FITNESS TRACKER OBJECT ORIENTED PROGRAMMING EXAMPLE

(download slides and .py files to follow along)

6.100L Lecture 20

Ana Bell

IMPLEMENTING THE CLASS vs USING THE CLASS

Implementing a new object type with a class

- **Define** the class
- Define **data attributes** (WHAT IS the object)
- Define **methods** (HOW TO use the object)

Class abstractly captures **common** properties and behaviors

Using the new object type in code

- Create **instances** of the object type
- Do **operations** with them

Instances have **specific values** for attributes

Two different coding perspectives

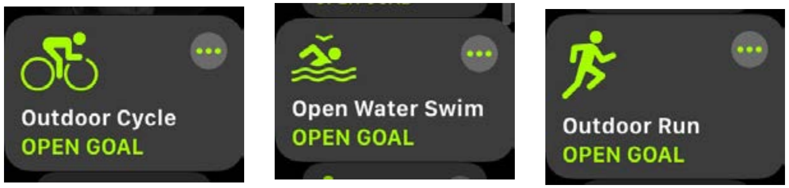
Workout Tracker Example

Thanks to Sam Madden for this OOP example (his slides have been modified)

- Suppose we are writing a program to track workouts, e.g., for a smart watch

Different kinds of workouts

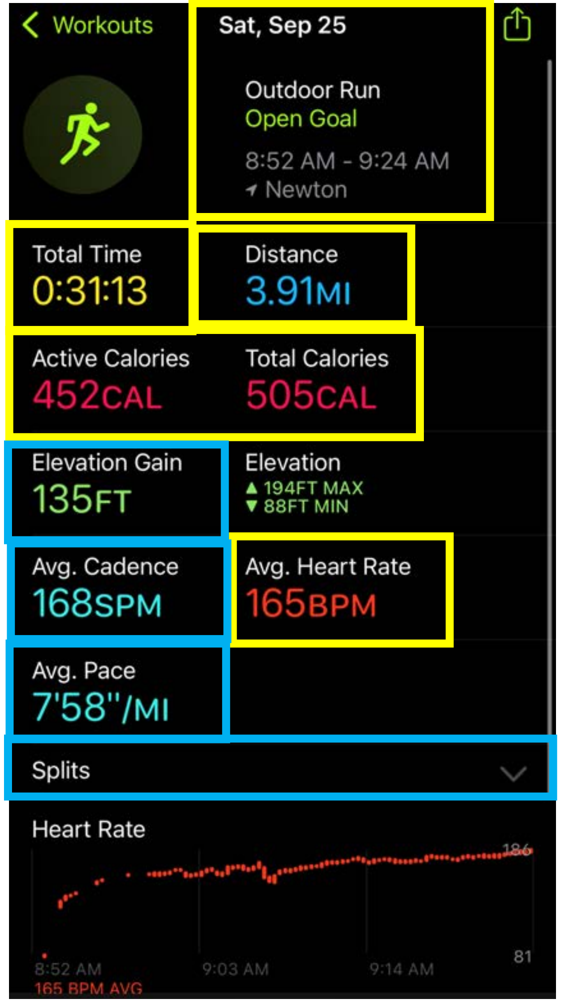
Fitness Tracker


Outdoor Cycle
OPEN GOAL

Open Water Swim
OPEN GOAL

Outdoor Run
OPEN GOAL

Different types of workouts


< Workouts

Sat, Sep 25

Outdoor Run
Open Goal
8:52 AM - 9:24 AM
Newton

Total Time
0:31:13

Distance
3.91MI

Active Calories
452CAL

Total Calories
505CAL

Elevation Gain
135FT

Elevation
▲ 194FT MAX
▼ 88FT MIN

Avg. Cadence
168SPM

Avg. Heart Rate
165BPM

Avg. Pace
7'58"/MI

Splits

Heart Rate

Common properties:

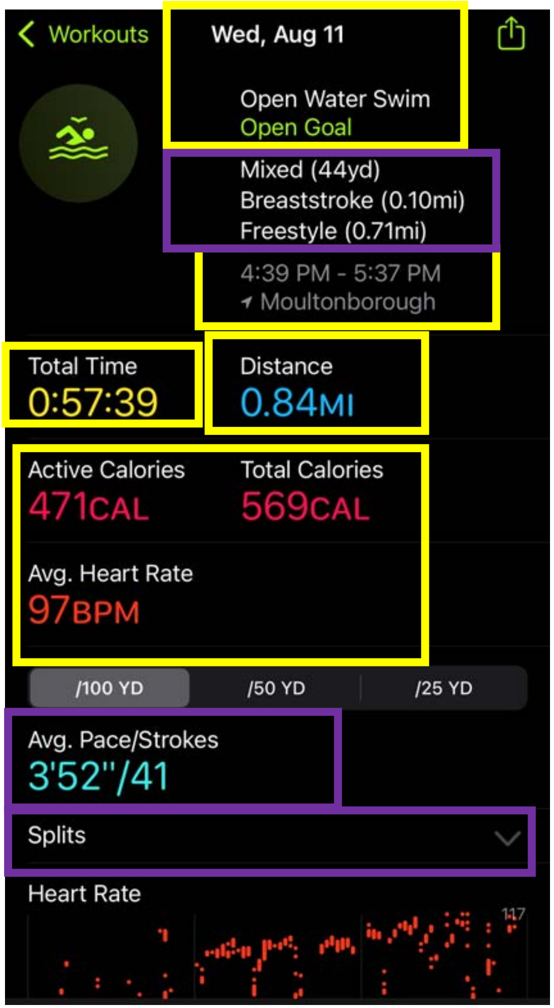
Icon *Kind*
Date *Start Time*
End Time *Calories*
Heart Rate Distance

Swimming Specific:

Swimming Pace
Stroke Type
100 yd Splits

Running Specific:

Cadence
Running Pace
Mile Splits
Elevation


< Workouts

Wed, Aug 11

Open Water Swim
Open Goal
Mixed (44yd)
Breaststroke (0.10mi)
Freestyle (0.71mi)
4:39 PM - 5:37 PM
Moultonborough

Total Time
0:57:39

Distance
0.84MI

Active Calories
471CAL

Total Calories
569CAL

Avg. Heart Rate
97BPM

/100 YD /50 YD /25 YD

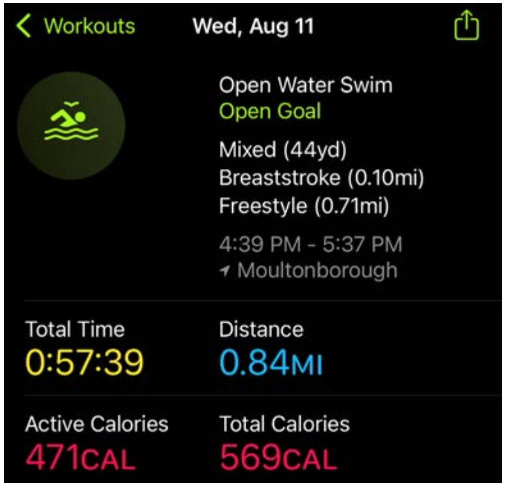
Avg. Pace/Strokes
3'52"/41

Splits

Heart Rate

GROUPS OF OBJECTS HAVE ATTRIBUTES (RECAP)

