CIS 1100

Object Oriented Programming

Python
Fall 2024
University of Pennsylvania

Learning Objectives

- Describe the purpose of a `class` in Python
- Create new data types by defining `class`es in Python and construct objects that belong to that data type.
- Define, understand the purpose of, and write member attributes and methods
- Write an `__init__` method for a class
- Understand and use the `self` as a method argument
- Expand understanding of variable scope in Python

CIS 1100

Objects

Python
Fall 2024
University of Pennsylvania

Objects: Beyond Primitives

Our primitive types (`int`, `float`, `bool`, etc.) are called primitive in part because:

- they express simple, individual values
- they have limited and fixed sets of operations that are generally defined by simple symbol operators

Objects: Beyond Primitives

Strings are not primitive in Python, and so we could *call methods* on them to perform certain operations:

```
name = "Harry"
idx_of_r = name.find("r")
yelling = name.upper()
```

Strings are examples of *objects*—more complex than primitives!

Objects: Beyond Primitives

Lists, for example, are also *non-primitive objects*

```
rhyme = ["What's", "The", "Story", "Morning", "Glory"]
rhyme.append("?")
rhyme.extend(["By", "Oasis"])
```

Objects: Definition

Objects are values in Python that:

- can store their own attributes
- can have functions (methods) called on them directly

New Example: `datetime.datetime`

Python has a built-in library called `datetime`. Inside of this library live a few new types of object, including:

- `datetime` (yes, same as the library name 😬)
- `timedelta`

Using Objects

Use `.` syntax to access attributes of objects and to call methods on objects

```
>>> from datetime import datetime, timedelta
>>> right_now = datetime.now()   # returns a datetime object with the current time
>>> right_now
2024-09-17 11:59:57.608895
>>> right_now.replace(year=2014) # call the replace method on the datetime object
2014-09-17 11:59:57.608895
>>> right_now.year               # access the year attribute of the object
2024
>>> offset = timedelta(days=60)  # represent a duration of 60 days
>>> offset
60 days, 0:00:00
>>> right_now + offset           # move forward the datetime by the timedelta with + operator
2024-11-16 11:59:57.608895
```

CIS 1100

Classes & State

Python
Fall 2024
University of Pennsylvania

Classes

A **class** in Python is a construct that allows us to *"bundle data and functionality together."**

- A `class` defines a new data type!
- Allows **instances** of that `class` to be created.

* *From the Python documentation on classes*

Classes & State

State is the notion of information stored by a program or an entity within a program that may change over time.

- Previously: represent state by defining some variables storing primitive values
- Next: Create instances of a class that store their own attributes that can be changed over time.

Classes & State

State arises from information stored in our program.

The "state of a square" in a drawing is represented by the values of the variables used to draw it over time:

```
import penndraw as pd
x_center = 0.5 # SETUP
while True:
    pd.clear()
    pd.filled_square(x_center, 0.5, 0.1)
    x_center += 0.01
    if x_center - 0.1 > 1.0:
      x_center = -0.1
    pd.advance()
```

Classes & State

State arises from information stored in our program—*as long as we remember what each variable is supposed to be representing!*

- Classes allow us to define objects that keep track of their own attributes
  - ...their own state!

CIS 1100

Abstraction

Python
Fall 2024
University of Pennsylvania

Class Design & Abstraction

Classes provide **abstractions** of real-world entities that
can be represented and manipulated by a program.

- If we write a program designed to "register students for courses", the entities of the student and the course are not literally contained within a computer.

Abstraction in Concrete Terms

An **abstraction** of an entity is *the set of information properties relevant to a stakeholder about an entity*

- **Information Property** (or just "property"): a named, objective and quantifiable aspect of an entity
- **Stakeholder**: a real or imagined person (or a class of people) who is seen as the audience for, or user of the abstraction being defined

A Class for Movies

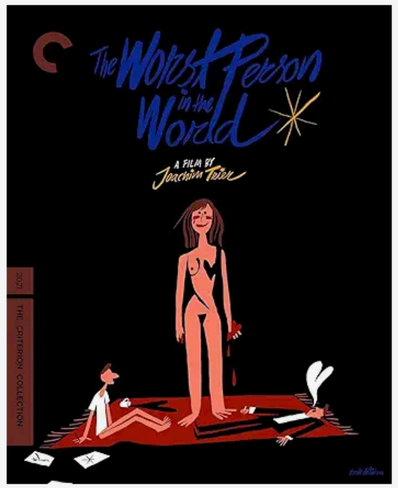
...and no, I'm not talking about Cinema Studies!

