameters, and `borrowed`, a new empty list for each card.
- **Methods:** `borrow(title)` raises `ValueError` when the card is full and otherwise appends the title; `give_back(title)` removes it.
- **`__str__`:** the number, the owner, and the count of books out.

One class that follows this plan, with a short test:

library_card.py

```

1 class LibraryCard:
2     limit = 5
3
4     def __init__(self, owner, number):
5         self.owner = owner
6         self.number = number
7         self.borrowed = []
8
9     def borrow(self, title):
10        if len(self.borrowed) >= LibraryCard.limit:
11            raise ValueError(f"{self.owner} already has {LibraryCard.limit} books")
12        self.borrowed.append(title)
13
14    def give_back(self, title):
15        self.borrowed.remove(title)
16
17    def __str__(self):
18        count = len(self.borrowed)
19        return f"Card {self.number} ({self.owner}): {count} of {self.limit} books"
20
21
22 card = LibraryCard("Ana", 1042)
23 card.borrow("Holes")
24 card.borrow("Wonder")
25 card.give_back("Holes")
26 print(card)
27 print(card.borrowed)

```

```

Card 1042 (Ana): 1 of 5 books
['Wonder']

```

A correct answer finds the nouns in the description that differ per card (instance attributes), the one shared rule (class attribute), and the verbs (methods); makes the list in `__init__` so cards do not share it; and has `__str__` return, not print, its text.

Reread 16.1 *Why Classes: Bundling Data with Behavior* and 16.5 *Instance Attributes Versus Class Attributes* if your design differed.

Question 21

Model answer: `print(obj)` uses `__str__`, but printing a list shows each item in its repr form, and without `__repr__` that form is the default class name and memory address. Add `__repr__` first: when a class has no `__str__`, Python uses `__repr__` for `print` and `str()` too, so one `__repr__`

makes the object readable everywhere, in lists, in the REPL, and in `print`. Add `__str__` later only if users should see a friendlier form than the code-like one.

A correct answer says that containers use `__repr__` for their items and that `__repr__` also covers printing when `__str__` is missing.

Reread *16.7 Showing an Object to a Programmer* if your reasoning differed.