- **Data attributes**
  - How can you represent your object with data?
  - **What it is**
  - *for a coordinate: x and y values*
  - *for a workout: start time, end time, calories*
- **Functional attributes** (behavior/operations/**methods**)
  - How can someone interact with the object?
  - **What it does**
  - *for a coordinate: find distance between two*
  - *for a workout: display an information card*



Workouts

Wed, Aug 11

Open Water Swim

Open Goal

Mixed (44yd)

Breaststroke (0.10mi)

Freestyle (0.71mi)

4:39 PM - 5:37 PM

Moultonborough

Total Time

0:57:39

Distance

0.84MI

Active Calories

471CAL

Total Calories

569CAL

DEFINE A SIMPLE CLASS (RECAP)

class definition

name

class parent

variable to refer to an instance of the class

Start and end time, and calories burned

```
class Workout(object):
      def __init__(self, start, end, calories):
          self.start = start
          self.end = end
          self.calories = calories
          self.icon = '😓'
          self.kind = 'Workout'

my_workout = Workout('9/30/2021 1:35 PM', '9/30/2021 1:57 PM', 200)
```

“constructor” - special method to create an instance

Icon and kind are attributes even though an instance is not initialized with them as a param (And python strings can contain emojis! 🤯)

one instance

Mapped to start, end, calories in constructor

GETTER AND SETTER METHODS (RECAP)

```
class Workout(object):
    def __init__(self, start, end, calories):
        self.start = start
        self.end = end
        self.calories = calories
        self.icon = '😓'
        self.kind = 'Workout'
```

getter

```
    def get_calories(self):
        return self.calories
    def get_start(self):
        return self.start
    def get_end(self):
        return self.end
```

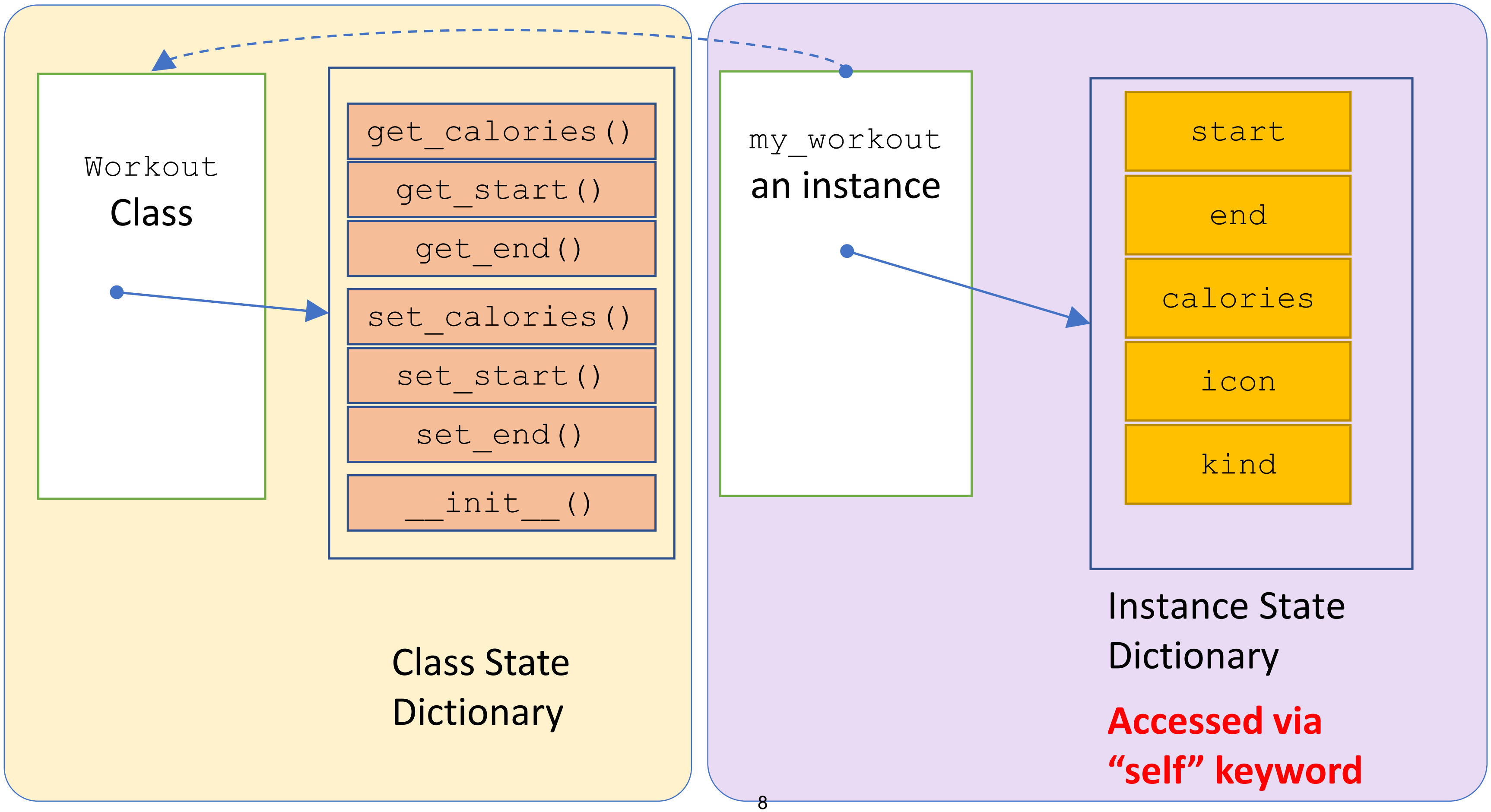
setter

```
    def set_calories(self, calories):
        self.calories = calories
    def set_start(self, start):
        self.start = start
    def set_end(self, end):
        self.end = end
```

Getters and setters used outside of class to access data attributes

SELF PROVIDES ACCESS TO CLASS STATE

```
my_workout = Workout('9/30/2021 1:35 PM', 9/30/2021 1:57 PM', 200)
```



AN INSTANCE and DOT NOTATION (RECAP)

- Instantiation creates an **instance of an object**

```
myWorkout = Workout('9/30/2021 1:35 PM', '9/30/2021 1:57 PM', 200)
```

- **Dot notation** used to access attributes (data and methods)
- It's better to use getters and setters to access data attributes

```
my_workout.calories
```

- access data attribute directly
- allowed, but not recommended

```
my_workout.get_calories()
```

- access attribute via method
- better, because it supports information hiding

WHY INFORMATION HIDING?

- Keep the **interface** of your class as **simple** as possible
- Use getters & setters, not attributes
  - i.e., `get_calories()` method NOT `calories` attribute
  - Prevents bugs due to changes in implementation
- May seem **inconsequential in small programs**, but for large programs complex interfaces increase the potential for bugs
- If you are writing a class for others to use, you are **committing to maintaining its interface**!