Suppose we are writing software for an online storefront
and we want to sell digital & physical copies of movies.

- **The entity:** a Movie
- **The stakeholders:** someone shopping for a movie on an online storefront

An Example of an Abstraction of a Movie

Overall Pick



The Worst Person in the World (The Criterion Collection) [Blu-ray]

★★★★★ 183

Blu-ray

$27.10 List: ~~$39.95~~

prime Two-Day
FREE delivery **Wed**

More Buying Choices
$20.78 (16 used & new offers)

DVD

$16.81 List: ~~$29.95~~

prime One-Day
FREE delivery **Tomorrow**
Only 15 left in stock (more on the way).

More Buying Choices
$9.95 (20 used & new offers)

- Starring: Renate Reinsve , Anders Danielsen Lie , Herbert Nordrum , et al.
- Directed by: Joachim Trier

A Class for Movies

Entity: Movie

Properties:

- Title
- Year
- Length
- Genre
- Release type
- Price

Class Design

Entity: Movie

Properties:

- Title (`str`)
- Year (`int`)
- Length (`int`)
- Genre (`str`)
- Release type (`str`)
- Price (`float`)

Instances of the Movie Class

- The header of the table represents the properties that all entities of this class will have.
- Each row represents an individual instance of the class
  - All movies can be described using the same properties
  - Different movies have different values for those properties

Title	Year	Length	Genre	Release	Price
"Moneyball"	2011	133	"Sports"	"Blu-ray"	15.00
"Gone With the Wind"	1939	219	"Drama"	"Streaming"	10.95
"Jurassic Park"	1993	127	"SciFi"	"DVD"	12.50
"Pirates of the Caribbean"	2003	143	"Comedy"	"Blu-ray"	17.50

Classes and Objects

Classes define data types that represent abstractions of real or imaginary entities.

We can create instances of these classes in our programs called **objects.**

- (hence: Object Oriented Programming)
- An object always belongs to a class and therefore has all of the properties that the class specifies.
- Any individual object may have different values for those properties than any other object of that type.

CIS 1100

Data Classes

Python
Fall 2024
University of Pennsylvania

`dataclasses`

In Python, a `dataclass` is the simplest kind of class.

- Defined (in most basic case) just by what properties that members of this class should have.
- Behaves very similar to a `dict` or a row of a `DataFrame`,
  except that we define the keys ahead of time.

```
# Our movie entity modeled by a dictionary
favorite_movie = {"name" : "The Worst Person in the World",
                  "year" : 2022, "length" : 132, "genre" : "Drama",
                  "release" : "Blu-ray", "price" : 27.10}
```

Movie as a `dataclass`

Going to work through the features & syntax of `dataclasses`
with examples first, then recap the rules at the end...

```python
from dataclasses import dataclass

@dataclass
class Movie:
    name: str
    year: int
    length: int
    genre: str
    release: str
    price: float
```

Defines a new data type for a `Movie` class.

Creating Movie Instances

This snippet creates two new `Movie` instances:

```
favorite_movie = Movie('The Worst Person in the World', 2022,
                       132, 'Drama', 'Blu-ray', 27.1)

not_so_good_movie = Movie('Trap', 2024, 108, 'Thriller', 'Streaming', 9.99)
```

- Provide the properties of each `Movie` as argument to the `Movie()` function to initialize a new `Movie`
  - Give the properties in the order that the `dataclass` has them defined
  - Defining the `dataclass` makes this special function available for us automatically

Getting the Attributes of a Movie

The **attributes** of an object are the names for the variables that store the properties of an object

- often we'll elide the difference between an attribute and a property—not that important
- access with `object_name.attribute_name`, e.g.:

```
favorite_movie.length
not_so_good_movie.title
```

Remember: a class defines attributes, but only an object

has values for those attributes. This doesn't work:

```
>>> Movie.price
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Movie' has no attribute 'price'
```

Printing Movies

Objects that belong to a `dataclass` can be printed easily:

```
>>> print(not_so_good_movie)
Movie(name='Trap', year=2024, length=109, genre='Thriller', release='Streaming', price=9.99)
```

The string representation points out how an instance of a `dataclass` is a lot like a `tuple` with named values.

