Object Oriented Programming (OOP)

Intro2CS – week 7b
What is Object Oriented Programming?

- A programing paradigm that is based on **data**, and **code** that are coupled together.

- Objects store information and carry the code that accesses and manipulates information.

Our first example will be the Car **class**.

- The class Car is the type.
- Specific cars of that class are objects (**instances**).
How “Car” will be used

Things to notice:
• When we “use” the car, it changes its internal state (mileage increases, gas is consumed)
• Cars “remember” things about their current state,
• have functions (methods) that can change that state.
• Each car is separate.

```python
car1 = Car()
car2 = Car()
car1.add_fuel(15)
car1.drive(500)
car2.drive(100)
print("car 1 fuel: ", car1.get_fuel(), " millage: ", car1.get_milage())
print(car2)
```
Class vs Object

```python
class Car:
    pass

car1 = Car()
car2 = Car()
```

Defines a new class named Car. Class names should have capital letters in the beginning of each word. Examples: Car, BigNumber, ChessBoard

Additional statements will come here.

Create two instances (objects) of type Car. Notice the parentheses: ()
To check if an object is of a certain instance:

```python
isinstance(car1, Car)
```

This is usually not good code.
The class "Car":
<class '__main__.Car'>

Add a field to each object noting the amount of fuel in the car. Each object has its own value.
These are called by many names: member variables, member fields, attributes
A class variable. Shared by all objects.

```python
class Car:
    pass

car1 = Car()
car2 = Car()

print(type(car1))
car1.fuel = 3
car2.fuel = 5
Car.NUM_WHEELS = 4

print("car1.fuel:", car1.fuel)
print("car2.fuel:", car2.fuel)

print("car1.NUM_WHEELS:", car1.NUM_WHEELS)
print("car2.NUM_WHEELS:", car2.NUM_WHEELS)
```

Output:

```
<class '__main__.Car'>
car1.fuel: 3
car2.fuel: 5
car1.NUM_WHEELS: 4
car2.NUM_WHEELS: 4
```
Initializing objects

```python
class Car:
    def __init__(self, fuel_amount):
        self.fuel = fuel_amount

NUM_WHEELS = 4

car1 = Car(3)
car2 = Car(5)

print("car1.fuel:", car1.fuel)
print("car2.fuel:", car2.fuel)

print("car1.NUM_WHEELS:", car1.NUM_WHEELS)
print("car2.NUM_WHEELS:", car2.NUM_WHEELS)
print("Car.NUM_WHEELS:", Car.NUM_WHEELS)
```

Output:
```
car1.fuel: 3
car2.fuel: 5
car1.NUM_WHEELS: 4
car2.NUM_WHEELS: 4
Car.NUM_WHEELS: 4
```
Initializing objects

```python
class Car:
    def __init__(self, fuel_amount):
        self.fuel = fuel_amount

NUM_WHEELS = 4

car1 = Car(3)
car2 = Car(5)

print("car1.fuel:", car1.fuel)
print("car2.fuel:", car2.fuel)

print("car1.NUM_WHEELS:", car1.NUM_WHEELS)
print("car2.NUM_WHEELS:", car2.NUM_WHEELS)
print("Car.NUM_WHEELS:", Car.NUM_WHEELS)
```

Output:
```
car1.fuel: 3
car2.fuel: 5
car1.NUM_WHEELS: 4
car2.NUM_WHEELS: 4
Car.NUM_WHEELS: 4
```
Objects Have State

• The data inside objects represents their state

• The state is changed by performing actions on the object: Invoking “methods”
  (methods are functions that run and may change its state)
Methods

class Car:
    KM_PER_LITRE = 16

    def __init__(self, fuel_amount):
        self.fuel = fuel_amount
        self.milage = 0

    def add_fuel(self, amount):
        self.fuel += amount

    def drive(self, distance):
        self.milage += distance
        self.fuel -= distance / Car.KM_PER_LITRE

In the code above, functions within the class definition are called methods, or member functions. They add functionality to the objects.

The first parameter is usually called self. It's the object they run on.

Here is how they are invoked.

car1 = Car(3)
car1.add_fuel(7)
car1.drive(10)
print(car1.fuel)
These two calls are equivalent

car1.add_fuel(15)
Car.add_fuel(car1, 15)
Hiding the internal representation

• Good object oriented programming helps objects behave in a consistent manner

• Example: No way to increase the mileage of the car without driving it (and burning fuel)

• If the programmer directly changes car1.mileage=-2 he can make mistakes.

• It is therefore good practice to write objects that hide internal data and only change as a result of their own method calls.
Hiding the internal representation

class Car:
    KM_PER_LITRE = 16

    def __init__(self, fuel_amount):
        self.__fuel = fuel_amount
        self.__milage = 0

    def add_fuel(self, amount):
        self.__fuel += amount

    def drive(self, distance):
        self.__milage += distance
        self.__fuel -= distance / Car.KM_PER_LITRE

    def get_fuel(self):
        return self.__fuel

    def print_fuel(self):
        print(self.__fuel)

car1 = Car(30)
print(car1.get_fuel())
print(car1.__fuel)

Private members are denoted with a double underscore.

access data only through methods.