CHANGING THE CLASS IMPLEMENTATION

- Author of class definition may **change internal representation or implementation**
  - Use a class variable
  - Now `get_calories` estimates calories based of workout duration if calories are not passed in
- If **accessing data attributes** outside the class and class **implementation changes**, may get errors

Demo

CHANGING THE CLASS IMPLEMENTATION

```
class Workout:
    cal_per_hr = 200

    def __init__(self, start, end, calories=None):
        self.start = parser.parse(start)
        self.end = parser.parse(end)
        self.calories = calories # may be None
        self.icon = '😓'
        self.kind = 'Workout'


    def get_calories(self):
        if (calories == None):
            return Workout.cal_per_hr*(self.end-self.start).total_seconds()/3600
        else:
            return self.calories
```

Class variable – all instances of Workout can read this

Defaults to None if not passed in

self.start and self.end are objects of type datetime, not strings

If calories was not passed in, estimate based on elapsed time

Allowed on datetime objects

If calories was passed in, just use that value

ASIDE: `datetime` OBJECTS
OTHER PYTON LIBRARIES

- Takes the string representing the date and time and **converts it to a `datetime` object**

```
from dateutil import parser
```

Brings in a bunch of functions and classes

```
start = '9/30/2021 1:35 PM'
end = '9/30/2021 1:45 PM'
start_date = parser.parse(start)
end_date = parser.parse(end)
```

```
type(start_date)
```

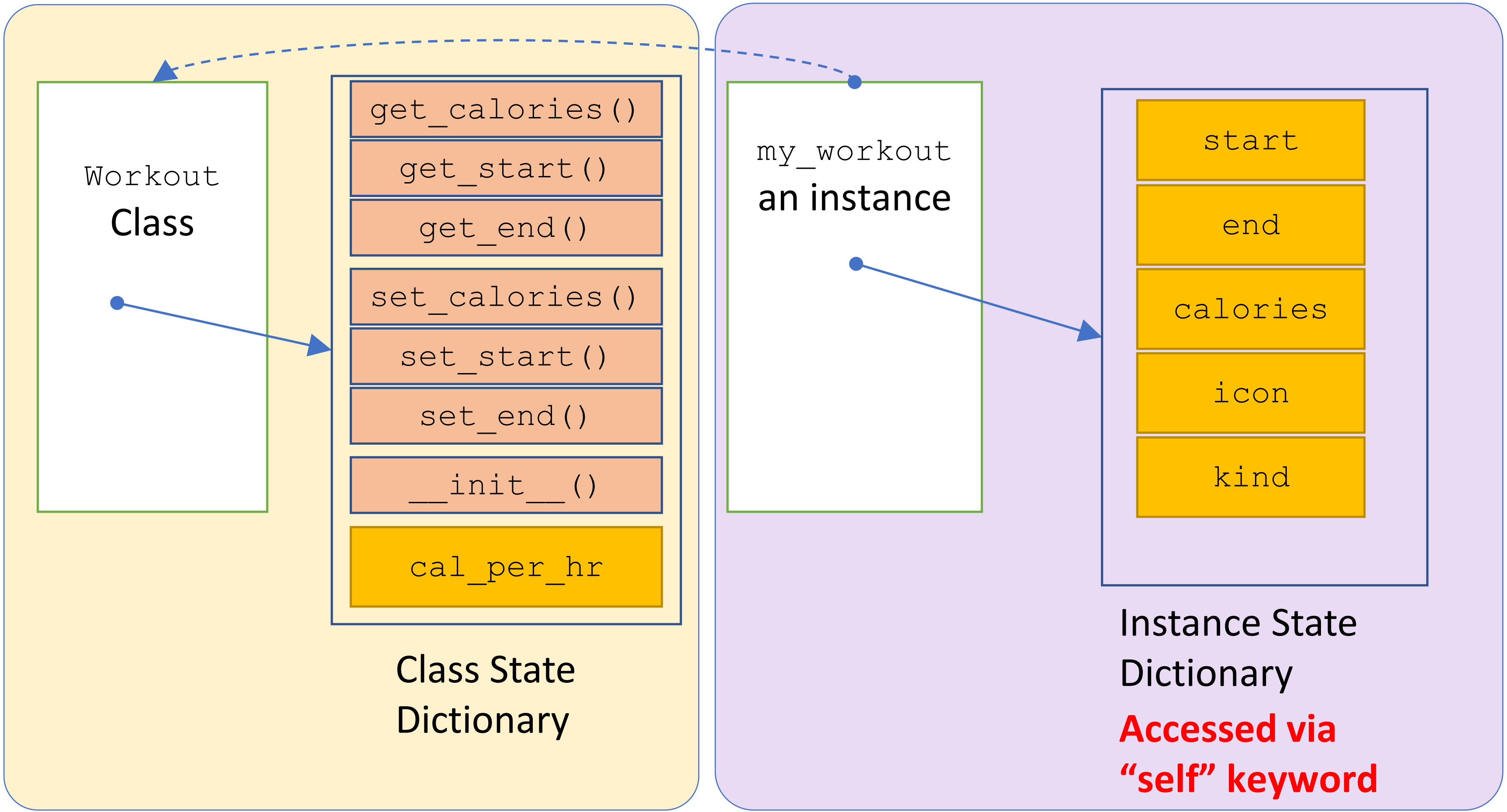
Type is `datetime.datetime`

- Why do this? Because it **makes operations with dates easy**! The datetime object takes care of everything

```
print((end_date-start_date).total_seconds())
```

Prints 600

CLASS VARIABLES LIVE IN CLASS STATE DICTIONARY



Accessed via “self” keyword

CLASS VARIABLES

- Associate a **class variable with all instances** of a class
- Warning: if an instance changes the class variable, it’s changed for all instances

```
class Workout:
    cal_per_hr = 200
    def __init__(self, start, end, calories):
        …
```

cal_per_hr is set to 200 for all new instances of Workout

```
print(Workout.cal_per_hr)
```

No instance required, prints 200

```
w = Workout('1/1/2021 2:34', '1/1/2021 3:35', None)
print(w.cal_per_hr)
```

