

Day 17 Classes and objects, part 2

Self-Study Guide · About 90 minutes · With answer key

By the end you will be able to

- Build a subclass that inherits from a class, extends `__init__` with `super()`, and overrides methods.
- Choose between inheritance and composition, and decide when a class is worth it and when a function and a dictionary are enough.
- Make `==`, `sorted`, and `len()` work on your own objects with `__eq__`, `__lt__`, and `__len__`.
- Use `@property` for computed attributes and `@dataclass` for classes that mostly hold data.

What to remember from earlier

- `class SomethingError(Exception)` followed by `pass` makes a new exception type. ([Day 15: Errors and exceptions, 15.8 Defining a Custom Exception](#))
- `__init__(self, ...)` runs for every new object, and `self.name = name` stores an attribute on that object. ([Day 16: Classes and objects, part 1, 16.3 The Initializer and self](#))
- `obj.method(args)` runs the method with `obj` as `self`, so `self.other()` calls another method on the same object. ([Day 16: Classes and objects, part 1, 16.4 Methods](#))
- For your own classes, `==` asks whether two names refer to the same object until you define equality yourself. ([Day 16: Classes and objects, part 1, 16.8 Objects Are References: Identity Versus Equality](#))
- `sorted(items, key=...)` sorts by the value the key function returns, and tuples compare item by item. ([Day 10: Functions, part 2, 10.8 Sorting with key](#))
- 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](#))
- Set items and dictionary keys must be hashable, which a list is not. ([Day 8: Dictionaries, 8.2 Which Types Can Be Keys](#))

Make a `day17` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days. Several of today's programs import Day 16's `BankAccount` class, so save the file `bank.py` below into `day17` first; the scripts that import it sit beside it, exactly like the modules you split up on Day 13.

bank/bank.py

```

1  """The BankAccount class from Day 16, ready to import."""
2
3
4  class BankAccount:
5      """An account with an owner and a balance that cannot go negative."""
6
7      def __init__(self, owner, balance=0):
8          if balance < 0:
9              raise ValueError(f"opening balance cannot be negative, not {balance}")
10         self.owner = owner
11         self.balance = balance
12
13     def deposit(self, amount):
14         if amount <= 0:
15             raise ValueError(f"deposit must be more than 0, not {amount}")
16         self.balance += amount
17
18     def withdraw(self, amount):
19         if amount <= 0:
20             raise ValueError(f"withdrawal must be more than 0, not {amount}")
21         if amount > self.balance:
22             raise ValueError(f"only {self.balance:.2f} available, not {amount:.2f}")
23         self.balance -= amount
24
25     def __str__(self):
26         return f"{self.owner}'s account: {self.balance:.2f}"

```

Yesterday you wrote classes from scratch. Today you learn the tools that make classes fit together and fit into Python. You will build a new class **on top of** an existing one instead of copying it, teach your objects to answer `==`, `<`, and `len()`, make attributes that work themselves out when you read them, and let Python write the routine parts of a class for you. The day ends with a question that matters as much as any syntax: when is a class worth writing at all?

17.1 Inheritance: Building on an Existing Class

The bank wants a savings account. A savings account is a bank account in every way (it has an owner and a balance, and it takes deposits and withdrawals with the same rules), plus one extra ability: it can add interest. With only Day 16's tools, you would copy the whole `BankAccount` class, rename it `SavingsAccount`, and add one method. Now there are two copies of the deposit and withdrawal rules. If you later find a bug in `withdraw`, you have to fix it twice, and sooner or later someone fixes only one copy.

You have actually done the better thing already. On Day 15 you wrote

```
class TaskNumberError(Exception):
```

and got a working exception type in two lines, without writing any of the code that lets an exception be raised, caught, and printed. The `(Exception)` part told Python: "make this new class a kind of `Exception`, and give it everything an `Exception` already has." That idea has a name.

Inheritance means defining a new class as an extended version of an existing one. The new class automatically has every method and class attribute of the existing class, and you add only what is

different. The existing class is called the **parent class** (also the **base class** or **superclass**), and the new one is the **child class** (also the **subclass**). The child **inherits** from the parent.

```
1 class SavingsAccount(BankAccount):
2     def add_interest(self, rate):
3         self.deposit(round(self.balance * rate, 2))
```

- `class SavingsAccount` starts a class definition as on Day 16.
- `(BankAccount)` in parentheses names the parent. Leave it out and you get an ordinary class with no `deposit`, `withdraw`, or anything else.
- The body holds only what is **new**: one method. `__init__`, `deposit`, `withdraw`, and `__str__` all come from `BankAccount`.

WORKED EXAMPLE A savings account that reuses everything

bank/savings.py

```
1 from bank import BankAccount
2
3
4 class SavingsAccount(BankAccount):
5     def add_interest(self, rate):
6         self.deposit(round(self.balance * rate, 2))
7
8
9 savings = SavingsAccount("Ana", 200)
10 savings.deposit(50)
11 savings.add_interest(0.02)
12 print(savings)
13 print(savings.owner, savings.balance)
14
15 plain = BankAccount("Ben", 40)
16 print(isinstance(savings, SavingsAccount), isinstance(savings, BankAccount))
17 print(isinstance(plain, SavingsAccount), isinstance(plain, object))
```

```
Ana's account: 255.00
Ana 255.0
True True
False True
```

- Line 1 imports the parent class from `bank.py` (Day 13).
- Line 9 calls `SavingsAccount("Ana", 200)`. `SavingsAccount` has no `__init__` of its own, so Python uses the parent's, which stores the owner and balance and checks that the balance is not negative.
- Line 10 calls `deposit`, which `SavingsAccount` also never defined: the balance becomes 250.
- Line 11 calls the new method. Inside it, `self` is the savings account, so `self.balance * rate` is $250 \times 0.02 = 5.0$, and `self.deposit(5.0)` uses the inherited `deposit`. The balance becomes 255.0.
- Line 12 prints with the parent's `__str__`.

When you call `savings.deposit(50)`, Python searches for `deposit` in a fixed order: first on the object itself, then in its class, `SavingsAccount`, then in the parent, `BankAccount`, where it finds it. The search stops at the first match.

Name looked up	On savings?	In SavingsAccount?	In BankAccount?	Found in
balance	yes			the object
add_interest	no	yes		SavingsAccount
deposit	no	no	yes	BankAccount
__str__	no	no	yes	BankAccount

Lines 15 to 17 introduce a built-in function you need with inheritance. `isinstance(obj, SomeClass)` returns `True` if `obj` is an instance of `SomeClass` **or of any child of it**. A child object counts as an instance of its parent, because a savings account really **is a** bank account. The reverse is not true: plain, an ordinary `BankAccount`, is not a `SavingsAccount`. That is why inheritance is often described as an **"is-a" relationship**: a `SavingsAccount` is a `BankAccount`, and a `TaskNumberError` is an `Exception`.

The last `True` on line 17 shows that even `BankAccount` has a parent. Every class you write without parentheses quietly inherits from a built-in class called `object`. That is where Day 16's defaults came from: the `<__main__.Player object at 0x...>` text and the "same object" meaning of `==` are methods that every class inherits from `object` until it defines its own.

Because a child is also an instance of its parent, code written for the parent keeps working for the child. Here is the same idea with the exception classes you already know. `AmountError` inherits from the built-in `ValueError`, so an `except ValueError` clause catches it too:

exception_family.py

```

1 class AmountError(ValueError):
2     pass
3
4
5 def check_amount(amount):
6     if amount <= 0:
7         raise AmountError(f"amount must be more than 0, not {amount}")
8     return amount
9
10
11 for text in ["25", "-5", "ten"]:
12     try:
13         print(check_amount(int(text)))
14     except ValueError as error:
15         print(f"Refused {text}: {error}")

```

```

25
Refused -5: amount must be more than 0, not -5
Refused ten: invalid literal for int() with base 10: 'ten'

```

`-5` raises an `AmountError`, which is a kind of `ValueError`, so `except ValueError` catches it. `"ten"` makes `int()` raise a plain `ValueError`, which the same clause catches. An `except` clause catches the type it names **and every child of that type**.

KEY IDEA

`class Child(Parent):` makes a class that inherits every method and class attribute of `Parent`; write only what is new. Python looks up a name on the object, then in its class, then in the parent. `isinstance(obj, Parent)` is `True` for instances of `Parent` and of its children, because a child "is a" parent.

WATCH OUT

Inheritance only goes down. A child gets the parent's methods, but the parent never gets the child's. This loop gives interest to every account in the list:

bank/one_way.py

```
1 from bank import BankAccount
2
3
4 class SavingsAccount(BankAccount):
5     def add_interest(self, rate):
6         self.deposit(round(self.balance * rate, 2))
7
8
9 accounts = [SavingsAccount("Ana", 200), BankAccount("Ben", 40)]
10 for account in accounts:
11     account.add_interest(0.02)
12     print(account)
```

Ana's account: 204.00

Traceback (most recent call last):

```
File ".../day17/one_way.py", line 11, in <module>
    account.add_interest(0.02)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

AttributeError: 'BankAccount' object has no attribute 'add_interest'

Ana's savings account worked. Ben's account is a plain `BankAccount`, and `add_interest` lives only in `SavingsAccount`, so the search (object, `BankAccount`, object) never finds it. Call a child-only method only on objects that are that child, for example by checking with `isinstance` first:

bank/one_way_fixed.py

```
1 from bank import BankAccount
2
3
4 class SavingsAccount(BankAccount):
5     def add_interest(self, rate):
6         self.deposit(round(self.balance * rate, 2))
7
8
9 accounts = [SavingsAccount("Ana", 200), BankAccount("Ben", 40)]
10 for account in accounts:
11     if isinstance(account, SavingsAccount):
12         account.add_interest(0.02)
13     print(account)
```

Ana's account: 204.00
Ben's account: 40.00

PAUSE AND TRY

Question 1 Tracing

What does this program print?

ticket_trace.py

```
1 class Ticket:
2     def __init__(self, seat, price):
3         self.seat = seat
4         self.price = price
5
6     def describe(self):
7         return f"Seat {self.seat}: {self.price}"
8
9
10 class VipTicket(Ticket):
11     def lounge(self):
12         return f"Seat {self.seat} may use the lounge"
13
14
15 regular = Ticket("B4", 12)
16 vip = VipTicket("A1", 30)
17 print(vip.describe())
18 print(vip.lounge())
19 print(isinstance(vip, Ticket), isinstance(regular, VipTicket))
```

Question 2 Multiple choice

Manager inherits from Employee. Employee defines a `pay()` method, and Manager adds an `approve()` method. `boss` is a Manager and `clerk` is an Employee. Which line raises an `AttributeError`?

- A. `boss.pay()`
- B. `boss.approve()`
- C. `clerk.approve()`
- D. `isinstance(boss, Employee)`

17.2 Extending `__init__` with `super()`

The savings account from 17.1 takes the interest rate as an argument every time. A real savings account has its own rate, agreed when it is opened, so the rate belongs in an instance attribute set in `__init__`. That means `SavingsAccount` needs an `__init__` of its own, with a `rate` parameter.

There is a catch. Defining a method in the child with the **same name** as one in the parent **replaces** it for the child: the name lookup from 17.1 finds the child's version first and stops. So the child's `__init__` runs **instead of** `BankAccount.__init__`, and the parent's work (checking the opening

balance, storing the owner and balance) would not happen. You do not want to copy that work into the child, which is the duplication inheritance was supposed to remove. You want the child's `__init__` to say "first do everything the parent's `__init__` does, then add the rate."

The built-in `super()` gives you exactly that. Inside a method, `super()` means "the parent class, working on this same object". `super().__init__(owner, balance)` calls the **parent's** `__init__` on the object being set up, with those arguments.

WORKED EXAMPLE A savings account with its own rate

bank/savings_rate.py