Will not work. Variable is still there, but has actually been renamed to _Car__fuel
Hiding the Internals

• If the class is written properly, it will be impossible to change the internal state to something inconsistent.

• A car with negative fuel?
• A car with negative milage?
Ensuring internal consistency

class Car:
    KM_PER_LITRE = 16

    def __init__(self, fuel_amount):
        self.__fuel = fuel_amount
        self.__milage = 0

    def add_fuel(self, amount):
        if amount > 0:
            self.__fuel += amount

    def drive(self, distance):
        if distance > 0:
            driving_distance = min(distance, self.__fuel * self.KM_PER_LITRE)
            self.__milage += driving_distance
            self.__fuel -= driving_distance / self.KM_PER_LITRE

    def get_fuel(self):
        return self.__fuel

Similarly with get_milage
Application Interfaces (APIs) and Design by Contract

• The methods of the class define its API – how programmers interact with its instances

• The “design by contract” paradigm:
  – APIs make promises about the behavior of objects (but without committing to internal representation).
  – Encapsulation: Programmer needs only the interface, doesn’t care about internals.
All Things are Objects

```python
x = 30
print(x.real)
print(x.bit_length())

lst = ['a', 'b', 'c', 'a']
print(lst.count('a'))
lst.remove('b')
lst.insert(1, 'h')
lst.sort()
print(lst)
```

```
30
5
2
['a', 'a', 'c', 'h']
```
Polymorphism

Actions can sometimes be applied to objects of many different types. This is very natural in Python.

```python
class Cat:
    def make_sound(self):
        print("Meow!")

class Dog:
    def make_sound(self):
        print("Woof Woof!")

some_animals = [Cat(), Dog(), Dog()]
for animal in some_animals:
    animal.make_sound()
```
Polymorphism

Actions can sometimes be applied to objects of many different types. This is very natural in Python.

```python
class Cat:
    def make_sound(self):
        print("Meow!")

class Dog:
    def make_sound(self):
        print("Woof Woof!")

some_animals = [Cat(), Dog(), Dog()]
for animal in some_animals:
    animal.make_sound()
```

(There are actually better ways to do this in OOP using inheritance)
Documenting classes

• Write docstrings for each method in the class.
  – Write what the API promises regarding each method

• Write a docstring at the beginning of the class
  – Here you should also document attributes, but only the public ones

```python
class Cat:
    """ this class represents a kitty cat

    name: the name of the cat

    """
    def __init__(self, name):
        """
        creates a cat with the given name
        """
        self.name = name
```
Objects are Mutable

def my_func(some_car):
    some_car.add_fuel(20)

my_car = Car(10)
print(my_car.get_fuel())
my_func(my_car)
print(my_car.get_fuel())

No “return” of car object. What happens?
class Point2D:
    def __init__(self, x=0, y=0):
        self.__x = x
        self.__y = y

    def translate(self, dx, dy):
        self.__x += dx
        self.__y += dy

    def distance_from(self, other):
        dx = self.__x - other.__x
        dy = self.__y - other.__y
        return (dx**2 + dy**2)**(1/2)

    def copy(self):
        return Point2D(self.__x, self.__y)

    def get_x(self):
        return self.__x

    def get_y(self):
        return self.__y
class Rectangle2D:
    def __init__(self, corner, width, height):
        self.__corner = corner.copy()
        self.__width = width
        self.__height = height

    def get_area(self):
        return self.__width*self.__height

    def translate(self, dx, dy):
        self.__corner.translate(dx, dy)

    def get_midpoint(self):
        return Point2D(self.__corner.get_x()+self.__width/2,
                       self.__corner.get_y()+self.__height/2)

    def copy(self):
        return Rectangle2D(self.__corner.copy(),
                            self.__width, self.__height)
Now we can work directly with the point and rectangle classes.

```python
pt = Point2D(0,0)
print(Point2D(1,3).distance_from(pt))
rect = Rectangle2D(Point2D(1,1), 3, 4)
rect.translate(2, 1)
print(rect.get_area())
```
Adding more behavior to objects

```python
pt1 = Point2D(0,0)
print(pt1)

<__main__.Point2D object at 0x027C4BD0>
```

We’d like something nicer to be printed
The `__str__` method is called when a conversion to a string is needed.

```python
class Point2D:
    def __init__(self, x=0, y=0):
        self.__x = x
        self.__y = y

    def __str__(self):
        return "("+str(self.__x)+ " \n"+"","+str(self.__y)+")"

pt1 = Point2D(0,0)
print(pt1)
print(str(pt1))
```

```
(0,0)
(0,0)
```
Defining how operators behave

pt1 = Point2D(0,0)
pt2 = Point2D(0,0)
print(pt1==pt2)  ➔ False

The default behavior of == on objects it to check if the are the exact same object, ignoring data members.
• To redefine how == behaves, simply implement the method `__eq__()` within the class.

```python
def __eq__(self, other):
    return self.__x == other.__x and \
    self.__y == other.__y
```
Other operators