Passing Movies Around

Since we know what attributes a `Movie` object will have, we could write functions that expect inputs with type `Movie`:

```
def stylize_title(movie):
    return f"{movie.name} ({movie.year})"
```

```
>>> stylize_title(favorite_movie)
'The Worst Person in the World (2022)'
>>> stylize_title(not_so_good_movie)
'Trap (2024)'
```

We'll come back to classes & functions soon.

Modifying Attributes

The attributes of a `dataclass` are modifiable the same way variables are usually modifiable.

Say we need to reduce the price of a `Movie` in our catalogue because it's not so good...

```
>>> not_so_good_movie.price
9.99
>>> not_so_good_movie.price = 5.99
>>> not_so_good_movie.price
5.99
```

A `dataclass` in General:

Create by giving the `dataclass` a name and specifying the set of attributes with their types:

```
@dataclass
class MyClass:
    attr_one: type
    attr_two: type
    ...
```

Access (either to read or write) attributes of a `dataclass` by referring to the attribute name of **a particular object** of the class:

```
my_object.attr_one
my_object.attr_two += 13
```

Print objects that belong to a `dataclass` with `print(my_object)`.

CIS 1100

Classes

Python
Fall 2024
University of Pennsylvania

`dataclass` vs. `class`

- Defining `dataclass` is a quick way of creating a new data type defined **entirely** by what it knows.
  - Makes assumptions about how we want to create objects of this type
  - i.e. assumes that we know all the values of the attributes when we want to create an instance
- A `class` can be written in a way that gives us control over object creation
  - Disclaimer: a `dataclass` is also a `class` and so it can actually do all of the next things we're going to do

Features of Classes

A `class` typically contains three essential elements:

- attribute variables that define the type's *properties*
- methods that define the type's *behaviors*
- a special initializer method to define how objects of this type should be created
  - (this is actually where the attribute variables are set)

Features of Classes

Here's a `class` for a square shape:

```
import penndraw as pd
class Square:
    def __init__(self, x, y, hsl):
        self.x_center = x
        self.y_center = y
        self.half_side_length = hsl

    def move_by(self, dx, dy):
        self.x_center += dx
        self.y_center += dy

    def draw(self):
        pd.square(self.x_center, self.y_center, self.half_side_length)
```

`__init__`: the Initializer

- Sometimes we'll—somewhat inaccurately—call this a *constructor.*
- Special function defined within a `class` that defines how we create an object of this class.
- Like with a `dataclass`, we create a new instance of `MyClass` by calling `MyClass()`
  - e.g. create a new `Square` by calling `Square(0.5, 0.5, 0.2)`
  - `__init__` is actually the function that defines the behavior of this call 🤯

`__init__` & Attributes

The `Square`'s attribute variables are actually declared and created within the body of `__init__`!

If we call `Square(0.5, 0.5, 0.2)`:

- `x` gets the value of `0.5` in the call to `__init__`
- we declare a new variable called `x_center` that belongs to the object that we're creating
- we store the value of `x` inside of the variable `x_center` that will live as long as the object does

```
class Square:
    def __init__(self, x, y, hsl):
        self.x_center = x
        self.y_center = y
        self.half_side_length = hsl
    ...
```

What is Self?

Well, according to Aristotle (according to Wikipedia), the psyche is the core essence of a living being...

What is `self`?

Just kidding.

`self` is the name of a variable that we use to refer to the object itself.

- `self.x_center` is the `x_center` variable that belongs to the object that we're working with
- Always use `self.attr_name` to refer to the attribute called `attr_name` from **within** the `class`

Don't Forget Your `self`!

```
class Square:
    def __init__(self, x, y, hsl):
        self.x_center = x
        self.y_center = y
        self.half_side_length = hsl
        self.area = (2 * half_side_length) * (2 * half_side_length)
```

will lead to a `NameError` down the line:

```
NameError: name 'half_side_length' is not defined
```

Don't Forget Your `self`!