```
1 from bank import BankAccount
2
3
4 class SavingsAccount(BankAccount):
5     def __init__(self, owner, balance=0, rate=0.02):
6         super().__init__(owner, balance)
7         self.rate = rate
8
9     def add_interest(self):
10        self.deposit(round(self.balance * self.rate, 2))
11
12
13 savings = SavingsAccount("Ana", 200, rate=0.05)
14 savings.add_interest()
15 print(savings, savings.rate)
16 try:
17     SavingsAccount("Cy", -10)
18 except ValueError as error:
19     print(f"Refused: {error}")
```

Ana's account: 210.00 0.05

Refused: opening balance cannot be negative, not -10

Line 13, `SavingsAccount("Ana", 200, rate=0.05)`, does this:

Step	What Python does
1	Makes a new, empty <code>SavingsAccount</code> object
2	Finds <code>__init__</code> in <code>SavingsAccount</code> (the child's own), and calls it with <code>self</code> = the new object
3	Line 6: <code>super().__init__("Ana", 200)</code> runs <code>BankAccount.__init__</code> on the same object: checks 200, stores owner and balance
4	Back in the child, line 7 stores <code>rate = 0.05</code>
5	The object now has <code>owner</code> , <code>balance</code> , and <code>rate</code> , and line 13 names it <code>savings</code>

Line 14 adds interest using the stored rate: $200 \times 0.05 = 10.0$, so the balance becomes 210.0.

Notice what you do **not** pass: `self`. `super()` already knows which object it is working on, just as `savings.deposit(50)` passes `savings` for you.

The second half of the program shows why calling the parent matters. `SavingsAccount("Cy", -10)` runs the child's `__init__`, which calls the parent's, and the parent's check on line 8 of `bank.py`

refuses the negative balance. The rule "an opening balance cannot be negative" is written once, in `BankAccount`, and every kind of account obeys it.

You may also see the parent named directly, `BankAccount.__init__(self, owner, balance)`, which is Day 16's long form of a method call (and there you **do** pass `self`). It works, but `super()` is the usual choice: it does not repeat the parent's name, so the line stays correct if the parent class is ever renamed.

KEY IDEA

A child method with the same name as a parent method replaces it for the child. When a child needs its own `__init__`, call `super().__init__(...)` first, so the parent sets up its attributes and runs its checks, then add the child's own attributes.

WATCH OUT

Forgetting to call `super().__init__`. Without it, the parent's `__init__` never runs, and the attributes it would have made do not exist:

bank/forgot_super.py

```
1 from bank import BankAccount
2
3
4 class SavingsAccount(BankAccount):
5     def __init__(self, owner, balance=0, rate=0.02):
6         self.rate = rate
7
8     def add_interest(self):
9         self.deposit(round(self.balance * self.rate, 2))
10
11
12 savings = SavingsAccount("Ana", 200)
13 print(savings.rate)
14 savings.add_interest()
```

0.02

Traceback (most recent call last):

File ".../day17/forgot_super.py", line 14, in <module>

savings.add_interest()

~~~~~^

File ".../day17/forgot\_super.py", line 9, in add\_interest

self.deposit(round(self.balance \* self.rate, 2))

~~~~~

AttributeError: 'SavingsAccount' object has no attribute 'balance'

Making the account worked, and `rate` printed, so the mistake stays hidden until something reads an attribute the parent should have stored. When a child object is missing an attribute that the **parent's** `__init__` sets, look for a missing `super().__init__(...)` line. Add it as the first line of the child's `__init__`, as in `savings_rate.py`.

PAUSE AND TRY**Question 3** Tracing

What does this program print? Look closely at what line 12 passes to the parent.

manager_trace.py

```

1 class Employee:
2     def __init__(self, name, hourly):
3         self.name = name
4         self.hourly = hourly
5
6     def pay(self, hours):
7         return self.hourly * hours
8
9
10 class Manager(Employee):
11     def __init__(self, name, hourly, team):
12         super().__init__(name, hourly + 5)
13         self.team = team
14
15
16 lead = Manager("Ana", 20, ["Ben", "Cy"])
17 print(lead.name, lead.hourly, len(lead.team))
18 print(lead.pay(10))

```

Question 4 Debugging

This program should print beagle and then Rex the beagle. It prints beagle and then crashes. Find the bug and fix it.

dog_breed.py

```

1 class Pet:
2     def __init__(self, name):
3         self.name = name.title()
4
5
6 class Dog(Pet):
7     def __init__(self, name, breed):
8         self.breed = breed
9
10
11 rex = Dog("rex", "beagle")
12 print(rex.breed)
13 print(f"{rex.name} the {rex.breed}")

```

17.3 Overriding Methods and Polymorphism

Some accounts charge a fee: every withdrawal costs an extra 1.50. A fee account is still a bank account, so inheritance fits, but its withdrawal works **differently**. 17.2 showed that a child method with the same name as a parent method replaces it. Doing this on purpose, to change how a child behaves, is called **overriding** the method. The child's version is the **override**.

An override does not have to start from nothing. Inside it, `super().withdraw(...)` still reaches the parent's version, so the child can do its own part and hand the rest to the parent.

WORKED EXAMPLE An account that charges for withdrawals

bank/fee_account.py

```

1  from bank import BankAccount
2
3
4  class FeeAccount(BankAccount):
5      fee = 1.5
6
7      def withdraw(self, amount):
8          super().withdraw(amount + self.fee)
9
10     def __str__(self):
11         return f"{super().__str__()} (fee {self.fee:.2f} per withdrawal)"
12
13
14 accounts = [BankAccount("Ana", 50), FeeAccount("Ben", 50)]
15 for account in accounts:
16     account.withdraw(10)
17     print(account)

```

```

Ana's account: 40.00
Ben's account: 38.50 (fee 1.50 per withdrawal)

```

- Line 5 is a class attribute (Day 16): one fee shared by every fee account.
- Lines 7 and 8 override `withdraw`. The override adds the fee and passes the total to the parent's `withdraw`, which still checks the amount and the balance. So a fee account can never go negative either, and the checking code exists only once.
- Lines 10 and 11 override `__str__`. `super().__str__()` gets the parent's text, "Ben's account: 38.50", and the override adds a note after it.

Now look at the loop on lines 15 to 17. It treats every item the same way: `account.withdraw(10)`, then `print(account)`. It does not check which kind of account it has. Yet each object responds in its own way:

Account	Class	withdraw that runs	Balance after	__str__ that runs
Ana's	BankAccount	BankAccount.withdraw(10)	50 - 10 = 40.0	BankAccount's
Ben's	FeeAccount	FeeAccount.withdraw(10), which calls the parent's with 11.5	50 - 11.5 = 38.5	FeeAccount's

One line of code, `account.withdraw(10)`, ran two different methods, chosen by each object's class at the moment of the call. This is **polymorphism** (Greek for "many forms"): the same method call works on objects of different classes, and each object answers in its own way. You have relied on it since Day 1 without knowing: `print` works on ints, strings, lists, and your own objects, because each type has its own `__str__`, and `print` just asks.

Polymorphism lets you write a loop or a function once and use it with kinds of objects that did not exist when you wrote it. To plan for that, a parent class can define a method that **only** its children

are meant to fill in. The parent's version raises `NotImplementedError`, a built-in exception that means "this has to be provided by a child class":

notifiers.py

```
1 class Notifier:
2     def send(self, text):
3         raise NotImplementedError("each kind of notifier must define send()")
4
5
6 class EmailNotifier(Notifier):
7     def __init__(self, address):
8         self.address = address
9
10    def send(self, text):
11        return f"Email to {self.address}: {text}"
12
13
14 class TextNotifier(Notifier):
15     def __init__(self, phone):
16         self.phone = phone
17
18    def send(self, text):
19        return f"Text to {self.phone}: {text[:20]}"
20
21
22 class Banner:
23     def send(self, text):
24         return f"Banner: {text.upper()}"
25
26
27 notifiers = [EmailNotifier("ana@example.com"), TextNotifier("555-0142"), Banner()]
28 for notifier in notifiers:
29     print(notifier.send("Your library book is due tomorrow"))
```

```
Email to ana@example.com: Your library book is due tomorrow
Text to 555-0142: Your library book is
Banner: YOUR LIBRARY BOOK IS DUE TOMORROW
```

- `Notifier` (lines 1 to 3) is a parent that only says what every notifier can do: `send(text)`. Its own `send` refuses to run.
- `EmailNotifier` and `TextNotifier` each override `send`. A text message is short, so `TextNotifier` keeps only the first 20 characters with a slice (Day 3).
- `Banner` (lines 22 to 24) does **not** inherit from `Notifier` at all, yet the loop works with it too. Python never checks an object's class before calling a method; it only looks the name up. Any object that has a `send` method works in this loop. This style, caring about what an object can do rather than what class it is, is called **duck typing**, from the saying "if it walks like a duck and quacks like a duck, it is a duck."

Inheritance is still useful here: the parent documents what every notifier must provide, and its `NotImplementedError` catches a notifier that forgot.

KEY IDEA

Overriding means a child defines a method with the parent's name to change how it behaves; inside the override, `super().method(...)` still reaches the parent's version. Polymorphism means one call, such as `account.withdraw(10)`, runs whichever version belongs to each object's class.

WATCH OUT

A misspelled override is a new method, not an override. Python cannot know that `sned` was meant to be `send`, so it adds a second method and leaves the parent's `send` in place:

override_typo.py

```

1 class Notifier:
2     def send(self, text):
3         raise NotImplementedError("each kind of notifier must define send()")
4
5
6 class PopupNotifier(Notifier):
7     def sned(self, text):
8         return f"Popup: {text}"
9
10
11 popup = PopupNotifier()
12 print(popup.send("Practice starts at 4"))

```

```

Traceback (most recent call last):
  File ".../day17/override_typo.py", line 12, in <module>
    print(popup.send("Practice starts at 4"))
    ~~~~~^~~~~~
  File ".../day17/override_typo.py", line 3, in send
    raise NotImplementedError("each kind of notifier must define send()")
NotImplementedError: each kind of notifier must define send()

```

The traceback shows that the `send` that ran was on line 3, the **parent's**. This is the parent's `NotImplementedError` doing its job: without it, the parent's version would run silently and you might never notice. When an override seems to be ignored, compare its name with the parent's letter by letter. Rename `sned` to `send`.

PAUSE AND TRY**Question 5** Tracing

What does this program print? `label` is defined only in `Price`.

price_trace.py

```
1 class Price:
2     def __init__(self, amount):
3         self.amount = amount
4
5     def final(self):
6         return self.amount
7
8     def label(self):
9         return f"{self.final():.2f}"
10
11
12 class Sale(Price):
13     def final(self):
14         return self.amount * 0.8
15
16
17 class Clearance(Sale):
18     def final(self):
19         return super().final() - 1
20
21
22 for price in [Price(10), Sale(10), Clearance(10)]:
23     print(price.label())
```

Question 6 Multiple choice

Notifier.send raises NotImplementedError, as in notifiers.py. A new class is written as `class PushNotifier(Notifier):` with a body of just `pass`. What happens when a program runs `PushNotifier().send("Hi")`?

- A. It returns `None`, because `PushNotifier` has no `send` method.
- B. It raises `NotImplementedError`, from `Notifier`'s `send`.
- C. It raises `AttributeError`, because `PushNotifier` has no `send` method.
- D. It prints `Hi`.

17.4 Composition Versus Inheritance

Inheritance is not the only way to build a class out of other classes. Think about a bank branch. It holds many accounts, but a branch is not an account; you would never deposit money "into the branch" the way you deposit into an account. The branch **has** accounts. When one object keeps other objects as its attributes and uses them to do its job, that is **composition**, a "**has-a**" **relationship**. You have already done it: Day 16's `Cart` kept a dictionary of prices, and its library kept lists of `Book` objects.

WORKED EXAMPLE A branch that has accounts

bank/branch.py

```

1  from bank import BankAccount
2
3
4  class Branch:
5      def __init__(self, city):
6          self.city = city
7          self.accounts = []
8
9      def open_account(self, owner, deposit):
10         account = BankAccount(owner, deposit)
11         self.accounts.append(account)
12         return account
13
14     def total(self):
15         return sum(account.balance for account in self.accounts)
16
17
18 branch = Branch("Springfield")
19 ana = branch.open_account("Ana", 50)
20 branch.open_account("Ben", 20)
21 ana.deposit(30)
22 print(f"{branch.city}: {len(branch.accounts)} accounts, {branch.total():.2f} in
    total")
23 print(ana)

```