Prints 200

```
Workout.cal_per_hr = 250
print(w.cal_per_hr)
```

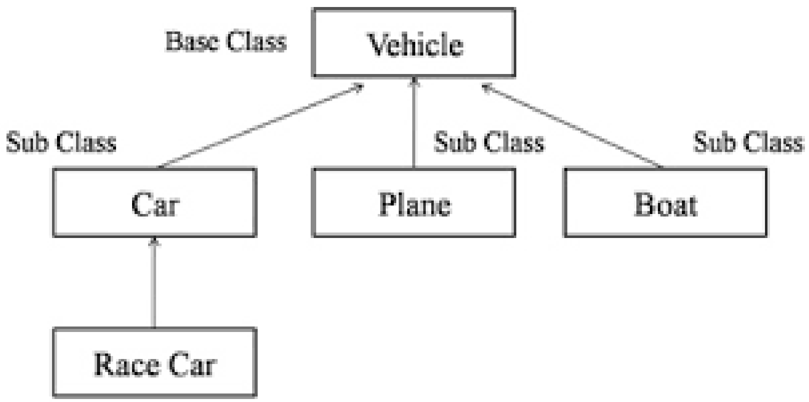
Prints 250

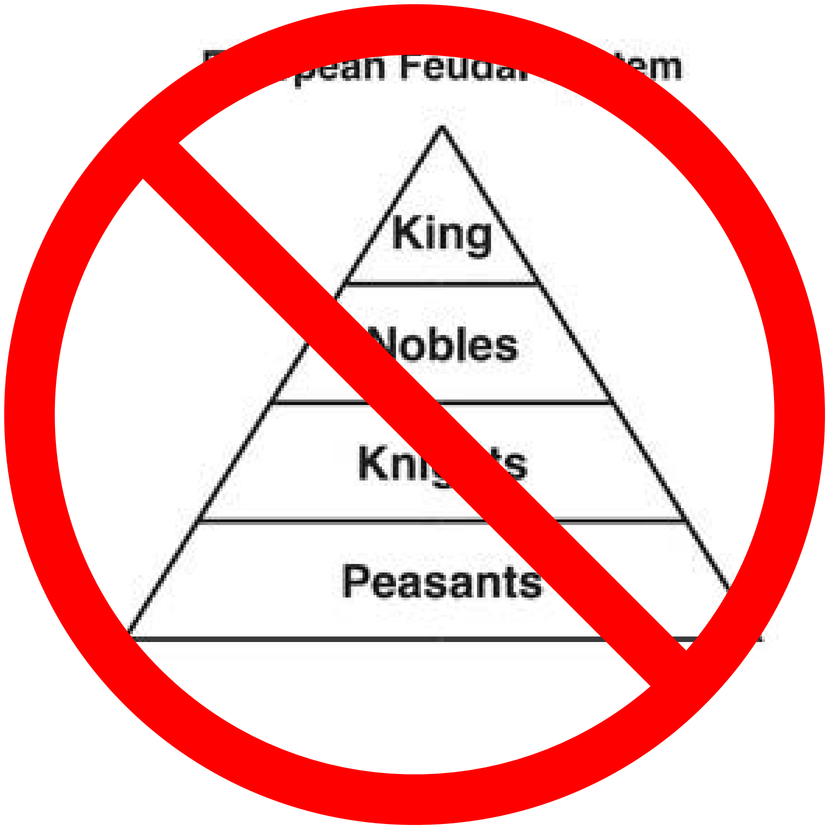
Bad style to change the class variable outside the class definition. Write a method to do it!

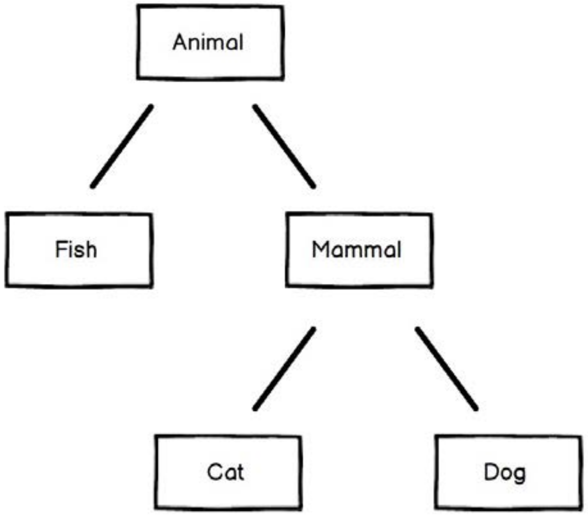
YOU TRY IT!

- Write lines of code to create two Workout objects.
  - One Workout object saved as variable `w_one`, from `Jan 1 2021 at 3:30 PM` until `4 PM`. You want to estimate the calories from this workout. Print the number of calories for `w_one.`
  - Another Workout object saved as `w_two`, from `Jan 1 2021 at 3:35 PM` until `4 PM`. You know you burned `300` calories for this workout. Print the number of calories for `w_two`.