Better:

```
class Square:
    def __init__(self, x, y, hsl):
        self.x_center = x
        self.y_center = y
        self.half_side_length = hsl
        self.area = (2 * self.half_side_length) * (2 * self.half_side_length)
```

Accessing Attribute Variables

If we define `__init__` like we have, then we can create new `Square` objects and access their properties:

```
small = Square(0.3, 0.3, 0.02)
big = Square(0.8, 0.8, 0.2)

small_perimeter = small.half_side_length * 2 * 4
big_center = (big.x_center, big.y_center)
```

Outside of the `class`, you access an object's attribute variables by appending `.attr_name` to the end of the object's name.

CIS 1100

Methods

Python
Fall 2024
University of Pennsylvania

Methods

Functions defined within a class are the **methods** that objects of that class can perform.

- These represent the *behaviors* that the class' entities should be expected to perform
  - For a `Dog` class, a method might be `bark()`
  - For a `Square` class, a method might be `move_by(dx, dy)`
- Every object from a class (usually) has the same methods and the same attribute variables.
  - The values of the attributes differ between the objects.
  - ➡️ Since methods behave differently based on the attributes, they can behave differently for different objects.

Writing Methods

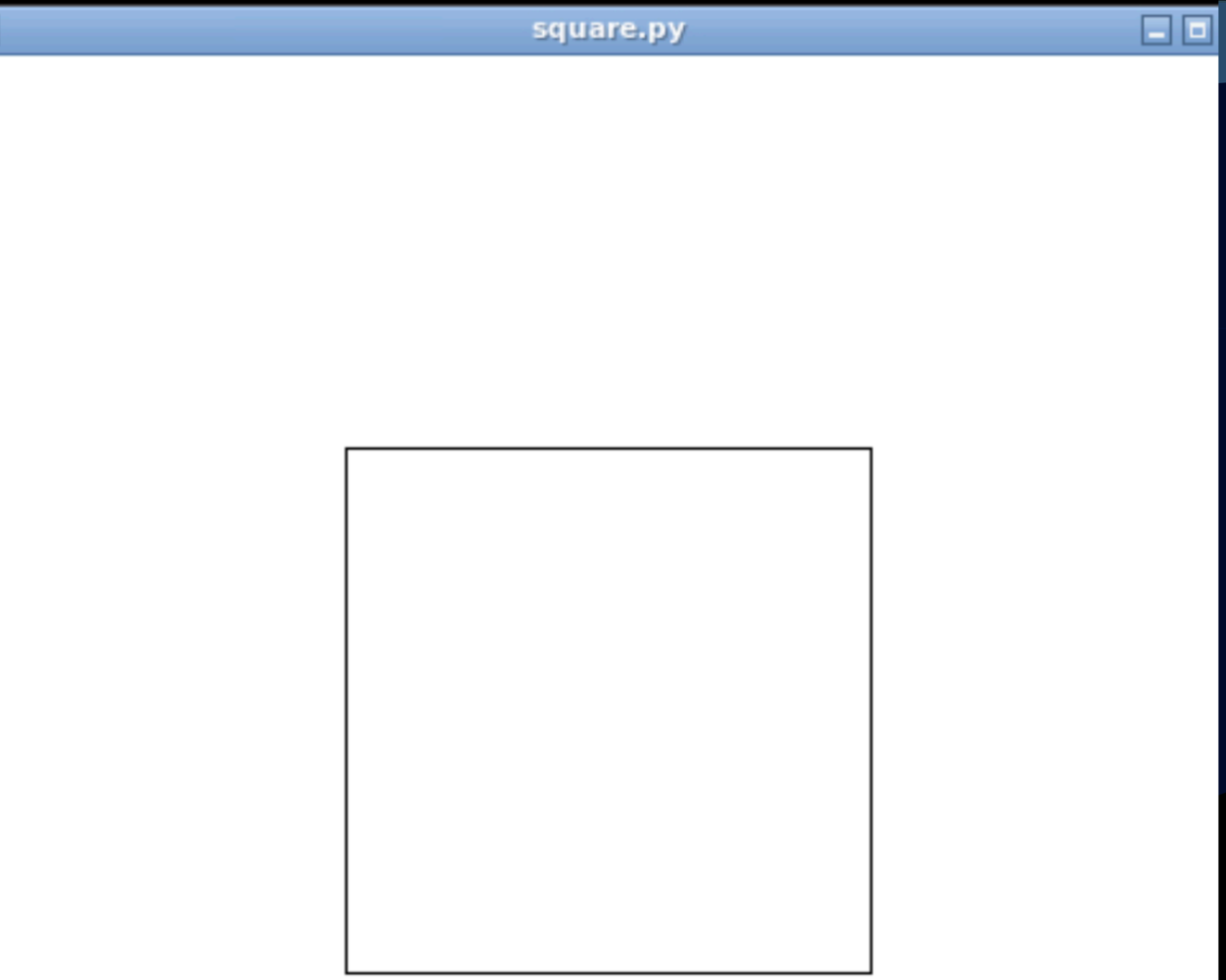
Methods are really just functions—only a couple small differences:

