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)

Using the new object type in code
- Create instances of the object type
- Do operations with them

Class abstractly captures common properties and behaviors

Instances have specific values for attributes

Two different coding perspectives
Suppose we are writing a program to track workouts, e.g., for a smart watch.
Different types of workouts

### 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**
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*
DEFINE A SIMPLE CLASS (RECAP)

```
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)
```
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'

    def get_calories(self):
        return self.calories

    def get_start(self):
        return self.start

    def get_end(self):
        return self.end

    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

  - access data attribute directly
  - allowed, but not recommended

  my_workout.calories

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

  my_workout.get_calories()
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
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 PYTHON LIBRARIES**

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

```python
from dateutil import parser

start = '9/30/2021 1:35 PM'
end = '9/30/2021 1:45 PM'

start_date = parser.parse(start)
end_date = parser.parse(end)

print(type(start_date))
```

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

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

Prints 600
CLASS VARIABLES LIVE IN CLASS STATE DICTIONARY

Workout Class

Class State Dictionary

get_calories()
get_start()
get_end()
set_calories()
set_start()
set_end()
__init__()
cal_per_hr

my_workout
an instance

Instance State Dictionary

start
calories
icon
kind

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

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

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

Diagram:
- **Workout**
  - **Outdoor Workout**
  - **Running**
  - **Swimming**
  - **Indoor Workout**
  - **Treadmill**
  - **Weights**
## Different kinds of workouts

<table>
<thead>
<tr>
<th>Icon Kind</th>
<th>Date</th>
<th>Start Time</th>
<th>End Time</th>
<th>Calories</th>
<th>Heart Rate</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outdoor Run</td>
<td>Sat, Sep 25</td>
<td>8:52 AM - 9:24 AM</td>
<td></td>
<td></td>
<td></td>
<td>3.91 MI</td>
</tr>
<tr>
<td>Outdoor Run</td>
<td>Wed, Aug 11</td>
<td>4:39 PM - 5:37 PM</td>
<td></td>
<td></td>
<td></td>
<td>0.84 MI</td>
</tr>
</tbody>
</table>

### Common properties:

**Icon Kind**: Outdoor Run
**Date**: Wed, Aug 11
**Start Time**: 4:39 PM - 5:37 PM
**End Time**: | 4:39 PM - 5:37 PM |
**Calories**: 569 CAL
**Heart Rate**: 97 BPM
**Distance**: 0.84 MI

### Swimming Specific:

**Swimming Pace**: 471 CAL
**Stroke Type**: Mixed (44yd)
**100 yd Splits**: Breaststroke (0.10mi)
**Elevation Gain**: 135 FT
**Avg. Pace**: 8'58"/MI
**Avg. Cadence**: 168 RPM
**Avg. Heart Rate**: 165 BPM
**Elevation**: 194 FT MAX; 89 FT MIN

### Running Specific:

**Running Pace**: 3'52"/41
**Mile Splits**: 186 BPM AVG
**Avg. Cadence**: 187 RPM
**Avg. Pace**: 8'56"/MI
**Heart Rate**: 186 BPM AVG

© Apple. All rights reserved. This content is excluded from our Creative Commons license. For more information, see [https://ocw.mit.edu/help/faq-fair-use/](https://ocw.mit.edu/help/faq-fair-use/)
INHERITANCE:
PARENT CLASS

```python
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

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

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

Parent is Workout
Inherits all attributes of Workout:
- start, end, calories
- get_calories(), get_start(),
- get_end(), __str__()
INHERITANCE REPRESENTATION
IN MEMORY

Workout Class
- super()
- __init__()
- get_start()
- get_end()
- set_start()
- set_end()
- set_calories()
- get_calories()
- cal_per_hr

RunWorkout Class
- get_elev()
- set_elev()

RunWorkout instance
- start
- end
- calories
- icon
- kind
- elev

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
Complex print function shared by all subclasses

```python
class Workout(object):
    ...

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

outputs

```
<table>
<thead>
<tr>
<th>Workout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:22:00</td>
</tr>
<tr>
<td>73 Calories</td>
</tr>
</tbody>
</table>
```

SUBCLASSES REUSE PARENT CODE
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 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

```python
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
```
WHERE CAN I USE AN INSTANCE OF A CLASS?

```python
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')
run1 = RunWorkout('9/30/2021 1:35 PM','9/30/2021 3:35 PM', 100)
run2 = RunWorkout('9/30/2021 1:35 PM','9/30/2021 3:35 PM', 200)
total_calories([w1, w2, run1, run2])  # (1) cal = 100+100+400+400
total_elevation([run1, run2])  # (2) elev = 100+200
total_elevation([w1, run1])  # (3) err! w1 has no elev method
```

27 min workouts = 100 cal
2 hr run workouts
30 min workouts = 100 cal
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()`

```python
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)
```
**OVERRIDING SUPERCLASSES**

- Overriding superclass – add calorie calculation w/ distance

```python
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
- get_calories() overridden since it is defined in both sub and superclass
- route_gps_points contains lat/lon pairs of route run
- 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

Workout Class
- super()
- get_calories()
- get_start()
- get_end()
- set_calories()
- set_start()
- set_end()
- __init__()
- cal_per_hr

RunWorkout Class
- get_elev()
- set_elev()
- get_calories()
- cals_per_km

RunWorkout instance
- start
- end
- calories
- icon
- kind
- elev

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
TESTING EQUALITY WITH SUBCLASSES

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

```python
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()

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