```

Springfield: 2 accounts, 100.00 in total
Ana's account: 80.00

```

- Line 7 gives each branch its own list of accounts. `Branch` inherits from nothing (other than object): it is built **from** accounts, not **as** an account.
- `open_account` makes a `BankAccount`, keeps it in the list, and returns it, so the caller can use it too. Line 19 names it `ana`; the list and `ana` refer to the same object (Day 16), so the deposit on line 21 shows up in the branch total.
- `total` asks each account for its balance: $80 + 20 = 100$. When an object hands part of its job to an object it holds, that is called **delegating**.

Why not always inherit, since it saves typing? Because a child gets **everything** the parent has, wanted or not. Here is a playlist that must never hold the same song twice. Inheriting from the built-in `List` class looks convenient (Python's built-in types are classes, so you can inherit from them):

playlist_inherit.py

```

1 class Playlist(list):
2     def add(self, song):
3         if song in self:
4             raise ValueError(f"{song} is already on the playlist")
5         self.append(song)
6
7
8 road_trip = Playlist()
9 road_trip.add("Holiday")
10 road_trip.add("Sunrise")
11 road_trip.append("Holiday")
12 road_trip.insert(0, 42)
13 print(road_trip)

```

```
[42, 'Holiday', 'Sunrise', 'Holiday']
```

add refuses duplicates, but a `Playlist` **is a** list, so it also has `append`, `insert`, `extend`, and every other list method. Line 11 adds a duplicate with `append`, and line 12 even puts a number in front with `insert`. The rule lives in one method, and a dozen other methods walk around it. Composition gives the playlist a list instead, and offers only the operations that keep the rule:

playlist_compose.py

```

1 class Playlist:
2     def __init__(self, name):
3         self.name = name
4         self.songs = []
5
6     def add(self, song):
7         if song in self.songs:
8             raise ValueError(f"{song} is already on {self.name}")
9         self.songs.append(song)
10
11     def __str__(self):
12         return f"{self.name}: {' '.join(self.songs)}"
13
14
15 road_trip = Playlist("Road trip")
16 road_trip.add("Holiday")
17 road_trip.add("Sunrise")
18 try:
19     road_trip.add("Holiday")
20 except ValueError as error:
21     print(f"Not added: {error}")
22 print(road_trip)
23 try:
24     road_trip.append("Holiday")
25 except AttributeError as error:
26     print(error)

```

```
Not added: Holiday is already on Road trip
Road trip: Holiday, Sunrise
'Playlist' object has no attribute 'append'
```

Now `add` is the only way in (line 6), so the rule always holds, and `append` simply does not exist on a playlist (line 24). The list is still there doing the storing, in `self.songs`; the playlist decides what the outside world may do with it.

Two questions help you choose:

Question	If yes	Example
Is the new thing truly a kind of the old thing ("a savings account is a bank account")?	Inheritance may fit	<code>SavingsAccount(BankAccount)</code>
Does every method of the old thing make sense for the new thing?	Inheritance may fit	A fee account still deposits and withdraws
Does the new thing just hold or use the old thing ("a branch has accounts")?	Composition	<code>self.accounts = []</code>

Inheritance needs both of the first two answers to be yes. When you are unsure, choose composition. It is the safer default: it keeps the classes separate, so changing one does not quietly change the other, and the outer class shows only the methods you chose.

KEY IDEA

Inheritance is "is-a": the child is a kind of the parent and gets all its methods. Composition is "has-a": an object keeps other objects as attributes and delegates work to them. Prefer composition unless the new class truly is a kind of the old one and every inherited method makes sense for it.

PAUSE AND TRY

Question 7 Multiple choice

Which pair is the best fit for inheritance, with the first class inheriting from the second?

- A. `Car` and `Engine`
- B. `Classroom` and `Student`
- C. `Square` and `Shape`
- D. `Order` and `List`

Question 8 Short answer

A team may have at most 11 players. A classmate writes `class Team(list):` with an `add_player` method that refuses a twelfth player. Explain why this design cannot keep the rule, and describe a design that can.

17.5 Comparing with `__eq__`

Day 16 ended with a puzzle: two `Player` objects with the same name and score were not `==`, because for your own classes `==` starts out meaning "same object". You now know where that meaning comes from: `==` is a method that every class inherits from `object`. And 17.3 showed that a child can override any inherited method. So you can override equality.

When Python works out `a == b`, it calls `a.__eq__(b)` and uses what the method returns. `__eq__` is another dunder method (Day 16): you write it, and Python calls it for you whenever `==` is used. It takes `self` and one more parameter, by convention named `other`, for the object on the right of `==`. It returns `True` or `False`.

You decide what "equal" means. For points on a game board, two points are equal when they are at the same place: same `x` and same `y`.

WORKED EXAMPLE Points that are equal when they are at the same place

point_eq.py

```
1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __eq__(self, other):
7         return self.x == other.x and self.y == other.y
8
9     def __repr__(self):
10        return f"Point({self.x}, {self.y})"
11
12
13 start = Point(2, 3)
14 same_spot = Point(2, 3)
15 print(start == same_spot, start is same_spot)
16 print(start != Point(3, 2))
17 visited = [Point(0, 0), Point(2, 3), Point(5, 1)]
18 print(Point(2, 3) in visited, visited.index(Point(5, 1)))
19 visited.remove(Point(0, 0))
20 print(visited)
```

```
True False
True
True 2
[Point(2, 3), Point(5, 1)]
```

- Line 15: `start == same_spot` calls `start.__eq__(same_spot)`, which compares 2 with 2 and 3 with 3 and returns `True`. They are still two separate objects, so `is` is `False`. Equality and identity have come apart, exactly as for two equal lists on Day 4.
- Line 16: Python works out `!=` as "not `==`" when a class defines only `__eq__`. `Point(3, 2)` is a different place, so `!=` is `True`.
- Line 18: `in` and the list method `index` both search with `==` (Day 6), so they now find a point with the same coordinates even though it is a brand-new object. `Point(5, 1)` is at index 2.
- Line 19: `remove` also searches with `==`, so it removes the point at (0, 0).

Equality does not have to use every attribute. A library may own two copies of the same book, one new and one worn. For the catalog, they are the same book, because they have the same catalog

code. So `__eq__` compares only the code. This version also handles a question the first one ignored: what if the other value is not a `Book` at all? Look at what the first `Point` does when compared with a tuple:

point_tuple.py

```
1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __eq__(self, other):
7         return self.x == other.x and self.y == other.y
8
9
10 print(Point(2, 3) == Point(2, 3))
11 print(Point(2, 3) == (2, 3))
```

True

Traceback (most recent call last):

```
File ".../day17/point_tuple.py", line 11, in <module>
    print(Point(2, 3) == (2, 3))
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
File ".../day17/point_tuple.py", line 7, in __eq__
    return self.x == other.x and self.y == other.y
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

AttributeError: 'tuple' object has no attribute 'x'

Asking "is this point equal to that tuple?" should simply be `False`, not a crash. The careful pattern checks the other value's type first:

book_eq.py

```
1 class Book:
2     def __init__(self, title, code, condition):
3         self.title = title
4         self.code = code
5         self.condition = condition
6
7     def __eq__(self, other):
8         if not isinstance(other, Book):
9             return NotImplemented
10        return self.code == other.code
11
12
13 copy_1 = Book("Holes", "B-04", "new")
14 copy_2 = Book("Holes", "B-04", "worn")
15 wonder = Book("Wonder", "B-17", "new")
16 print(copy_1 == copy_2, copy_1 == wonder)
17 print(copy_1 == "Holes", copy_1 != (2, 3))
```

True False

False True

- Lines 8 and 9: if `other` is not a `Book`, the method returns `NotImplemented`, a special built-in value that means "I do not know how to compare with that." It is not an error. Python then gives the `other` value a chance to compare itself, and if neither side knows, `==` falls back to identity, which is `False` for two different objects.
- Line 16: the two copies of `Holes` have different conditions but the same code, so they are equal. `Wonder` has a different code.
- Line 17: comparing a book with a string or a tuple now gives `False` for `==` and `True` for `!=`, with no crash.

`NotImplemented` (a value you return) is not the same thing as `NotImplementedError` from 17.3 (an exception you raise). Only `__eq__` and its relatives return `NotImplemented`.

KEY IDEA

Define `__eq__(self, other)` to decide what `==` means for your class; `!=`, `in`, `index`, `count`, and `remove` all use it. Start it with

```
if not isinstance(other, YourClass): return NotImplemented, so comparing with another
type gives False instead of a crash.
```

WATCH OUT

A class with `__eq__` cannot go in a set. Day 8 explained that set items and dictionary keys must be hashable: Python files them by a number worked out from the value. Equal values must get the same number, and Python cannot know how to work one out from your `__eq__`, so when a class defines `__eq__`, Python makes its objects unhashable:

point_set.py

```
1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __eq__(self, other):
7         return self.x == other.x and self.y == other.y
8
9
10 visited = [Point(0, 0), Point(2, 3)]
11 print(len(visited))
12 seen = {Point(0, 0), Point(2, 3)}
```

2

Traceback (most recent call last):

```
File ".../day17/point_set.py", line 12, in <module>
    seen = {Point(0, 0), Point(2, 3)}
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: cannot use 'Point' as a set element (unhashable type: 'Point')
```

A list of points works (line 11); a set of them does not. Older versions of Python print just `TypeError: unhashable type: 'Point'`. When you need a set, store something hashable that stands for each object, such as a tuple of its coordinates (Day 7):

point_set_fixed.py

```

1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __eq__(self, other):
7         return self.x == other.x and self.y == other.y
8
9
10 visited = [Point(0, 0), Point(2, 3), Point(0, 0)]
11 seen = {(point.x, point.y) for point in visited}
12 print(len(seen), (0, 0) in seen)

```

2 True

Three points went in but the set has 2 items, because the two `(0, 0)` tuples are equal and a set keeps only one.

PAUSE AND TRY

Question 9 Tracing

What does this program print? Coins are equal when they have the same value, whatever their year.

coin_trace.py

```

1 class Coin:
2     def __init__(self, value, year):
3         self.value = value
4         self.year = year
5
6     def __eq__(self, other):
7         if not isinstance(other, Coin):
8             return NotImplemented
9         return self.value == other.value
10
11
12 purse = [Coin(25, 2019), Coin(10, 2020), Coin(25, 2024)]
13 print(Coin(25, 1999) in purse)
14 print(purse.count(Coin(25, 2000)), purse.index(Coin(10, 1990)))
15 print(purse[0] == purse[2], purse[0] is purse[2], purse[0] == 25)

```

Question 10 Debugging

This program should print `True`, because both cards are the queen of hearts. It prints `None`. Find the bug and fix it.

card_eq.py

```

1 class Card:
2     def __init__(self, rank, suit):
3         self.rank = rank
4         self.suit = suit
5
6     def __eq__(self, other):
7         self.rank == other.rank and self.suit == other.suit
8
9
10 print(Card("Q", "hearts") == Card("Q", "hearts"))

```

17.6 Ordering with `__lt__` and Size with `__len__`

Ask Python to sort a list of your objects, and it does not know how:

runners_no_lt.py

```

1 class Runner:
2     def __init__(self, name, time):
3         self.name = name
4         self.time = time
5
6
7 runners = [Runner("Ana", 31), Runner("Ben", 28), Runner("Cy", 35)]
8 print(sorted(runners))

```

```

Traceback (most recent call last):
  File ".../day17/runners_no_lt.py", line 8, in <module>
    print(sorted(runners))
    ~~~~~^~~~~~
TypeError: '<' not supported between instances of 'Runner' and 'Runner'

```

The message gives away how sorting works: `sorted` compares items two at a time with `<`. Day 10 worked around this with `key=`, telling `sorted` to compare a value taken from each item. That is still the right tool for a one-off order. But if your objects have one natural order (runners by finishing time, versions by number), you can teach the class itself what `<` means. `a < b` calls `a.__lt__(b)` ("lt" is short for "less than"), and `__lt__` returns `True` if `self` should come before `other`.