NEXT UP: CLASS HIERARCHIES

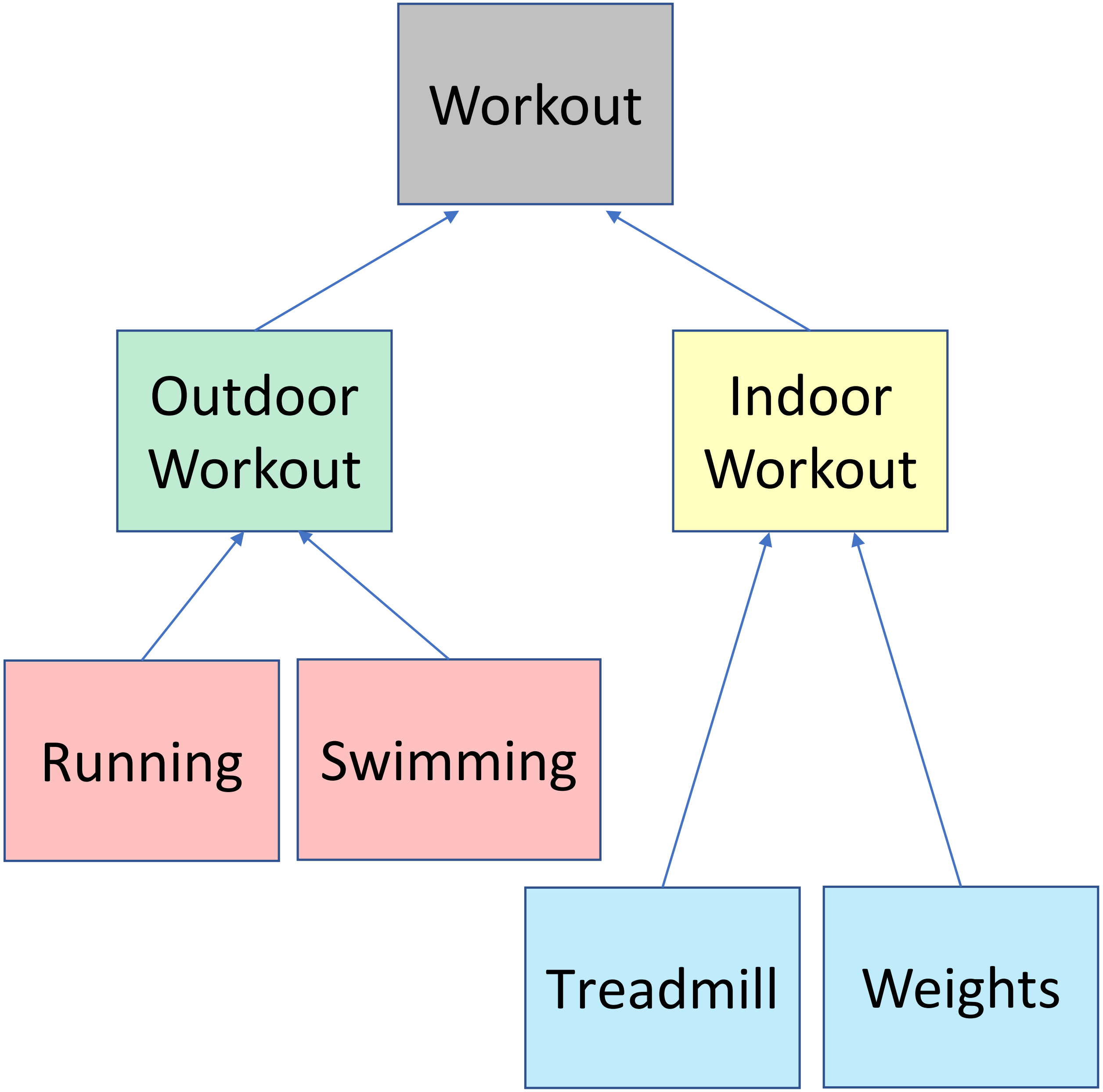






HIERARCHIES

- **Parent class** (superclass)
- **Child class** (subclass)
  - **Inherits** all data and behaviors of parent class
  - **Add** more **info**
  - **Add** more **behavior**
  - **Override** behavior



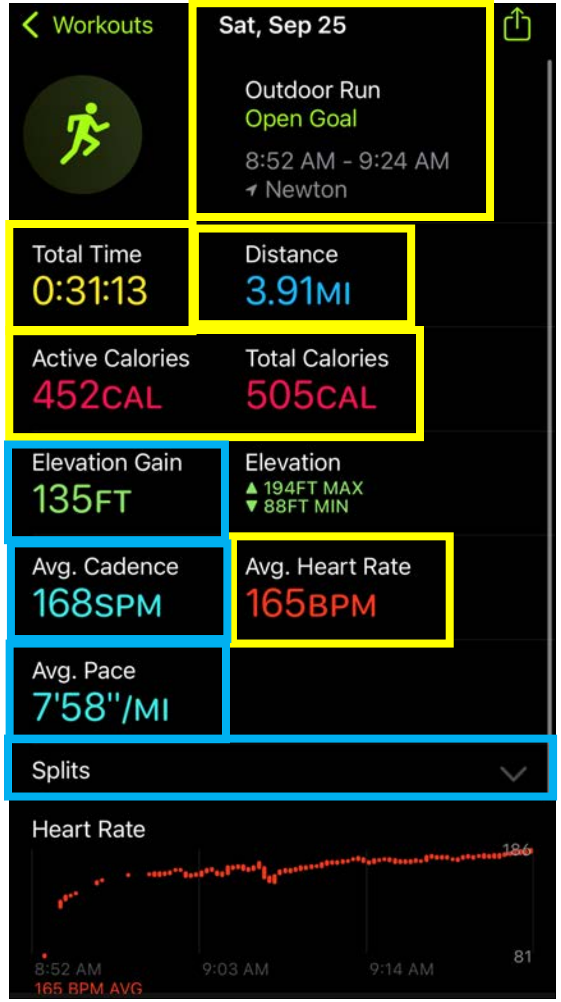
Fitness Tracker

Outdoor Cycle
OPEN GOAL

Open Water Swim
OPEN GOAL

Outdoor Run
OPEN GOAL

Different kinds of workouts


< Workouts

Sat, Sep 25

Outdoor Run
Open Goal
8:52 AM - 9:24 AM
Newton

Total Time
0:31:13

Distance
3.91MI

Active Calories
452CAL

Total Calories
505CAL

Elevation Gain
135FT

Elevation
▲194FT MAX
▼88FT MIN

Avg. Cadence
168SPM

Avg. Heart Rate
165BPM

Avg. Pace
7'58''/MI

Splits

Heart Rate

Common properties:

Icon
Date
T
Er
He

Kind
rt
Calories
ate
Distance

Workout “Superclass”

Swimming Specific:

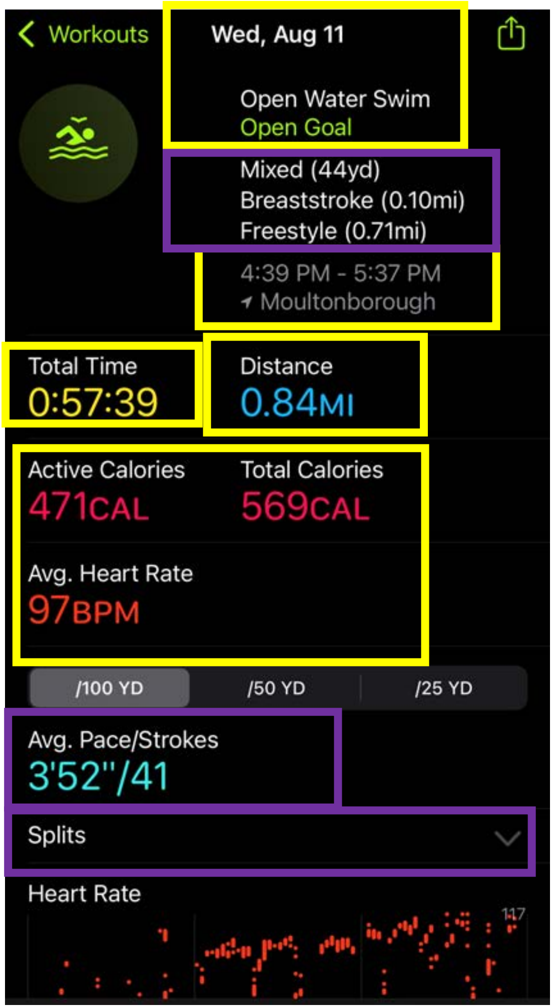
Swimming Pac
Stroke T
1

Swimming “Subclass”

Running Specific:

Cadence
Runni
M
Ele

Running “Subclass”


< Workouts

Wed, Aug 11

Open Water Swim
Open Goal

Mixed (44yd)
Breaststroke (0.10mi)
Freestyle (0.71mi)

4:39 PM - 5:37 PM
Moultonborough

Total Time
0:57:39

Distance
0.84MI

Active Calories
471CAL

Total Calories
569CAL

Avg. Heart Rate
97BPM

/100 YD | /50 YD | /25 YD

Avg. Pace/Strokes
3'52''/41

Splits

Heart Rate

INHERITANCE: PARENT CLASS

```
class Workout(object):
    cal_per_hr = 200
    def __init__(self, start, end, calories=None):
        …
```

- Everything is an object
- Class `object` implements basic operations in Python, e.g., binding variables

INHERITANCE: SUBCLASS

Parent is Workout
Inherits all attributes of `Workout`:
start,end,calories
get_calories(), get_start()
get_end(),… ,__str__()

```
class RunWorkout(Workout):

    def __init__(self, start, end, elev=0, calories=None):
        super().__init__(start,end,calories)
        self.icon = '🏃'
        self.kind = 'Running'
        self.elev = elev

    def get_elev(self):
        return self.elev
    def set_elev(self, e):
        self.elev = e
```