- First argument of a method should always be `self`
- Methods can refer to attribute variables that are declared outside the body of the method (!)

```
class Square:
    def __init__(self, x, y, hsl):
        self.x_center = x
        self.y_center = y
        self.half_side_length = hsl

    def move_by(self, dx, dy):
        self.x_center += dx     # self.x_center was declared outside of this method!
        self.y_center += dy

    def draw(self):
        pd.square(self.x_center, self.y_center, self.half_side_length)
```

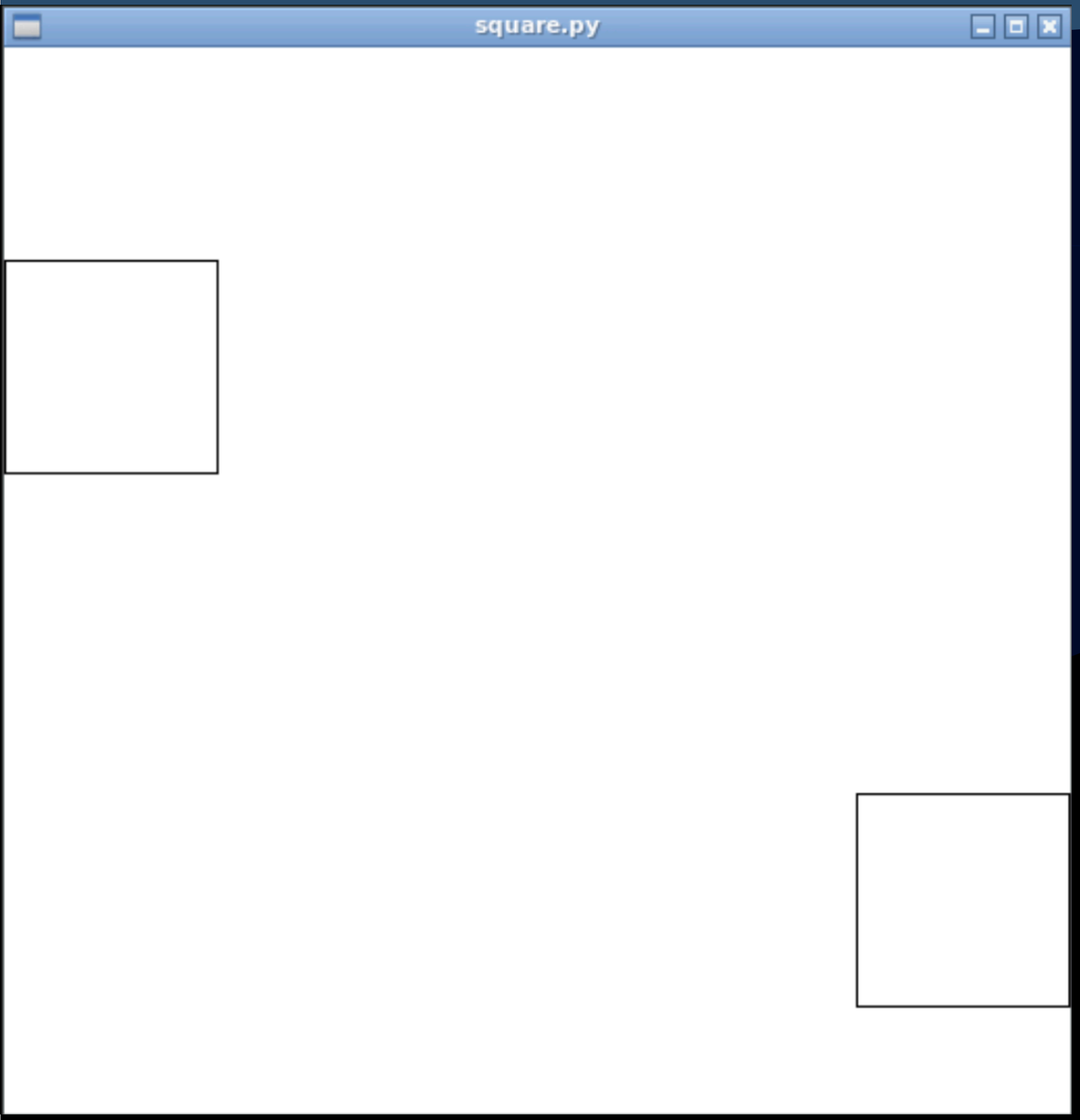


Using Methods

Methods are functions that belong to an object, so they are called (mostly) like any function

- Call by name and pass in arguments within parentheses
- Make sure to call the method *on the object* that you want to perform that behavior!

```
my_square = Square(0.4, 0.4, 0.2)
my_square.move_by(0.1, 0.1)
my_square.draw()
```



Methods Are Called on Individual Objects

You might have several instances of a class in your program.

- A method called on an object should *modify/use just that object.*
- Other objects will be unchanged by another object's method call.

```
left_square = Square(0.1, 0.2, 0.1)
right_square = Square(0.9, 0.2, 0.1)
left_square.move_by(0, 0.5)
left_square.draw()
right_square.draw()
```

CIS 1100

Stopwatch Demo

Python
Fall 2024
University of Pennsylvania

Building a Stopwatch

Let's design a class for a **stopwatch.** Our requirements are that:

- Each stopwatch that we create should be able to keep time separately from any other stopwatches.
- A stopwatch should be able to start a timer, display the current timer, and stop a timer.
- When we create a stopwatch, it should not be running at first

Stopwatch: Attributes

What does a stopwatch need to keep track of in order to be effective?

- It needs to be able to *display* how long it's been running, but that's a piece of information that will need to be calculated.