WORKED EXAMPLE Runners that sort by time

runners_sort.py

```

1 class Runner:
2     def __init__(self, name, time):
3         self.name = name
4         self.time = time
5
6     def __lt__(self, other):
7         return self.time < other.time
8
9     def __repr__(self):
10        return f"Runner({self.name!r}, {self.time})"
11
12
13 runners = [Runner("Ana", 31), Runner("Ben", 28), Runner("Cy", 35)]
14 print(sorted(runners))
15 print(min(runners), max(runners))
16 print(runners[0] < runners[1], runners[0] > runners[1])
17 runners.sort(reverse=True)
18 print(runners)

```

```

[Runner('Ben', 28), Runner('Ana', 31), Runner('Cy', 35)]
Runner('Ben', 28) Runner('Cy', 35)
False True
[Runner('Cy', 35), Runner('Ana', 31), Runner('Ben', 28)]

```

- Line 14: `sorted` now works with no `key`, putting the fastest time first. Each comparison it makes, such as "is Ana before Ben?", calls `Ana.__lt__(Ben)`, which works out `31 < 28`.
- Line 15: `min` and `max` compare with `<` too, so they find the fastest and the slowest runners.
- Line 16: `runners[0] < runners[1]` asks whether `31 < 28`, which is `False`. The class has no "greater than" method, yet `>` works: when the left side cannot answer `a > b`, Python flips it into `b < a` and asks the right side's `__lt__`. Ben's `28 < 31` is `True`.
- Line 17: `sort(reverse=True)` (Day 6) uses the same `__lt__` and reverses the result.

The lists print with `__repr__` for each item (Day 16).

`__lt__` gives you `<` and `>`, but not `<=` or `>=`. Those are separate methods (`__le__` and `__ge__`), and this class does not define them:

runner_le.py

```

1 class Runner:
2     def __init__(self, name, time):
3         self.name = name
4         self.time = time
5
6     def __lt__(self, other):
7         return self.time < other.time
8
9
10 ana = Runner("Ana", 31)
11 ben = Runner("Ben", 28)
12 print(ben < ana)
13 print(ben <= ana)

```

True

Traceback (most recent call last):

File ".../day17/runner_le.py", line 13, in <module>

print(ben <= ana)

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

TypeError: '<=' not supported between instances of 'Runner' and 'Runner'

For sorting, `min`, and `max`, `__lt__` is all you need. 17.8 shows a way to get all four comparisons at once.

The other special method today answers "how big is it?" `len(obj)` calls `obj.__len__()`, and `__len__` must return a whole number of 0 or more. It makes sense for objects that hold a collection of things, like the playlist from 17.4:

playlist_len.py

```

1 class Playlist:
2     def __init__(self, name):
3         self.name = name
4         self.songs = []
5
6     def add(self, song):
7         self.songs.append(song)
8
9     def __len__(self):
10        return len(self.songs)
11
12
13 road_trip = Playlist("Road trip")
14 print(len(road_trip), bool(road_trip))
15 if not road_trip:
16     print("Road trip is empty")
17 road_trip.add("Holiday")
18 road_trip.add("Sunrise")
19 print(len(road_trip), bool(road_trip))

```

0 False

Road trip is empty

2 True

`__len__` delegates to the list inside (composition again): the playlist's length is the number of songs. Defining `__len__` has a second effect. Day 4 taught that an empty string or list counts as `False` in an `if`. Python decides that for your objects too, using `__len__`: an object whose length is 0 is falsy. So `bool(road_trip)` is `False` while the playlist is empty, and `if not road_trip:` reads like English. After two songs it is truthy.

Here are the special methods so far, and what triggers each:

You write	Python calls
<code>a == b</code> , <code>a != b</code>	<code>a.__eq__(b)</code> (<code>!=</code> flips its answer)
<code>a < b</code>	<code>a.__lt__(b)</code>
<code>a > b</code>	<code>b.__lt__(a)</code> , when the class has only <code>__lt__</code>
<code>sorted</code> , <code>min</code> , <code>max</code> , <code>sort</code>	<code>__lt__</code> , many times
<code>len(a)</code>	<code>a.__len__()</code>
<code>if a:</code> , <code>bool(a)</code>	<code>a.__len__()</code> , and checks whether it is 0
<code>print(a)</code> , <code>str(a)</code>	<code>a.__str__()</code> (Day 16)

KEY IDEA

Define `__lt__(self, other)` so `<`, `>`, `sorted`, `min`, and `max` work on your objects with no key. Define `__len__(self)` to return a count, so `len(obj)` works and an empty object is falsy.

WATCH OUT

`__len__` must return a whole number. It is tempting to use `len` for any measurement, such as a playlist's length in minutes. But `len()` means "how many", and Python refuses anything that is not a whole number:

len_float.py

```

1 class Playlist:
2     def __init__(self, name):
3         self.name = name
4         self.seconds = []
5
6     def add(self, seconds):
7         self.seconds.append(seconds)
8
9     def __len__(self):
10        return sum(self.seconds) / 60
11
12
13 road_trip = Playlist("Road trip")
14 road_trip.add(200)
15 road_trip.add(250)
16 print(len(road_trip))

```

```
Traceback (most recent call last):
  File ".../day17/len_float.py", line 16, in <module>
    print(len(road_trip))
          ~~~^^^^^^^^^^^^^^^^
TypeError: 'float' object cannot be interpreted as an integer
```

450 seconds / 60 is 7.5, a float, and the error appears at the call to `len`. Keep `__len__` for counting the items an object holds (here, `len(self.seconds)`), and give other measurements their own name, such as a `minutes` method or the computed attribute you meet in 17.7.

PAUSE AND TRY

Question 11 Tracing

What does this program print? Boxes compare by weight, and a box's length is the length of its label.

box_trace.py

```
1 class Box:
2     def __init__(self, label, weight):
3         self.label = label
4         self.weight = weight
5
6     def __lt__(self, other):
7         return self.weight < other.weight
8
9     def __len__(self):
10        return len(self.label)
11
12    def __repr__(self):
13        return self.label
14
15
16 boxes = [Box("books", 12), Box("lamp", 4), Box("kettle", 9)]
17 print(sorted(boxes))
18 print(max(boxes), len(boxes), len(boxes[0]))
19 print(sorted(boxes, key=len))
```

Question 12 Multiple choice

A `Cart` class defines `__len__` to return the number of items in the cart. `cart` has no items yet. What does `print("full" if cart else "empty")` print?

- A. `full`, because every object counts as true.
- B. `empty`, because `cart` has length 0, so it is falsy.
- C. `full`, because `cart` is not `None`.
- D. It raises a `TypeError`, because a `cart` is not a `bool`.

17.7 Computed Attributes with `@property`

A rectangle's area depends on its width and height. A natural first try stores it in `__init__`, next to them:

stale_area.py

```
1 class Rectangle:
2     def __init__(self, width, height):
3         self.width = width
4         self.height = height
5         self.area = width * height
6
7
8 room = Rectangle(4, 3)
9 print(room.area)
10 room.width = 5
11 print(room.width, room.area)
```

```
12
13 5 12
```

The width changed to 5, but the area still says 12, which is now wrong (5×3 is 15). `self.area` was worked out once, when the object was made, and nothing updates it. A value copied from other attributes goes **stale** as soon as they change.

A method fixes that, because it works the area out each time: `room.area()`. But the area feels like a fact about the rectangle, like its width, not like an action. Python lets you have both: a method that is **read like an attribute**. Put the line `@property` directly above the method's `def`:

```
1 @property
2 def area(self):
3     return self.width * self.height
```

A line that starts with `@` and sits right above a `def` is a **decorator**: it changes what the `def` makes. Day 18 shows how decorators work and how to write your own. Today you only use two that come with Python. `@property` turns the method into a **property**: an attribute whose value is computed by running the method **each time** the attribute is read. You read it with no parentheses, `room.area`, and Python calls the method for you, the same way it calls `__str__` for `print`.

WORKED EXAMPLE An area that is always up to date

rectangle_property.py

```

1 class Rectangle:
2     def __init__(self, width, height):
3         self.width = width
4         self.height = height
5
6     @property
7     def area(self):
8         return self.width * self.height
9
10    @property
11    def is_square(self):
12        return self.width == self.height
13
14
15 room = Rectangle(4, 3)
16 print(room.area, room.is_square)
17 room.width = 3
18 print(room.area, room.is_square)
19 try:
20     room.area = 100
21 except AttributeError as error:
22     print(error)

```

```

12 False
9 True
property 'area' of 'Rectangle' object has no setter

```

Line	What happens
16	Reading <code>room.area</code> runs the <code>area</code> method: $4 \times 3 = 12$. <code>is_square</code> runs too: $4 == 3$ is <code>False</code>
17	Only <code>width</code> is stored, so only <code>width</code> changes
18	Reading the properties again runs the methods again: $3 \times 3 = 9$, and $3 == 3$ is <code>True</code>
20	Assigning to <code>room.area</code> raises an <code>AttributeError</code> , which line 22 prints

Line 20 shows that a property is **read-only** unless you do more work. That is right for a computed value: setting the area of a rectangle directly makes no sense, because the area must always match the width and height. A property whose name starts with `is_` and answers yes or no, like `is_square`, is a common, readable pattern. Older versions of Python word the last message differently (for example `can't set attribute`).

Being read-only solves a problem from Day 16. The bank account's rules hold only if every change goes through `deposit` and `withdraw`, but any code could still write `account.balance = -5`. With a property, outside code can **read** the balance but not assign it:

account_property.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     @property
9     def balance(self):
10         return self._balance
11
12     def deposit(self, amount):
13         if amount <= 0:
14             raise ValueError(f"deposit must be more than 0, not {amount}")
15         self._balance += amount
16
17
18 account = BankAccount("Ana", 50)
19 account.deposit(25)
20 print(account.balance)
21 try:
22     account.balance = -5
23 except AttributeError as error:
24     print(f"Refused: {error}")
25 print(account.balance)

```

75

Refused: property 'balance' of 'BankAccount' object has no setter

75

- Line 6 stores the real number in an attribute named `_balance`. A name that starts with one underscore is a convention, not a rule: it tells other programmers "this is internal; use it only inside the class." Python does not stop anyone from reading or changing `_balance`, but anyone who does has been warned.
- Lines 8 to 10 make `balance` a property that just returns `_balance`. Outside code reads `account.balance` exactly as before.
- Line 15: the methods change `_balance`, the internal attribute.
- Line 22: assigning `account.balance` is refused, so the balance can change only through the methods, where the checks are.

When should something be a property rather than a method? Use a property for a value that describes the object, takes no arguments, is quick to work out, and changes nothing (area, `is_square`, a full name built from first and last names). Use a method for an action (`deposit`), for anything that needs arguments (`withdraw(amount)`), or for slow work, because a reader expects reading an attribute to be instant.

KEY IDEA

`@property` above a method makes it a computed attribute: reading `obj.name` with no parentheses runs the method every time, so the value is never stale, and assigning to it raises an `AttributeError`. Pair it with an underscore attribute, such as `_balance`, to let outside code read a value but not set it.

WATCH OUT

Calling a property with parentheses. After `@property`, `room.area` is already the number, so `room.area()` tries to call the number:

property_call.py

```
1 class Rectangle:
2     def __init__(self, width, height):
3         self.width = width
4         self.height = height
5
6     @property
7     def area(self):
8         return self.width * self.height
9
10
11 room = Rectangle(4, 3)
12 print(room.area())
```

```
Traceback (most recent call last):
  File "../day17/property_call.py", line 12, in <module>
    print(room.area())
           ~~~~~~^^
TypeError: 'int' object is not callable
```

"Not callable" means "you put `()` after something that is not a function". Reading `room.area` gave 12, and then `12()` failed. When you see this message on a line that uses a property, drop the parentheses: `print(room.area)`.

PAUSE AND TRY**Question 13** Tracing

What does this program print?

thermometer_trace.py

```

1 class Thermometer:
2     def __init__(self, celsius):
3         self.celsius = celsius
4
5     @property
6     def fahrenheit(self):
7         return self.celsius * 9 / 5 + 32
8
9     @property
10    def is_freezing(self):
11        return self.celsius <= 0
12
13
14 reading = Thermometer(25)
15 print(reading.fahrenheit, reading.is_freezing)
16 reading.celsius = -10
17 print(reading.fahrenheit, reading.is_freezing)

```

Question 14 Multiple choice

A Student class stores `first` and `last` names. Which of these is the best fit for a property?

- A. `enroll(course)`, which adds a course to the student's list
- B. `full_name`, which returns the first and last names joined by a space
- C. `save(path)`, which writes the student to a JSON file
- D. `add_grade(score)`, which appends a score

17.8 Data Classes with @dataclass

Many classes mostly hold data. A point, a song, a task, a grade: each has a few attributes and not much behavior. Yet a well-behaved data-holding class needs the same routine methods every time. Here is a point written by hand with `__init__`, a `__repr__` in the Day 16 style, and a careful `__eq__` from 17.5:

point_by_hand.py