- We can define other **magic methods** in python that make other operators work with our code

<table>
<thead>
<tr>
<th>Operator</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td><code>__add__</code></td>
</tr>
<tr>
<td>-</td>
<td><code>__sub__</code></td>
</tr>
<tr>
<td>*</td>
<td><code>__mul__</code></td>
</tr>
<tr>
<td>//</td>
<td><code>__floordiv__</code></td>
</tr>
<tr>
<td>/</td>
<td><code>__div__</code></td>
</tr>
<tr>
<td>%</td>
<td><code>__mod__</code></td>
</tr>
<tr>
<td>**</td>
<td><code>__pow__</code></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td><code>__lshift__</code></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td><code>__rshift__</code></td>
</tr>
<tr>
<td>&amp;</td>
<td><code>__and__</code></td>
</tr>
<tr>
<td>^</td>
<td><code>__xor__</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td><code>__lt__</code></td>
</tr>
<tr>
<td>&lt;=</td>
<td><code>__le__</code></td>
</tr>
<tr>
<td>==</td>
<td><code>__eq__</code></td>
</tr>
<tr>
<td>!=</td>
<td><code>__ne__</code></td>
</tr>
<tr>
<td>&gt;=</td>
<td><code>__ge__</code></td>
</tr>
<tr>
<td>&gt;</td>
<td><code>__gt__</code></td>
</tr>
</tbody>
</table>

There are more...
An example

• Python has a set object (a container that doesn’t allow items to be included twice)
• But what if it didn’t?
• How would we implement one?
class MySet:
    def __init__(self):
        self.__items=[]

    def add_item(self,item):
        if item not in self.__items:
            self.__items.append(item)

    def remove_item(self,item):
        if item in self.__items:
            self.__items.remove(item)

    def add_set(self,other):
        for item in other.__items:
            self.add_item(item)

This implementation is inefficient. (Why?)
How can we make it better using things we’ve seen?
def __contains__(self, item):
    return item in self.__items

def __eq__(self, other):
    if len(self.__items) == len(other.__items):
        result = True
        for item in self.__items:
            result = result and item in other.__items
        return result
    else:
        return False

def __add__(self, other):
    result = MySet()
    result.__items = self.__items[:]
    result.add_set(other)
    return result

def __str__(self):
    return "{"+",".join([str(s) for s in self.__items])}"

def __len__(self):
    return len(self.__items)
set1 = MySet()
set1.add_item(2)
set1.add_item(4)
set2 = MySet()
set2.add_item(4)
set2.add_item(3)
set2.add_item(5)
set1.remove_item(4)

print(set1)  # {2}
print(set2)  # {4,3,5}
print(set1+set2)  # {2,4,3,5}
print(2 in set1)  # True
print(20 in set1)  # False
print(len(set1))  # 1
Another example

class Knapsack:
    def __init__(self, max_weight, max_volume):
        self.__max_volume = max_volume
        self.__max_weight = max_weight
        self.__weight = 0
        self.__volume = 0
        self.__items = set()

    def get_total_weight(self):
        return self.__weight

    def get_total_volume(self):
        return self.__volume
def add_item(self, item):
    weight = item.get_weight()
    volume = item.get_volume()
    if self.__weight + weight <= self.__max_weight and 
        self.__volume + volume <= self.__max_volume and 
        item not in self.__items:
        self.__volume += volume
        self.__weight += weight
        self.__items.add(item)
        return True
    else:
        return False

What assumptions are we making about items we add in?
```python
def clear(self):
    self.__weight = 0
    self.__volume = 0
    self.__items = set()

def remove_random_item(self):
    if len(self.__items) == 0:
        return None
    item = random.sample(self.__items, 1)[0]
    self.__items.remove(item)
    self.__weight -= item.get_weight()
    self.__volume -= item.get_volume()
    return item
```
```python
k = Knapsack(55, 10)
print(k.add_item(Book()))
print(k.get_total_volume())
print(k.get_total_weight())
print(k.add_item(Brick()))
print(k.add_item(Brick()))
print(k.add_item(Brick()))
print(k.remove_random_item())
print(k.remove_random_item())
print(k.remove_random_item())
```

```
class Book:
    def get_weight(self): return 5
    def get_volume(self): return 1

class Brick:
    def get_weight(self): return 50
    def get_volume(self): return 1
```

```
True
1
5
True
False
<__main__.Brick object at 0x0000000002B730F0>
<__main__.Book object at 0x0000000002B6B588>
None
```