- A stopwatch should know whether it's currently running, when it was started, and when it was stopped.

Stopwatch: `__init__`

The attributes of a stopwatch can be `started_at`, `stopped_at`, and `is_running`. What should the initial values of these variables be?

- By requirements, `is_running` should be `False` to start
- Since the stopwatch isn't running at first, starting values of `started_at` and `stopped_at` are just placeholders.
- Don't actually need any information passed in to create a new stopwatch.
  - Different from the `Square` class where we provided initial values for each attribute—this is OK!

Stopwatch: `__init__`

```
from datetime import datetime
class Stopwatch:
    def __init__(self):
        placeholder = datetime.now()
        self.started_at = placeholder
        self.stopped_at = placeholder
        self.is_running = False
```

Stopwatch: Methods

A stopwatch should be able to **start a timer**, **display the current timer**, and **stop a timer**.

- `start()`: start the timer, or do nothing if the stopwatch is running.
- `display()`:
  - if the timer is running, show the time elapsed since the stopwatch was started
  - if the timer is not running, show the time elapsed between `start()` and `stop()`
- `stop()`: stop the timer if it is running, or do nothing if it is not.

Stopwatch: Methods

```
from datetime import datetime
class Stopwatch:
    ... # __init__ omitted for space
    def start(self):
        if not self.is_running:
            self.started_at = datetime.now()
            self.is_running = True

    def stop(self):
        if self.is_running:
            self.stopped_at = datetime.now()
            self.is_running = False

    def display(self):
        if self.is_running:
            elapsed = datetime.now() - self.started_at
            verb = "Running"
        else:
            elapsed = self.stopped_at - self.started_at
            verb = "Ran"
        print(f"{verb} for {elapsed.total_seconds()} seconds.")
```

Stopwatch: Demo