Parent accessed via super()

Initialize the parent class (Workout)

Add new instance variables

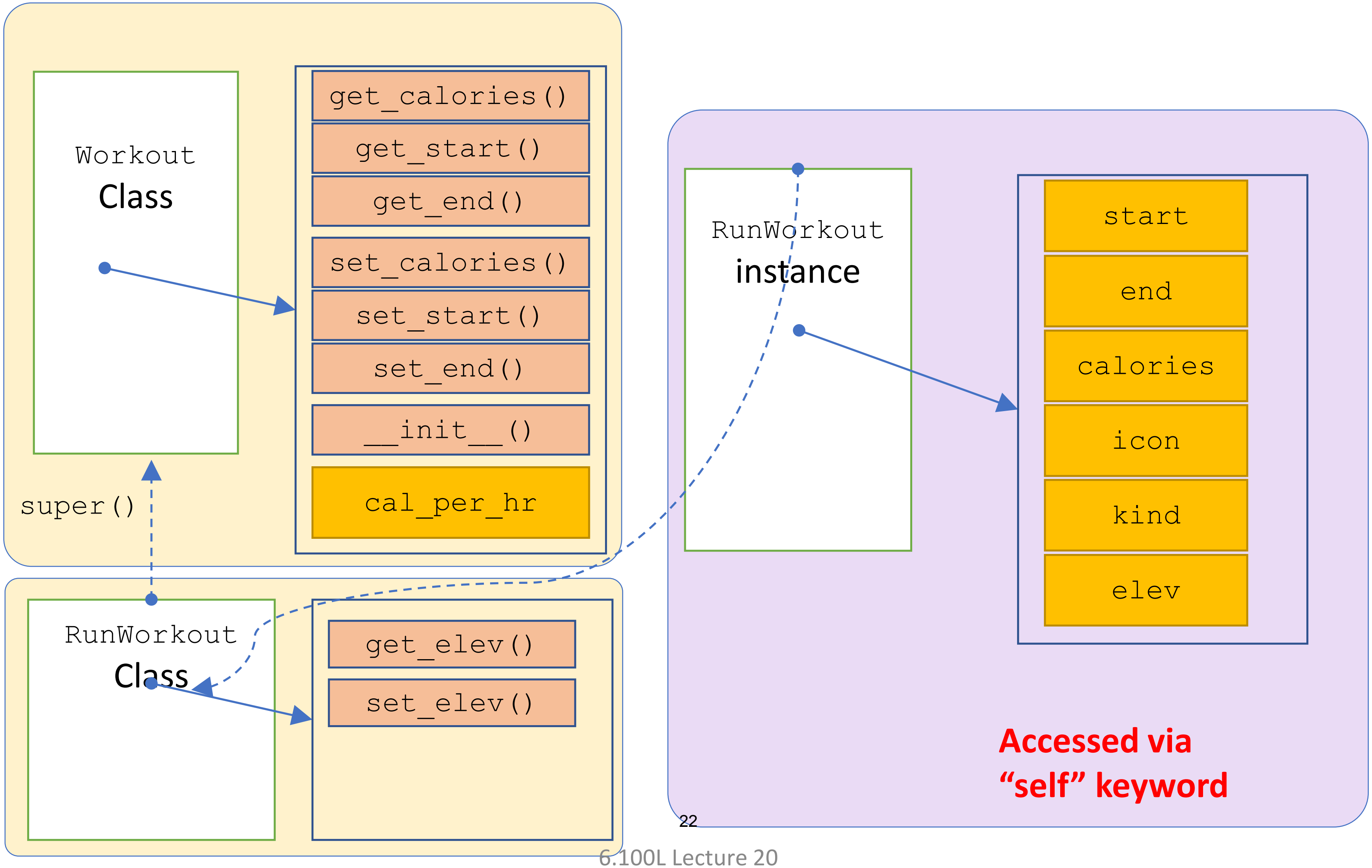
Override parents default

Add new functionality

Add **new functionality** e.g., `get_elev()`

- New methods can be called on instance of type `RunWorkout`
- `__init__` uses `super()` to setup Workout base instance (can also call `Workout.__init__(start,end,calories)` directly

INHERITANCE REPRESENTATION IN MEMORY



Accessed via "self" keyword

WHY USE INHERITENCE?

- Improve **clarity**
  - Commonalities are explicit in parent class
  - Differences are explicit in subclass
- **Reuse** code
- Enhance **modularity**
  - Can pass subclasses to any method that uses parent

SUBCLASSES REUSE PARENT CODE

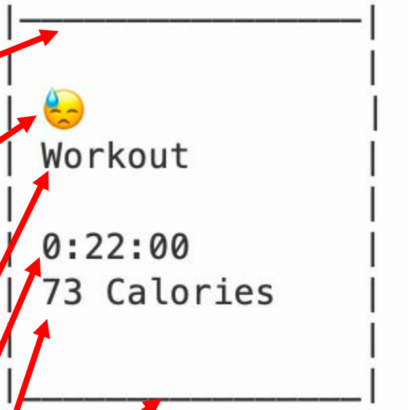
- Complex print function shared by all subclasses

```
class Workout(object)
………
    def __str__(self):
        width = 16
        retstr =  f"|{'-'*width}|\n"
        retstr += f"|{' ' *width}|\n"
        iconLen = 0
        retstr += f"| {self.icon}{' '*(width-3)}|\n"
        retstr += f"| {self.kind}{' '*(width-len(self.kind)-1)}|\n"
        retstr += f"|{' ' *width}|\n"
        duration_str = str(self.get_duration())
        retstr += f"| {duration_str}{' '*(width-len(duration_str)-1)}|\n"
        cal_str = f"{self.get_calories():.0f}"
        retstr += f"| {cal_str} Calories {' '*(width-len(cal_str)-11)}|\n"

        retstr += f"|{' ' *width}|\n"
        retstr +=  f"|{'_'*width}|\n"

        return retstr
```

outputs



```
|----------------|
|                |
| 😓             |
| Workout        |
|                |
| 0:22:00        |
| 73 Calories    |
|                |
|________________|
```