```

1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __repr__(self):
7         return f"Point(x={self.x!r}, y={self.y!r})"
8
9     def __eq__(self, other):
10        if not isinstance(other, Point):
11            return NotImplemented
12        return self.x == other.x and self.y == other.y
13
14
15 print(Point(2, 3), Point(2, 3) == Point(2, 3))

```

```
Point(x=2, y=3) True
```

Twelve lines for two attributes, and `x` and `y` each appear six times. Add a field and you must update three methods; forget one and the class is quietly wrong. Python can write these methods for you. The module `dataclasses` in the standard library provides a decorator, `@dataclass`, that goes above a **class** line:

WORKED EXAMPLE The same point as a data class

point_dataclass.py

```
1 from dataclasses import dataclass
2
3
4 @dataclass
5 class Point:
6     x: int
7     y: int
8
9
10 start = Point(2, 3)
11 print(start)
12 print(start == Point(2, 3), start.x + start.y)
13 print(Point(y=1, x=4))
14 print(Point("two", "three"))
```

```
Point(x=2, y=3)
True 5
Point(x=4, y=1)
Point(x='two', y='three')
```

The class body (lines 6 and 7) lists the class's **fields**: the attributes every object will have, one per line, in the form `name: type`. The part after the colon is a **type annotation**: a note saying what kind of value the field is meant to hold. A class decorated with `@dataclass` is called a **data class**. From the fields, `@dataclass` writes three methods for you:

Generated method	What it does	Line that uses it
<code>__init__</code>	Takes one parameter per field, in order, and stores each one	10, 13
<code>__repr__</code>	Shows the class name and every field, like <code>Point(x=2, y=3)</code>	11
<code>__eq__</code>	Equal when both are the same class and every field is equal	12

- Line 10: `Point(2, 3)` passes `x` and `y` in field order. Line 13 shows that the generated `__init__` takes keyword arguments too (Day 10).
- Line 11: `print` has no `__str__` to use, so it uses the generated `__repr__` (Day 16).
- Line 12: `==` compares the fields, and the fields are ordinary attributes: `start.x + start.y` is 5.
- Line 14: Python does **not** check annotations. `x: int` documents what `x` should hold, but `Point("two", "three")` is accepted. Day 19 shows tools that do check them.

A data class is still a class. You can give fields **default values**, and add methods and properties exactly as before:

song_dataclass.py

```

1 from dataclasses import asdict, dataclass
2
3
4 @dataclass
5 class Song:
6     title: str
7     artist: str
8     seconds: int = 180
9
10    @property
11    def minutes(self):
12        return round(self.seconds / 60, 1)
13
14
15 hit = Song("Sunrise", "The Larks", 210)
16 demo = Song("Untitled", "The Larks")
17 print(hit)
18 print(demo.seconds, hit.minutes)
19 print(hit == Song("Sunrise", "The Larks", 210))
20 print(asdict(demo))

```

```

Song(title='Sunrise', artist='The Larks', seconds=210)
180 3.5
True
{'title': 'Untitled', 'artist': 'The Larks', 'seconds': 180}

```

- Line 8 gives seconds a default, so Song("Untitled", "The Larks") on line 16 gets 180, like a default parameter value (Day 10).
- Lines 10 to 12 are a property from 17.7: $210 / 60 = 3.5$ minutes.
- Line 20: the dataclasses module also has a function `asdict()`, which turns a data class object into a dictionary of its fields. That is exactly what `json.dump` needs (Day 14).

Fields with defaults must come **after** fields without them, for the same reason as in a `def` line: the generated `__init__` lists the fields in order, and a parameter without a default cannot follow one with a default.

dataclass_order.py

```

1 from dataclasses import dataclass
2
3
4 @dataclass
5 class Song:
6     title: str
7     seconds: int = 180
8     artist: str

```

```
Traceback (most recent call last):
  File ".../day17/dataclass_order.py", line 4, in <module>
    @dataclass
    ^^^^^^^^^
...
TypeError: non-default argument 'artist' follows default argument 'seconds'
```

The ... stands for several lines from inside the `dataclasses` module; the first and last lines are the ones that matter. Notice that the error comes when the **class is defined**, before any object is made. Move `artist` above `seconds`.

Finally, `@dataclass(order=True)` also writes `__lt__`, `__le__`, `__gt__`, and `__ge__`. They compare two objects like tuples of their fields (Day 10): first fields first, and the next field only to break a tie.

`grade_order.py`

```
1 from dataclasses import dataclass
2
3
4 @dataclass(order=True)
5 class Grade:
6     score: int
7     name: str
8
9
10 grades = [Grade(88, "Ben"), Grade(95, "Ana"), Grade(88, "Ali")]
11 print(sorted(grades))
12 print(max(grades).name)
```

```
[Grade(score=88, name='Ali'), Grade(score=88, name='Ben'), Grade(score=95, name='Ana')]
Ana
```

The two 88s tie on `score`, so `name` decides: `'Ali'` comes before `'Ben'`. Put the fields in the order you want them compared.

KEY IDEA

`@dataclass` above a class with fields written as `name: type` (and optional defaults) generates `__init__`, `__repr__`, and `__eq__`; `@dataclass(order=True)` adds `<` and its relatives. Annotations document types but are not checked. Add methods and properties as usual, and use `asdict()` to get a dictionary of the fields.

WATCH OUT

A list as a default field. Day 10's mutable default trap would strike here too: one `[]` made once and shared by every recipe. Python catches it for you and refuses to define the class:

list_default.py

```

1 from dataclasses import dataclass
2
3
4 @dataclass
5 class Recipe:
6     name: str
7     ingredients: list = []
8
9
10 soup = Recipe("Tomato soup")

```

Traceback (most recent call last):

```

File ".../day17/list_default.py", line 4, in <module>
    @dataclass
    ^^^^^^^^^^^

```

...

ValueError: mutable default <class 'list'> for field ingredients is not allowed: use default_factory

The message names the fix. `field(default_factory=list)`, with `field` imported from `dataclasses`, says "for each new object, call `list()` to make a fresh empty list". Pass the function `list` itself, without parentheses (Day 10: functions are values):

list_default_fixed.py

```

1 from dataclasses import dataclass, field
2
3
4 @dataclass
5 class Recipe:
6     name: str
7     ingredients: list = field(default_factory=list)
8
9
10 soup = Recipe("Tomato soup")
11 salad = Recipe("Green salad")
12 soup.ingredients.append("tomatoes")
13 print(soup)
14 print(salad)

```

```

Recipe(name='Tomato soup', ingredients=['tomatoes'])
Recipe(name='Green salad', ingredients=[])

```

Each recipe has its own list, so adding tomatoes to the soup leaves the salad's list empty.

PAUSE AND TRY

Question 15 Tracing

What does this program print?

order_trace.py

```
1 from dataclasses import dataclass
2
3
4 @dataclass
5 class Order:
6     item: str
7     quantity: int = 1
8     price: float = 2.5
9
10    def total(self):
11        return self.quantity * self.price
12
13
14 first = Order("bagel", 3)
15 second = Order("bagel", price=3.0)
16 print(first)
17 print(first.total(), second.total())
18 print(first == Order("bagel", 3, 2.5), first == second)
```

Question 16 Debugging

This program should print `Pet(name='Rex', age=3)`. It crashes. Explain the error and fix it.

forgot_dataclass.py

```
1 from dataclasses import dataclass
2
3
4 class Pet:
5     name: str
6     age: int = 1
7
8
9 rex = Pet("Rex", 3)
10 print(rex)
```

17.9 When a Class Is Worth It

You now have a lot of tools: plain functions, dictionaries, classes, inheritance, data classes. Having them does not mean using all of them every time. More structure is not better in itself; the best code is the simplest code that keeps the program correct and easy to change. So, before writing a class, ask what it would buy you.

Some jobs need nothing more than a function and a dictionary. Counting words takes some text and gives back counts. No data has to be kept between calls, and there are no rules to protect:

word_count.py

```

1 def count_words(text):
2     counts = {}
3     for word in text.lower().split():
4         counts[word] = counts.get(word, 0) + 1
5     return counts
6
7
8 print(count_words("The cat saw the dog"))

```

```
{'the': 2, 'cat': 1, 'saw': 1, 'dog': 1}
```

A WordCounter class here would add a name, an `__init__`, and a method call, and do nothing a function cannot. On the other hand, the bank account earned its class: it keeps data between operations, it has rules that must always hold (the balance cannot go negative), several operations work on the same data, and the program makes many accounts of the same shape. Day 16 started from exactly this kind of signal: several functions that all took the same dictionary, with nothing stopping a misspelled key.

Situation	Usually best
One calculation: values in, a result out, nothing kept	A function
A lookup table or a tally that you build and read	A dictionary
Many records with the same fields and little behavior	A data class
Data plus rules that must always hold, and several operations on it	A class
Several kinds of thing that share most of their behavior	A parent class and child classes

The table is a guide, not a law. A program often starts with a function and a dictionary and grows into a class when a second and third function appear that all work on the same data. That is a good moment to switch, not a failure to plan.

KEY IDEA

Write a class when data and its rules belong together, several operations share that data, or you need many objects of the same shape. A single calculation is a function, a simple lookup is a dictionary, and a record with little behavior is a data class.

WATCH OUT

A function wearing a class costume. This class stores two values only so that one method can use them once:

tip_class.py

```
1 class TipCalculator:
2     def __init__(self, bill, percent):
3         self.bill = bill
4         self.percent = percent
5
6     def calculate(self):
7         return round(self.bill * self.percent / 100, 2)
8
9
10 print(TipCalculator(40, 15).calculate())
```

6.0

The tell-tale signs: the class has only `__init__` and one other method, and every use makes an object, calls the method once, and throws the object away. Nothing is kept, and there are no rules to protect. A function says the same thing in a third of the lines, and the call reads better:

tip_function.py

```
1 def tip(bill, percent):
2     return round(bill * percent / 100, 2)
3
4
5 print(tip(40, 15))
```

6.0

PAUSE AND TRY

Question 17 Short answer

For each program, say whether you would use a function, a dictionary, a data class, or a class, and give one reason: (a) a program that converts a distance in miles to kilometers; (b) a phone book that maps each friend's name to a phone number; (c) a game in which each character has health that must stay between 0 and 100, and can be healed, hit, and printed.

Putting it together

A school library lends books and films. Films go out for less time and cost double in late fines. Each loan remembers what was borrowed and for how many days, and each member's loan card can work out the total fine. The program below uses every idea from today.

library_loans.py

```
1 from dataclasses import dataclass
2
3
```

```
4 class Item:
5     kind = "item"
6     loan_days = 21
7     daily_fine = 0.25
8
9     def __init__(self, title, code):
10         self.title = title
11         self.code = code
12
13     def fine(self, days_late):
14         return days_late * self.daily_fine
15
16     def __eq__(self, other):
17         if not isinstance(other, Item):
18             return NotImplemented
19         return self.code == other.code
20
21     def __lt__(self, other):
22         return self.title < other.title
23
24     def __str__(self):
25         return f"{self.title} ({self.kind})"
26
27
28 class Book(Item):
29     kind = "book"
30
31     def __init__(self, title, code, pages):
32         super().__init__(title, code)
33         self.pages = pages
34
35
36 class Film(Item):
37     kind = "film"
38     loan_days = 7
39
40     def fine(self, days_late):
41         return super().fine(days_late) * 2
42
43
44 @dataclass
45 class Loan:
46     item: Item
47     days_out: int
48
49     @property
50     def days_late(self):
51         return max(0, self.days_out - self.item.loan_days)
52
53     @property
54     def fine(self):
55         return self.item.fine(self.days_late)
56
57
58 class LoanCard:
```

```

59     def __init__(self, owner):
60         self.owner = owner
61         self.loans = []
62
63     def borrow(self, item, days_out):
64         if item in [loan.item for loan in self.loans]:
65             raise ValueError(f"{self.owner} already has {item}")
66         self.loans.append(Loan(item, days_out))
67
68     def __len__(self):
69         return len(self.loans)
70
71     @property
72     def total_fine(self):
73         return sum(loan.fine for loan in self.loans)
74
75
76 card = LoanCard("Ana")
77 card.borrow(Book("Wonder", "B-17", 310), 25)
78 card.borrow(Film("Up", "F-02"), 10)
79 card.borrow(Book("Holes", "B-04", 233), 3)
80 try:
81     card.borrow(Book("Wonder", "B-17", 310), 1)
82 except ValueError as error:
83     print(f"Refused: {error}")
84
85 for loan in sorted(card.loans, key=lambda loan: loan.item):
86     print(f"{loan.item}: {loan.days_late} days late, fine {loan.fine:.2f}")
87 print(f"{card.owner} has {len(card)} loans; total fine {card.total_fine:.2f}")

```

```

Refused: Ana already has Wonder (book)
Holes (book): 0 days late, fine 0.00
Up (film): 3 days late, fine 1.50
Wonder (book): 4 days late, fine 1.00
Ana has 3 loans; total fine 2.50

```

Lines	What happens	Idea
4–25	Item is the parent: class attributes for the rules, a <code>fine</code> method, equality by catalog code, order by title	17.1, 17.5, 17.6
28–33	Book adds pages, calling <code>super().__init__</code> so the parent still stores the title and code	17.2
36–41	Film overrides the class attribute <code>loan_days</code> and the <code>fine</code> method, using <code>super().fine</code> and doubling it	17.3
44–55	Loan is a data class that has an item; its properties work out the days late and the fine every time they are read	17.4, 17.7, 17.8
58–73	LoanCard has a list of loans (composition), refuses an item it already has, and defines <code>__len__</code>	17.4, 17.6
81	A new Book object with code "B-17" is == to the Wonder already on the card, so <code>in</code> finds it and <code>borrow</code> refuses	17.5
85	Sorting by <code>loan.item</code> uses <code>Item.__lt__</code> , so the loans come out in title order	17.6

86	<code>loan.fine</code> is polymorphic: a book's fine and a film's fine run different methods	17.3
87	<code>len(card)</code> calls <code>__len__</code> ; <code>total_fine</code> is a property	17.6, 17.7

Work through the fines. Wonder was out 25 days and a book may go out for 21, so it is 4 days late: $4 \times 0.25 = 1.00$. Up was out 10 days and a film may go out for 7, so 3 days late; `Film.fine` doubles the parent's $3 \times 0.25 = 0.75$ to 1.50. Holes was out 3 days, and `max(0, 3 - 21)` is 0, so no fine. The total is 2.50.

Look at what each tool bought. Adding a third kind of item, say a magazine with a 14-day loan, is a three-line child class; the loan card, the loans, and the fine total all work with it unchanged, through polymorphism. And `Loan`, which only holds two values and works things out from them, took five lines as a data class.

Common mistakes

Many of today's mistakes show up as error messages that point at the line where the problem **appears**, not where it was made. Use this table to go from a message to its usual cause.

Message	Usual cause	Fix
AttributeError: 'SavingsAccount' object has no attribute 'balance'	The child's <code>__init__</code> never calls <code>super().__init__</code>	Call <code>super().__init__(...)</code> first
AttributeError: 'BankAccount' object has no attribute 'add_interest'	Called a child-only method on a parent object	Call it only on child objects
TypeError: Pet.__init__() takes 2 positional arguments but 3 were given	Passed self to <code>super().__init__(self, name)</code>	<code>super().__init__(name)</code>
NotImplementedError from the parent's method	The child forgot the override, or misspelled its name	Define the method with the parent's exact name
TypeError: '<' not supported between instances of 'Runner' and 'Runner'	Sorted, or used <code><</code> , <code>min</code> , or <code>max</code> , without <code>__lt__</code>	Define <code>__lt__</code> , or pass <code>key=</code>
TypeError: cannot use 'Point' as a set element (unhashable type: 'Point')	The class defines <code>__eq__</code> , so its objects are unhashable	Store a hashable stand-in, such as a tuple
<code>==</code> gives None	<code>__eq__</code> works out the answer but does not return it	return the comparison
TypeError: 'int' object is not callable	Called a property with parentheses	Drop the <code>()</code>
AttributeError: property 'area' of 'Rectangle' object has no setter	Assigned to a property	Change the attributes it is computed from
TypeError: 'float' object cannot be interpreted as an integer	<code>__len__</code> returned something other than a whole number	Return a count; use a property for measurements

TypeError: non-default argument 'artist' follows default argument 'seconds'	A data class field without a default after one with a default	Put fields with defaults last
ValueError: mutable default <class 'list'> for field ...	A list default in a data class	<code>field(default_factory=list)</code>
TypeError: Pet() takes no arguments	The @dataclass line is missing	Add @dataclass above the class

Older versions of Python may word some of these messages a little differently.

Vocabulary

Term	Definition	Example
inheritance / inherit	Defining a class as an extended version of another, getting all its methods	<code>class SavingsAccount(BankAccount):</code>
parent class / base class / superclass	The class being inherited from	<code>BankAccount</code>
child class / subclass	The class that inherits	<code>SavingsAccount</code>
<code>isinstance()</code>	True if an object is an instance of a class or of any child of it	<code>isinstance(savings, BankAccount)</code>
"is-a" relationship	The link between a child and its parent: a child object is a kind of parent object	A savings account is a bank account
object	The built-in class every class inherits from in the end	<code>isinstance(plain, object)</code>
<code>super()</code>	The parent class, working on the same object	<code>super().__init__(owner, balance)</code>
overriding / override	Defining a method in a child with the parent's name, to change its behavior	<code>FeeAccount.withdraw</code>
polymorphism	One method call working on objects of different classes, each in its own way	<code>account.withdraw(10)</code> in a loop
<code>NotImplementedError</code>	An exception a parent's method raises when a child must provide that method	<code>raise NotImplementedError("...")</code>
duck typing	Using any object that has the needed method, whatever its class	<code>Banner().send(text)</code>
composition / "has-a" relationship	An object keeping other objects as attributes and using them	<code>self.accounts = []</code>
delegating	Handing part of a job to an object you hold	<code>len(self.songs)</code>

<code>__eq__</code>	The method <code>==</code> calls	<code>def __eq__(self, other):</code>
<code>NotImplemented</code>	A value <code>__eq__</code> returns to say "I cannot compare with that"	<code>return NotImplemented</code>
<code>__lt__</code>	The method <code><</code> (and sorting) calls	<code>return self.time < other.time</code>
<code>__len__</code>	The method <code>len()</code> calls; it must return a whole number, and 0 makes the object falsy	<code>return len(self.songs)</code>
decorator	A line starting with <code>@</code> above a <code>def</code> or <code>class</code> that changes what it makes	<code>@property</code>
property	A computed attribute: reading it runs a method, with no parentheses	<code>room.area</code>
<code>_name</code> (one leading underscore)	A convention meaning "internal; use only inside the class"	<code>self._balance</code>
data class / <code>@dataclass</code>	A class whose <code>__init__</code> , <code>__repr__</code> , and <code>__eq__</code> are written from its fields	<code>@dataclass</code> above <code>class Point:</code>
field	One attribute declared in a data class body	<code>x: int</code>
type annotation	A note after a colon saying what type a value should be; not checked when the program runs	<code>title: str</code>
<code>asdict()</code>	Turns a data class object into a dictionary of its fields	<code>asdict(song)</code>
<code>field(default_factory=list)</code>	A data class default that makes a fresh value for each object	<code>tags: list = field(default_factory=list)</code>

Summary

- **Inheritance** (`class Child(Parent):`) gives a child every method and class attribute of its parent, so you write only what is new. A child object **is a** parent object: `isinstance` says so, and `except Parent` catches it. Every class inherits from `object` in the end.
- A child method with a parent's name **overrides** it. `super()` reaches the parent's version, so a child's `__init__` calls `super().__init__(...)` and an override can build on the parent.
Polymorphism means one call runs each object's own version.
- **Composition** ("has-a") keeps other objects as attributes and delegates to them. Prefer it unless the new class truly is a kind of the old one and every inherited method makes sense for it.
- Special methods plug your objects into Python: `__eq__` for `==` (return `NotImplemented` for other types), `__lt__` for `<` and sorting, and `__len__` for `len()` and truthiness.
- `@property` makes a computed, read-only attribute that is never stale. `@dataclass` writes `__init__`, `__repr__`, and `__eq__` from the fields of a class that mostly holds data.

- A class is worth writing when data, its rules, and several operations belong together; otherwise a function, a dictionary, or a data class is often enough.

Practice

Do these in your `day17` folder. Type each program yourself, run it, and only then open the answer.

Question 18 Short answer

Shapes. Write a `Shape` parent class whose `area()` method raises `NotImplementedError`, and whose `__str__` describes the shape using `self.area()`. Then write `Circle` (made from a radius) and `Rectangle` (made from a width and a height) child classes that each compute `area()`. Put several shapes in a list, print each one, and print the total area.

Question 19 Short answer

Task as a data class. Rewrite the to-do manager's `Task` class from Day 16 as a data class, keeping its `complete()` method and the `__str__` from Day 16's practice. Use `asdict()` to save it. Then simplify `tasks.py` wherever the data class makes that possible.

Question 20 Short answer

Sortable objects. Write a `Student` class with a name and a test score. Add `__eq__` (same name and same score) and `__lt__` (lower score first; for equal scores, alphabetical by name), so that a list of students can be sorted with plain `sorted()`. Test sorting, `min`, `max`, and `in`.

Review questions

Question 21 Tracing

What does this program print? Watch both the class attribute and the method in each class.

review_vehicles.py

```

1 class Vehicle:
2     wheels = 4
3
4     def __init__(self, name):
5         self.name = name
6
7     def describe(self):
8         return f"{self.name}: {self.wheels} wheels, {self.power()}"
9
10    def power(self):
11        return "gas"
12
13
14    class Bike(Vehicle):
15        wheels = 2
16
17        def power(self):
18            return "pedals"
19
20
21    class EBike(Bike):
22        def power(self):
23            return super().power() + " and a battery"
24
25
26    for ride in [Vehicle("Van"), Bike("Road bike"), EBike("Cargo bike")]:
27        print(ride.describe())
28    print(EBike.wheels, isinstance(EBike("Mini"), Vehicle))

```

Question 22 Debugging

This program should print `Tom True`. It crashes. Explain the error and fix it.

super_self.py

```

1 class Pet:
2     def __init__(self, name):
3         self.name = name
4
5
6 class Cat(Pet):
7     def __init__(self, name, indoor):
8         super().__init__(self, name)
9         self.indoor = indoor
10
11
12 tom = Cat("Tom", True)
13 print(tom.name, tom.indoor)

```

Question 23 Multiple choice

Which methods does a plain `@dataclass` (with no `order=True`) write for you?

- A. `__init__`, `__repr__`, and `__eq__`
- B. `__init__`, `__repr__`, `__eq__`, and `__lt__`
- C. `__init__` and `__str__` only

D. `__init__`, `__len__`, and `__eq__`

Question 24 Tracing

What does this program print? `Item` is a data class with `order=True`, so it compares by `price` first and then by `name`.

review_cart.py