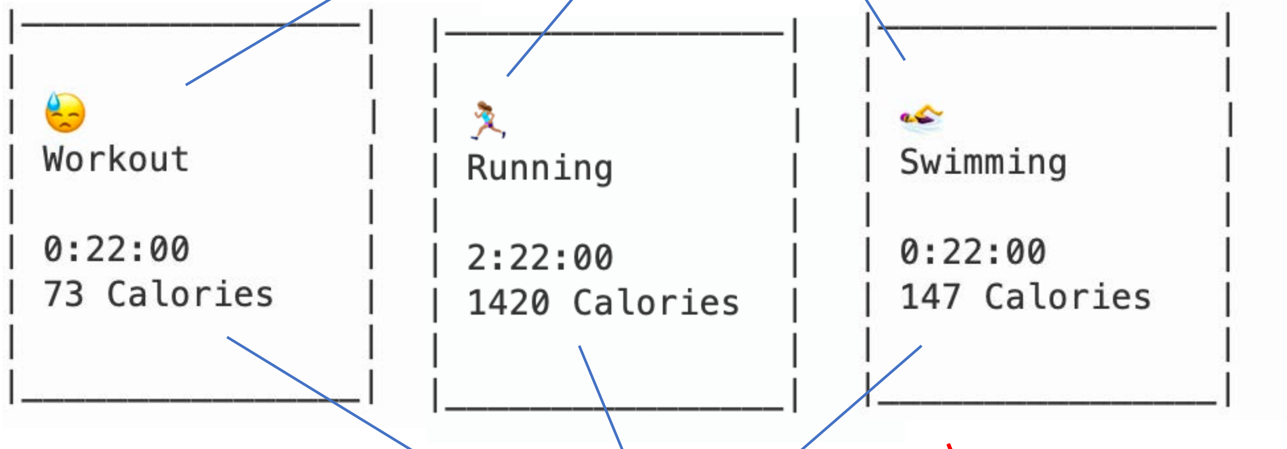
Demo

SUBCLASSES REUSE PARENT CODE

```
w=Workout(…)
rw=RunWorkout(…)
sw=SwimWorkout(…)

print(w)
print(rw)
print(sw)
```

All Workout subclasses can use Workout `__str__()` method!



```
|___________________|
|                   |
| 😓                 |
| Workout           |
|                   |
| 0:22:00           |
| 73 Calories       |
|                   |
|___________________|
```

```
|___________________|
|                   |
| 🏃‍♀️                 |
| Running           |
|                   |
| 2:22:00           |
| 1420 Calories     |
|                   |
|___________________|
```

```
|___________________|
|                   |
| 🏊‍♀️                 |
| Swimming          |
|                   |
| 0:22:00           |
| 147 Calories      |
|                   |
|___________________|
```

Workout specific icon and label

Calories calculated based on `cal_per_hr` for each subclass

WHERE CAN I USE AN INSTANCE OF A CLASS?

- We can use an instance of `RunWorkout` anywhere `Workout` can be used
- Opposite is not true (cannot use `Workout` anywhere `RunWorkout` is used)
- Consider two helper functions

```
def total_calories(workouts):
    cals = 0
    for w in workouts:
        cals += w.get_cals()
    return cals
```

```
def total_elevation(run_workouts):
    elev = 0
    for w in run_workouts:
        elev += w.get_elev()
    return elev
```

Demo

WHERE CAN I USE AN INSTANCE OF A CLASS?

```
def total_calories(workouts):
    cals = 0
    for w in workouts:
        cals += w.get_cals()
    return cals
```

```
def total_elevation(run_workouts):
    elev = 0
    for w in run_workouts:
        elev += w.get_elev()
    return elev
```

```
w1 = Workout('9/30/2021 1:35 PM','9/30/2021 2:05 PM')
w2 = Workout('9/30/2021 4:35 PM','9/30/2021 5:05 PM')
rw1 = RunWorkout('9/30/2021 1:35 PM','9/30/2021 3:35 PM', 100)
rw2 = RunWorkout('9/30/2021 1:35 PM','9/30/2021 3:35 PM', 200)
```

30 min workouts = 100 cal

2 hr run workouts

elevation val

```
total_calories([w1,w2,rw1,rw2]))   # (1) # cal = 100+100+400+400
total_elevation([rw1,rw2]))        # (2) # elev = 100+200
total_elevation([w1,rw1])          # (3) # err! w1 has no elev method
```

YOU TRY IT!

- For each line creating on object below, tell me:
    - What is the calories val through `get_calories()`
    - What is the elevation val through `get_elev()`

```
w1 = Workout('9/30/2021 2:20 PM','9/30/2021 2:50 PM')

w2 = Workout('9/30/2021 2:20 PM','9/30/2021 2:50 PM',450)

rw1 = RunWorkout('9/30/2021 2:20 PM','9/30/2021 2:50 PM',250)

rw2 = RunWorkout('9/30/2021 2:20 PM','9/30/2021 2:50 PM',250,300)

rw3 = RunWorkout('9/30/2021 2:20 PM','9/30/2021 2:50 PM',calories=300)
```

Demo

OVERRIDING SUPERCLASSES

- Overriding superclass – add calorie calculation w/ distance

```
class RunWorkout(Workout):
    cals_per_km = 100
                    …

    def get_calories(self):
        if (self.route_gps_points != None):
            dist = 0
            lastP = self.routeGpsPoints[0]
            for p in self.routeGpsPoints[1:]:
                dist += gpsDistance(lastP,p)
                lastP = p
            return dist * RunWorkout.cals_per_km
        else:
            return super().get_calories()
```

Add another class var

route_gps_points contains lat/lon pairs of route run

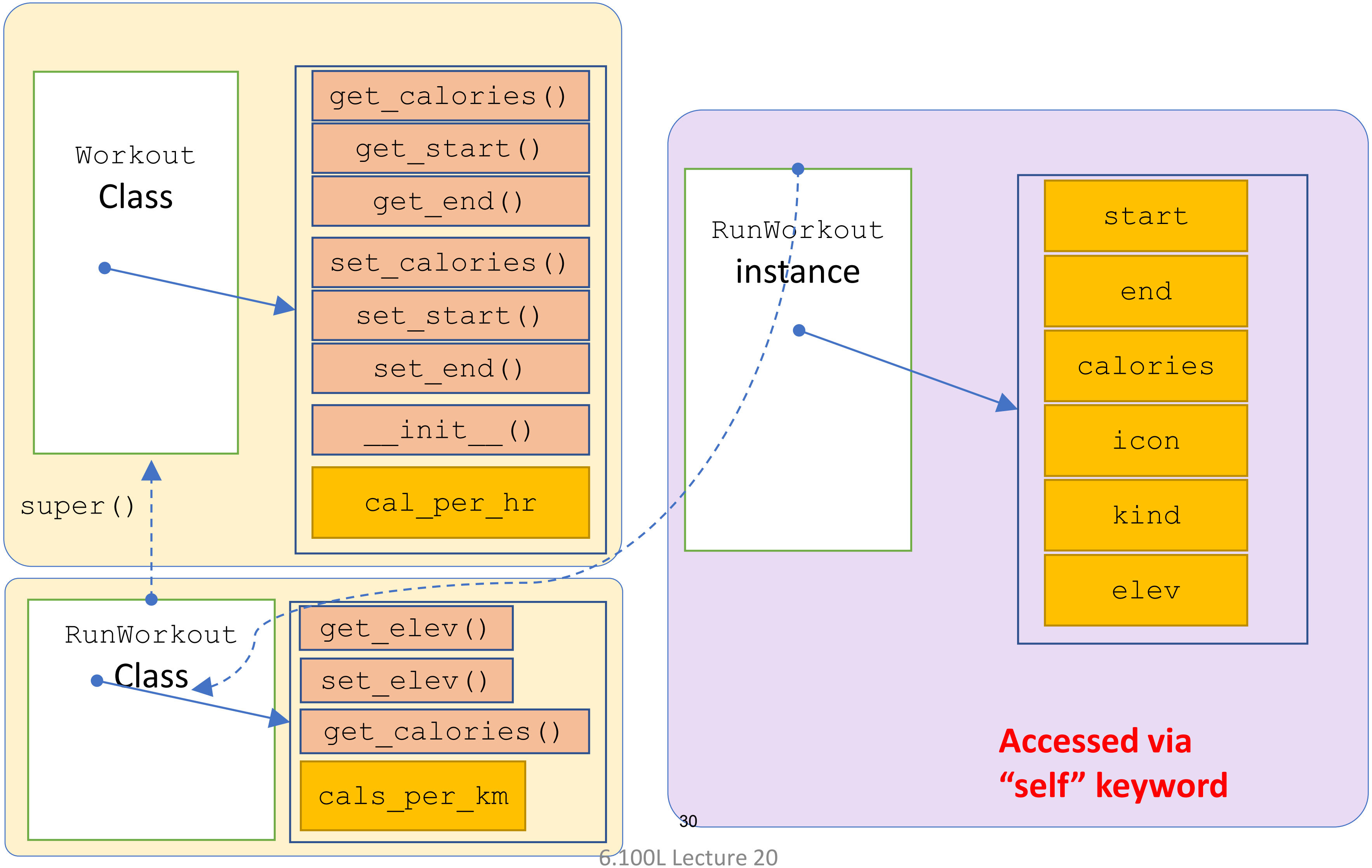
get_calories() overridden since it is defined in both sub and superclass