```

1  from dataclasses import dataclass
2
3
4  @dataclass(order=True)
5  class Item:
6      price: float
7      name: str
8
9
10 class Cart:
11     def __init__(self):
12         self.items = []
13
14     def add(self, item):
15         if item not in self.items:
16             self.items.append(item)
17
18     def __len__(self):
19         return len(self.items)
20
21     @property
22     def total(self):
23         return sum(item.price for item in self.items)
24
25
26 cart = Cart()
27 shopping = [
28     Item(2.5, "milk"),
29     Item(1.25, "bread"),
30     Item(2.5, "milk"),
31     Item(1.25, "apple"),
32 ]
33 for item in shopping:
34     cart.add(item)
35 print(len(cart), cart.total)
36 print([item.name for item in sorted(cart.items)])
37 print(bool(Cart()), max(cart.items).name)

```

Question 25 Short answer

In `fee_account.py`, `FeeAccount.withdraw` is a single line: `super().withdraw(amount + self.fee)`. A classmate instead copies the body of `BankAccount.withdraw` into `FeeAccount` and adds the fee to it. Give two reasons the `super()` version is better.

Question 26 Short answer

A program keeps each pet at a vet as a dictionary with `"name"`, `"weight"`, and `"vaccines"` keys. It has three functions, `add_vaccine(pet, name)`, `weight_change(pet, new_weight)`, and `describe(pet)`,

and every one takes a pet dictionary first. A weight must never be 0 or less. Would you turn this into a function-and-dictionary design, a data class, or a class? Explain your choice, and say what each part of today's toolkit would do in your design.

Go further

The official Python Tutorial covers inheritance, including classes with more than one parent, in [Classes: Inheritance](#), and the `dataclasses` module documentation lists every option `@dataclass` accepts. Tomorrow, Day 18, explains how decorators such as `@property` work by writing your own.

Reflection

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

☐ I can write a child class that inherits from a parent and say where Python finds each method.

Reread: [17.1 Inheritance: Building on an Existing Class](#)

☐ I can extend `__init__` in a child class by calling `super().__init__` first. Reread: [17.2 Extending `\_\_init\_\_` with `super\(\)`](#)

☐ I can override a method, reuse the parent version with `super()`, and explain polymorphism.

Reread: [17.3 Overriding Methods and Polymorphism](#)

☐ I can choose between inheritance (is-a) and composition (has-a) for a design. Reread: [17.4 Composition Versus Inheritance](#)

☐ I can write `__eq__` that returns `NotImplemented` for other types. Reread: [17.5 Comparing with `\_\_eq\_\_`](#)

☐ I can write `__lt__` and `__len__` so sorting and `len()` work on my objects. Reread: [17.6 Ordering with `\_\_lt\_\_` and Size with `\_\_len\_\_`](#)

☐ I can use `@property` for a computed, read-only attribute. Reread: [17.7 Computed Attributes with `@property`](#)

☐ I can write a data class with fields, defaults, and `default_factory`. Reread: [17.8 Data Classes with `@dataclass`](#)

☐ I can decide when a class is worth it and when a function and a dictionary are enough. Reread: [17.9 When a Class Is Worth It](#)

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
Seat A1: 30
Seat A1 may use the lounge
True False
```

VipTicket has no `__init__` or `describe` of its own, so `VipTicket("A1", 30)` uses Ticket's `__init__`, and `vip.describe()` finds `describe` in Ticket. `lounge` is found in VipTicket, and it reads `self.seat`, which the parent's `__init__` stored. `vip` is a VipTicket, which is a kind of Ticket, so `isinstance(vip, Ticket)` is True. `regular` is a plain Ticket, not a VipTicket, so the last check is False.

Reread 17.1 *Inheritance: Building on an Existing Class* if your output differed.

Question 2

C. `approve` exists only in Manager. Looking it up on `clerk` searches the object, then `Employee`, then object, and never reaches Manager, because a parent does not inherit from its children.

- **A** is wrong: `boss` is a Manager, which inherits `pay` from Employee.
- **B** is wrong: `approve` is defined in Manager itself.
- **D** is wrong: `isinstance` never raises for a class; it returns True here, because a manager is an employee.

Reread 17.1 *Inheritance: Building on an Existing Class* if you chose differently.

Question 3

```
Ana 25 2
250
```

`Manager("Ana", 20, ["Ben", "Cy"])` runs `Manager.__init__`. Line 12 calls the parent's `__init__` with `name = "Ana"` and `hourly + 5 = 25`, so the parent stores `hourly = 25`, not 20. Line 13 stores the team list, which has 2 names. `lead.pay(10)` is inherited from Employee and works out `self.hourly * hours = 25 * 10 = 250`.

Reread 17.2 *Extending __init__ with super()* if your output differed.

Question 4

The traceback ends with:

```
AttributeError: 'Dog' object has no attribute 'name'
```

`Dog` defines its own `__init__`, which replaces `Pet.__init__` for dogs. It stores `breed` but never runs the parent's `__init__`, so `name` is never stored. Call the parent first:

dog_breed_fixed.py

```

1 class Pet:
2     def __init__(self, name):
3         self.name = name.title()
4
5
6 class Dog(Pet):
7     def __init__(self, name, breed):
8         super().__init__(name)
9         self.breed = breed
10
11
12 rex = Dog("rex", "beagle")
13 print(rex.breed)
14 print(f"{rex.name} the {rex.breed}")

```

```

beagle
Rex the beagle

```

The parent's `__init__` also tidies the name with `title()`, which is why "rex" prints as Rex. That is one more reason to call the parent rather than copy its line: any rule the parent applies comes along for free.

Reread 17.2 *Extending `__init__` with `super()`* if you missed it.

Question 5

```

10.00
8.00
7.00

```

All three objects use `Price.label`, which calls `self.final()`. Which `final` runs depends on `self`'s class, not on where `label` is written:

Object	final that runs	Result
Price(10)	Price.final: 10	10.00
Sale(10)	Sale.final: 10 × 0.8	8.00
Clearance(10)	Clearance.final: super().final() is Sale.final, 8.0, then – 1	7.00

Clearance's parent is Sale, so `super()` there reaches Sale, not Price. This is polymorphism inside a class: a parent method that calls `self.something()` gets each child's override.

Reread 17.3 *Overriding Methods and Polymorphism* if your output differed.

Question 6

B. `PushNotifier` inherits everything from `Notifier`, including `send`. The lookup does not find `send` in `PushNotifier`, moves on to `Notifier`, and runs that version, which raises.

- **A** is wrong: the method is found in the parent, and it raises rather than returning.
- **C** is wrong: an `AttributeError` would need the whole search to fail, but `Notifier` has `send`.
- **D** is wrong: nothing in either class prints.

Reread 17.3 *Overriding Methods and Polymorphism* if you chose differently.

Question 7

C. A square **is** a shape, and anything every shape can do (work out its area, describe itself) makes sense for a square.

- **A** is wrong: a car **has** an engine; a car is not a kind of engine. Give the car an `engine` attribute.
- **B** is wrong: a classroom **has** students. That is composition, a list of `Student` objects.
- **D** is wrong: an order **has** a list of items. Inheriting from `list` would give it `insert`, `clear`, and other methods that could break the order's rules, as `playlist_inherit.py` showed.

Reread 17.4 *Composition Versus Inheritance* if you chose differently.

Question 8

Model answer: because `Team` inherits from `list`, every team also has `append`, `insert`, `extend`, and the other list methods, and none of them check the limit. Any code can call `team.append(player)` and make a twelfth player without `add_player` ever running. A design that keeps the rule uses composition: `Team` inherits from nothing, its `__init__` makes `self.players = []`, and `add_player` checks `len(self.players)` before appending. Then `add_player` is the only way to add a player, so the limit always holds.

A correct answer says that inheriting from `list` brings unchecked ways to add players, and that holding a list as an attribute (has-a) lets the class control every change.

Reread 17.4 *Composition Versus Inheritance* if your reasoning differed.

Question 9

```
True
2 1
True False False
```

- Line 13: `in` compares `Coin(25, 1999)` with each coin using `__eq__`; the first coin has value 25, so the answer is `True`.
- Line 14: `count` finds two coins worth 25 (2019 and 2024). `index` finds the first coin worth 10 at index 1.
- Line 15: `purse[0]` and `purse[2]` are both worth 25, so `==` is `True`, but they are two different objects, so `is` is `False`. `purse[0] == 25` compares a coin with an int: `__eq__` returns `NotImplemented`, the int does not know how to compare with a coin either, and Python falls back to identity, which gives `False`.

Reread 17.5 *Comparing with __eq__* if your output differed.

Question 10

There is no error message: the program runs and prints the wrong thing. Line 7 works out `True`, but nothing returns it. A function or method without `return` gives back `None` (Day 9), and `==` passes on whatever `__eq__` returns. Add `return`:

card_eq_fixed.py

```

1 class Card:
2     def __init__(self, rank, suit):
3         self.rank = rank
4         self.suit = suit
5
6     def __eq__(self, other):
7         return self.rank == other.rank and self.suit == other.suit
8
9
10 print(Card("Q", "hearts") == Card("Q", "hearts"))

```

True

This is the same mistake as printing instead of returning in `__str__` (Day 16): Python asks the method for a value, so the method must `return` it.

Reread 17.5 *Comparing with `__eq__`* if you missed it.

Question 11

```

[lamp, kettle, books]
books 3 5
[lamp, books, kettle]

```

- Line 16: `sorted` uses `__lt__`, so the boxes go from lightest to heaviest: lamp (4), kettle (9), books (12). `__repr__` returns just the label, so the list shows no quotes.
- Line 17: `max` uses `__lt__` too and finds books (12). `len(boxes)` is the length of the **list**, 3 boxes. `len(boxes[0])` calls `__len__` on the books box: "books" has 5 letters.
- Line 18: `key=len` sorts by each box's `len()`, which calls `__len__`: lamp 4, books 5, kettle 6.

Reread 17.6 *Ordering with `__lt__` and Size with `__len__`* if your output differed.

Question 12

B. When a class defines `__len__`, Python uses it to decide truthiness: a length of 0 is falsy, like an empty list (Day 4). The conditional expression (Day 4) picks "empty".

- **A** is wrong: that is true only for objects **without** `__len__`.
- **C** is wrong: truthiness is not a check for `None`; an empty list is not `None`, yet it is falsy.
- **D** is wrong: any object can be used in a condition; Python works out its truthiness.

Reread 17.6 *Ordering with `__lt__` and Size with `__len__`* if you chose differently.

Question 13

```

77.0 False
14.0 True

```

Only `celsius` is stored. Each read of `fahrenheit` or `is_freezing` runs its method with the current `celsius`:

Line	celsius	fahrenheit	is_freezing
14	25	$25 \times 9 / 5 + 32 = 77.0$	$25 \leq 0$: False

16	-10	$-10 \times 9 / 5 + 32 = 14.0$	$-10 \leq 0$: True
----	-----	--------------------------------	---------------------

The `/` makes a float (Day 2), so 77.0 prints with `.0`.

Reread 17.7 *Computed Attributes with @property* if your output differed.

Question 14

B. `full_name` describes the student, needs no arguments, is quick to work out, and changes nothing. Making it a property means it is always built from the current names and reads naturally: `student.full_name`.

- **A** is wrong: enrolling is an action that changes the object and needs an argument, so it is a method.
- **C** is wrong: saving needs an argument and does slow work with a file, which a reader would not expect from reading an attribute.
- **D** is wrong: adding a grade changes the object and needs an argument.

Reread 17.7 *Computed Attributes with @property* if you chose differently.

Question 15

```
Order(item='bagel', quantity=3, price=2.5)
7.5 3.0
True False
```

- Line 14: `Order("bagel", 3)` fills `item` and `quantity` by position; `price` keeps its default 2.5.
- Line 15: the keyword argument `price=3.0` skips `quantity`, which keeps its default 1.
- Line 16: `print` uses the generated `__repr__`, with every field.
- Line 17: $3 \times 2.5 = 7.5$ and $1 \times 3.0 = 3.0$.
- Line 18: the generated `__eq__` compares all three fields. `Order("bagel", 3, 2.5)` matches `first` exactly. `second` has a different quantity and price, so it is not equal.

Reread 17.8 *Data Classes with @dataclass* if your output differed.

Question 16

The traceback's last line is:

```
TypeError: Pet() takes no arguments
```

The `@dataclass` line is missing above `class Pet:`. Importing `dataclass` does nothing by itself; only the decorator line writes the methods. Without it, `Pet` is an ordinary class with no `__init__`, which is why it takes no arguments (Day 16 met this message for a misspelled `__init__`). The annotated lines just record notes, and `age = 1` becomes a class attribute. Add the decorator:

forgot_dataclass_fixed.py

```

1  from dataclasses import dataclass
2
3
4  @dataclass
5  class Pet:
6      name: str
7      age: int = 1
8
9
10 rex = Pet("Rex", 3)
11 print(rex)

```

```
Pet(name='Rex', age=3)
```

Reread 17.8 *Data Classes with @dataclass* if you missed it.

Question 17

Model answer:

- **(a) A function.** Miles in, kilometers out, and nothing is kept between calls, so a class would add nothing.
- **(b) A dictionary.** It is a lookup table: names are keys and numbers are values, and there are no rules beyond that.
- **(c) A class.** Each character keeps data (health) between operations, there is a rule that must always hold (0 to 100), and several operations (heal, hit, print) work on that data. Methods can enforce the rule, and a property could let outside code read the health without setting it.

A correct answer matches each choice to whether data is kept, whether rules must be protected, and how many operations share the data.

Reread 17.9 *When a Class Is Worth It* if your reasoning differed.

Question 18

One possible solution, which also adds a `Square` as a child of `Rectangle`:

shapes.py

```

1  import math
2
3
4  class Shape:
5      name = "shape"
6
7      def area(self):
8          raise NotImplementedError("each kind of shape must define area()")
9
10     def __str__(self):
11         return f"{self.name} with area {self.area():.2f}"
12
13
14     class Circle(Shape):
15         name = "circle"
16

```

```

17     def __init__(self, radius):
18         self.radius = radius
19
20     def area(self):
21         return math.pi * self.radius**2
22
23
24 class Rectangle(Shape):
25     name = "rectangle"
26
27     def __init__(self, width, height):
28         self.width = width
29         self.height = height
30
31     def area(self):
32         return self.width * self.height
33
34
35 class Square(Rectangle):
36     name = "square"
37
38     def __init__(self, side):
39         super().__init__(side, side)
40
41
42 shapes = [Circle(1), Rectangle(4, 2.5), Square(3)]
43 for shape in shapes:
44     print(shape)
45 print(f"Total area: {sum(shape.area() for shape in shapes):.2f}")

```

```

circle with area 3.14
rectangle with area 10.00
square with area 9.00
Total area: 22.14

```

How it works: `Shape.__str__` calls `self.area()`, and polymorphism picks each child's `area` (17.3). Each child sets a class attribute `name` that overrides the parent's, so the description says what kind of shape it is. `Circle` uses `math.pi` (Day 13) and `**` (Day 2): $\pi \times 1^2$ is about 3.14. `Square` has only an `__init__`, which hands the side to `Rectangle.__init__` twice through `super()` (17.2); it inherits `area` from `Rectangle`. The total adds `3.14159... + 10.0 + 9.0` with a generator expression (Day 11). Forgetting `area` in a new child raises the parent's `NotImplementedError` as soon as the shape is printed.

Reread 17.2 *Extending `__init__` with `super()`* and 17.3 *Overriding Methods and Polymorphism* if your result differed.

Question 19

In `todo_app/task.py`, the hand-written `__init__` becomes two fields, and `to_dict` uses `asdict`:

`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):
14        """Mark this task as done."""
15        self.done = True
16
17    def to_dict(self):
18        """Return this task as a dictionary, ready to save as JSON."""
19        return asdict(self)
20
21    def __str__(self):
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))

```

Run directly, its test block prints:

```

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

```

The first line comes from the generated `__repr__`. `print(task)` still uses your own `__str__`, because `@dataclass` writes `__repr__` but leaves a `__str__` you wrote alone. The last line uses the generated `__eq__`, which Day 16's class did not have.

In `tasks.py`, two things change. `list_tasks` prints each task with its `__str__`, and `load_tasks` builds each task with `Task(**item)`: each saved dictionary has exactly the keys `title` and `done`, which are the names of the generated `__init__`'s parameters, so `**` (Day 10) passes them as keyword arguments.

`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, using Task.__str__."""
21     if not tasks:
22         print("No tasks yet.")
23         return
24     for number, task in enumerate(tasks, 1):
25         print(f"{number}. {task}")
26
27
28 def complete_task(tasks, number):
29     """Mark task number (counting from 1) as done.
30
31     Return the Task, or None if no task has that number.
32     """
33     if not 1 <= number <= len(tasks):
34         return None
35     task = tasks[number - 1]
36     task.complete()
37     return task
38
39
40 def delete_task(tasks, number):
41     """Remove task number (counting from 1) from tasks.
42
43     Return the removed Task, or None if no task has that number.
44     """
45     if not 1 <= number <= len(tasks):
46         return None
47     return tasks.pop(number - 1)
48
49
50 def save_tasks(tasks, path):
51     """Write tasks to the JSON file at path, replacing what was there."""
52     data = [task.to_dict() for task in tasks]
53     with path.open("w", encoding="utf-8") as file:
54         json.dump(data, file, indent=2)
55
56
57 def load_tasks(path):
58     """Return the Tasks saved at path, or [] if there are none to load.
59
60     A missing file means this is the first run. A file that is not valid
```

```

61     JSON gets a warning, and the program starts with no tasks.
62     """
63     try:
64         with path.open(encoding="utf-8") as file:
65             data = json.load(file)
66     except FileNotFoundError:
67         return []
68     except json.JSONDecodeError:
69         print(f"Warning: {path.name} is not valid JSON, so no tasks loaded.")
70         return []
71     return [Task(**item) for item in data]
72
73
74 if __name__ == "__main__":
75     test_tasks = []
76     add_task(test_tasks, "Test the module")
77     print(complete_task(test_tasks, 1).done)
78     print(complete_task(test_tasks, 2))
79     list_tasks(test_tasks)

```

main.py does not change at all; it is the same file as on Day 16. With the same typed answers, the session is exactly Day 16's:

```

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: a task is a record with two fields and almost no behavior, which is exactly what a data class is for (17.9). The save file stays the same JSON because `asdict` produces the same

dictionary that `to_dict` built by hand. If a later version adds a field, such as `due`, you add one line to the class and `asdict` and the `__init__` pick it up.

Reread 17.8 *Data Classes with @dataclass* if your result differed.

Question 20

One possible solution:

students_sort.py

```
1 class Student:
2     def __init__(self, name, score):
3         self.name = name
4         self.score = score
5
6     def __eq__(self, other):
7         if not isinstance(other, Student):
8             return NotImplemented
9         return self.name == other.name and self.score == other.score
10
11    def __lt__(self, other):
12        if self.score != other.score:
13            return self.score < other.score
14        return self.name < other.name
15
16    def __repr__(self):
17        return f"Student({self.name!r}, {self.score})"
18
19
20 roster = [Student("Cy", 88), Student("Ana", 95), Student("Ben", 88)]
21 print(sorted(roster))
22 print(min(roster), max(roster))
23 print(Student("Ana", 95) in roster, Student("Ana", 90) in roster)
```

```
[Student('Ben', 88), Student('Cy', 88), Student('Ana', 95)]
Student('Ben', 88) Student('Ana', 95)
True False
```

How it works: `__lt__` compares scores first; only when they are equal does it compare names, so Ben and Cy (both 88) come out alphabetically. `min` and `max` use the same `__lt__`. `__eq__` starts with the `isinstance` check and returns `NotImplemented` for other types (17.5), so `in` can find `Student("Ana", 95)` but not `Student("Ana", 90)`. `__repr__` (Day 16) makes the printed lists readable. The same order could come from `@dataclass(order=True)` with the fields in the order `score, name` (17.8), at the cost of showing the score first in the repr.

Reread 17.5 *Comparing with __eq__* and 17.6 *Ordering with __lt__ and Size with __len__* if your result differed.

Question 21

```
Van: 4 wheels, gas
Road bike: 2 wheels, pedals
Cargo bike: 2 wheels, pedals and a battery
2 True
```

All three objects use `Vehicle.describe`, which reads `self.wheels` and calls `self.power()`. Each lookup starts at the object's own class:

Object	wheels found in	power that runs
Vehicle	Vehicle: 4	Vehicle.power: "gas"
Bike	Bike: 2	Bike.power: "pedals"
EBike	not in E Bike; Bike: 2	EBike.power: <code>super().power()</code> is Bike.power, plus " and a battery"

`EBike.wheels` is 2, found in its parent `Bike`. An `EBike` is a `Bike`, which is a `Vehicle`, so `isinstance` is `True` two levels up.

Reread [17.1 Inheritance: Building on an Existing Class](#) and [17.3 Overriding Methods and Polymorphism](#) if your output differed.

Question 22

The traceback's last line is:

```
TypeError: Pet.__init__() takes 2 positional arguments but 3 were given
```

Line 8 passes `self` to `super().__init__`. But `super()` already knows the object and passes it as `self` automatically, just as `tom.method()` does. So `Pet.__init__` receives the object, then `self` again, then `name`: three arguments for two parameters (`self` and `name`). As on Day 16, an error that counts one more argument than you meant points at a hidden `self`. Remove the extra `self`:

`super_self_fixed.py`

```
1 class Pet:
2     def __init__(self, name):
3         self.name = name
4
5
6 class Cat(Pet):
7     def __init__(self, name, indoor):
8         super().__init__(name)
9         self.indoor = indoor
10
11
12 tom = Cat("Tom", True)
13 print(tom.name, tom.indoor)
```

```
Tom True
```

Reread [17.2 Extending `\_\_init\_\_` with `super\(\)`](#) if you missed it.

Question 23

A. A data class gets an `__init__` with one parameter per field, a `__repr__` that shows every field, and an `__eq__` that compares every field.

- **B** is wrong: the ordering methods, including `__lt__`, come only with `@dataclass(order=True)`.
- **C** is wrong: it writes `__repr__`, not `__str__`. `print` shows the repr only because there is no `__str__` to use, and it also writes `__eq__`.
- **D** is wrong: a data class has no idea what its "length" would mean, so it writes no `__len__`.

Reread 17.8 *Data Classes with @dataclass* if you chose differently.

Question 24

```
3 5.0
['apple', 'bread', 'milk']
False milk
```

- Lines 33 and 34: `add` checks `item` not in `self.items`, and `in` uses the generated `__eq__`. The second `Item(2.5, "milk")` is a new object, but it equals the first, so it is not added. The cart holds milk, bread, and apple.
- Line 35: `len(cart)` calls `__len__`: 3. The `total` property adds $2.5 + 1.25 + 1.25 = 5.0$.
- Line 36: sorting compares like tuples. Bread and apple tie at 1.25, so the names decide: `'apple'` before `'bread'`. Milk at 2.5 is last.
- Line 37: a new, empty `Cart()` has length 0, so `bool` gives `False`. `max` finds the item that sorts last, milk.

Reread 17.6 *Ordering with __lt__ and Size with __len__* and 17.8 *Data Classes with @dataclass* if your output differed.

Question 25

Model answer:

1. **The rules live in one place.** The checks (the amount must be more than 0; the balance cannot go negative) stay only in `BankAccount.withdraw`. If a check changes or a bug is fixed there, `FeeAccount` gets the change automatically. With a copy, someone has to remember to change both, and the two kinds of account can quietly start following different rules.
2. **The override says only what is different.** One line states exactly how a fee account differs: it withdraws the amount plus the fee. A reader does not have to compare two long methods to find the difference.

A correct answer mentions that the parent's checks are reused rather than duplicated, so a later fix to the parent applies to the child, and that the override shows only the difference.

Reread 17.3 *Overriding Methods and Polymorphism* if your reasoning differed.

Question 26

Model answer: a **class**, perhaps written as a data class with methods. Three functions all take the same kind of dictionary first, which is the signal from Day 16 and 17.9 that the data and the functions belong together. There is also a rule to protect (weight above 0), which a dictionary cannot guarantee, since any code can set `pet["weight"] = -3` or misspell a key.

In the design: the fields `name`, `weight`, and `vaccines` become attributes, with `vaccines` made fresh for each pet (in `__init__`, or with `field(default_factory=list)` in a data class); `add_vaccine` and `weight_change` become methods, and `weight_change` raises `ValueError` for a weight of 0 or less; `describe` becomes `__str__`, so `print(pet)` works. A data class would also give a readable `__repr__` and `__eq__` for free. To stop outside code from assigning a bad weight directly, store it as `_weight` and expose it with a `@property`.

A correct answer names the signals (several functions sharing one dictionary, a rule to keep) and puts the rule inside a method so it is checked on every change.

Reread *17.9 When a Class Is Worth It* and *17.7 Computed Attributes with @property* if your reasoning differed.