Iterate through all pairs of GPS points

Summing up their distance

Didn't pass in gps coords, so just do whatever the superclass does

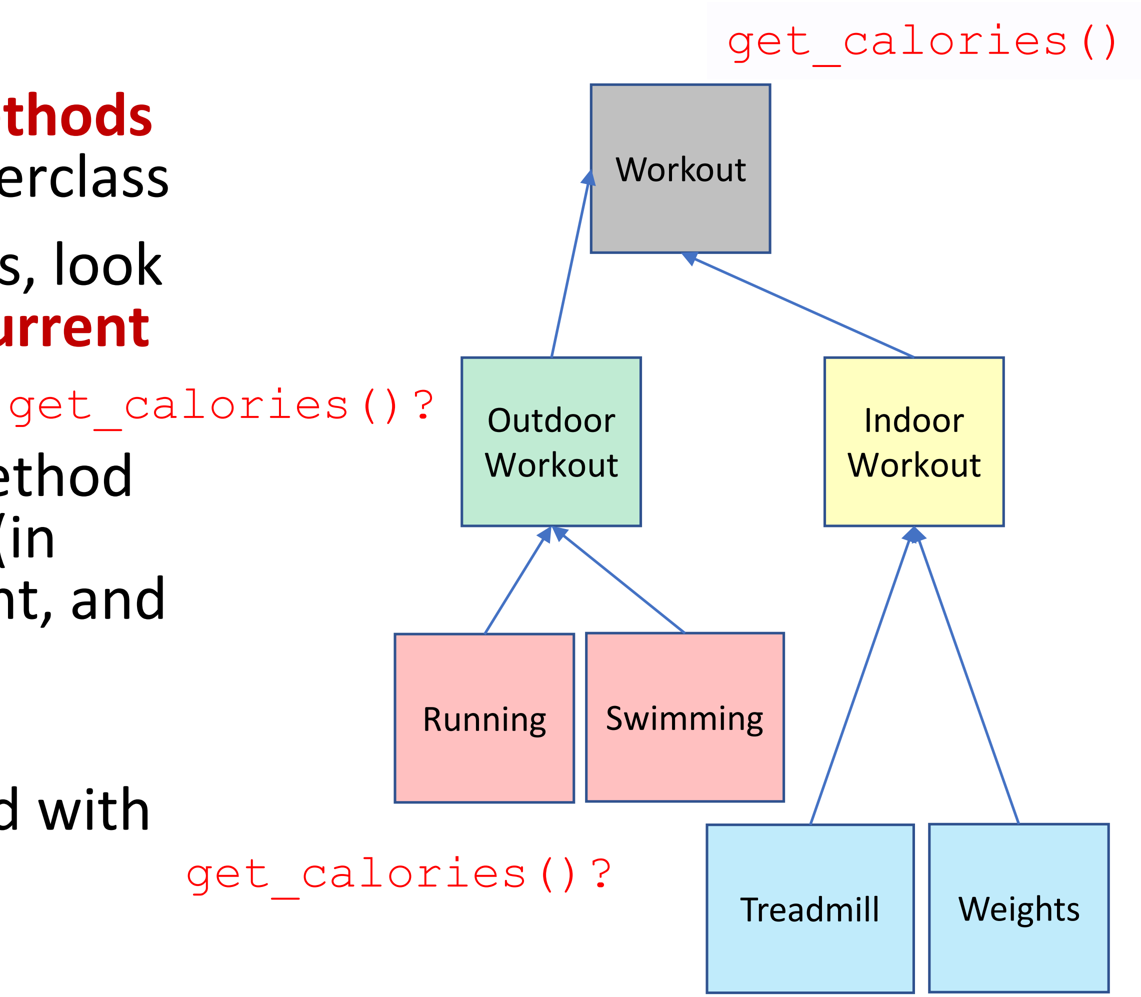
OVERRIDDEN METHODS IN MEMORY



Accessed via “self” keyword

WHICH METHOD WILL BE CALLED?

- **Overriding**: subclass **methods with same name** as superclass
- For an instance of a class, look for a method name in **current class definition**
- If not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- Use first method up the hierarchy that you found with that method name



Demo

TESTING EQUALITY WITH SUBCLASSES

- With subclasses, often want to ensure base class is equal, in addition to new properties in the subclass

```
class Workout(object):
......
     def __eq__(self, other):
        return type(self) == type(other) and \
                self.startDate == other.startDate and \
                self.endDate == other.endDate and \
                self.kind == other.kind and \
                self.get_calories() == other.get_calories()
```

Types must be the same

And all the other properties equal too

```
class RunWorkout(Workout):
......
    def __eq__(self,other):
        return super().__eq__(other) and self.elev == other.elev
```

Workout properties are equal

And new properties from RunWorkout are equal

OBJECT ORIENTED DESIGN: MORE ART THAN SCIENCE

- OOP is a powerful tool for **modularizing** your code and grouping state and functions together

BUT

- It’s **possible to overdo** it
  - New OOP programmers often create elaborate class hierarchies
  - Not necessarily a good idea
  - Think about the users of your code

    Will your decomposition make sense to them?
  - Because the function that is invoked is implicit in the class hierarchy, it can sometimes be difficult to reason about control flow
- The Internet is full of opinions OOP and “good software design” – you have to **develop your own taste through experience**!