Intro to Computer Science
CS-UH 1001, Spring 2022

Lecture 14 - Object Oriented Programming III

Recap

- The constructor can have multiple arguments (with default values)
  - `def __init__(self, color, face='Head'):`
- Objects can be appended to lists
  - `coin_list.append(Coin())`
- The `__str__()` is a special method that can be overwritten to get a string representation of an object
  - `def __str__(self):`
    `return "Side up is: " + self.__side_up`
- Operators can also be overloaded, for example:
  - `def __add__(self, target)`

Rectangle class (ex_13.1.py)

Now that we have created a point class, let's create a rectangle class. A rectangle is represented by its upper-left corner point and its size (width and height).

- Create the initializer method
  - Use the Point class to store the corner point (upper left) of the rectangle
  - Use two attributes to represent the width and the height
- Create the __str__ method (should display ((x,y), width, height))
- Create a method to resize the rectangle by a delta width and height
- Create a method that moves the rectangle coordinates (origin) by a delta x and delta y

Now, test the following:

```
r1 = Rectangle(10, 5, 100, 50)
print(r1)
r1.resize(25, -10)
print(r1)
r1.move(-10,10)
print(r1)
```

Copying Mutable Objects

- Remember, using the assignment operator to copy mutable objects only copies the reference of the object

```
p1 = Point(2, 4)
p2 = p1
print(p1 == p2)
```

 True

- Copying mutable objects can be done through the copy module using the copy() function:

```
import copy
p1 = Point(2, 4)
p2 = copy.copy(p1)
print(p1 == p2)
```

 False

This type of copy is called a *Shallow copy*

Shallow Copy

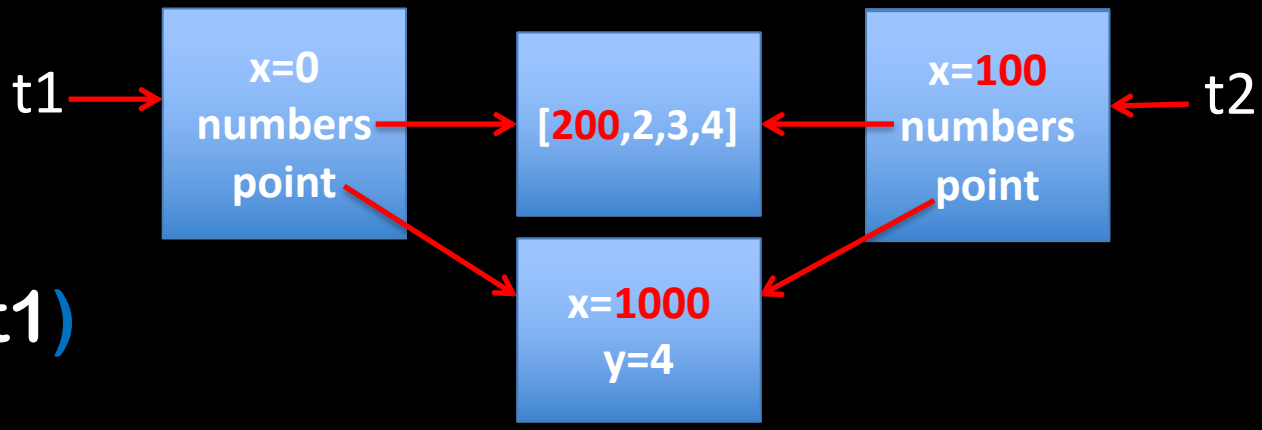
- A Shallow copy creates a new object in a new memory location
  - It copies all immutable attributes
  - But it does not copy values of embedded objects (objects with references, i.e. mutable objects)

- If an object contains attributes of:
  - Lists
  - Dictionaries
  - Other objects, except int, float, boolean, string

  Only the reference of the attributes will be copied!

```
import copy

class Test:
    def __init__(self):
        self.x = 0
        self.numbers = [1, 2, 3, 4]
        self.point = Point(2, 4)
```



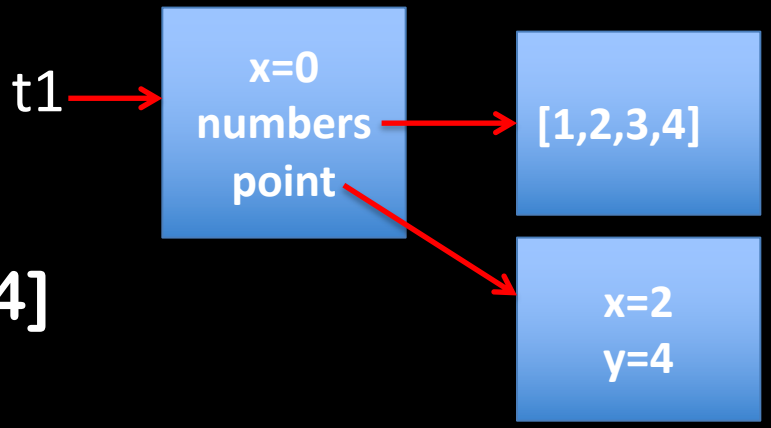

```
t1 = Test()
t2 = copy.copy(t1)
t2.x = 100
t2.numbers[0] = 200
t2.point.x = 1000
```

Deep Copy

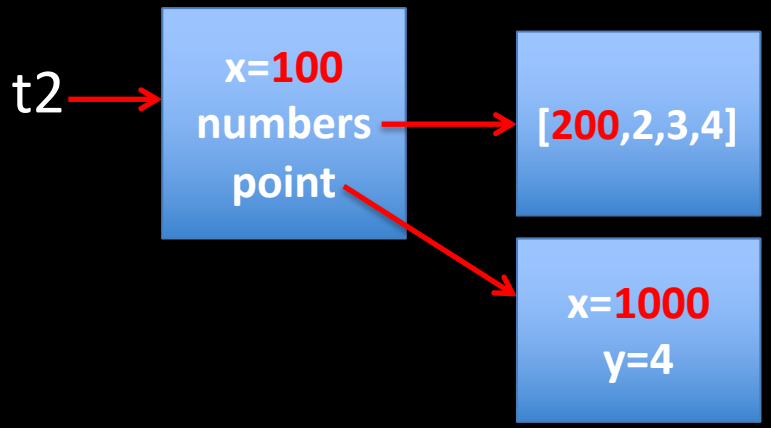
- The copy module has another copy function called deepcopy()
- deepcopy() copies recursively:
  - the content of an object
  - any embedded objects
  - all objects in embedded objects
  - And so on
- Result: a new/separate copy of the entire object is created, including all embedded objects

```
import copy

class Test:
    def __init__(self):
        self.x = 0
        self.numbers = [1, 2, 3, 4]
        self.point = Point(2, 4)
```




```
t1 = Test()
t2 = copy.deepcopy(t1)
t2.x = 100
t2.numbers[0]=200
t2.point.x = 1000
```



Breakout session I:
Card Class

Card and Deck class (ex_14.1.py)

1. Create a card class in OOP

   A card has the following attributes and method:

   - A suit (Spades, Hearts, Diamonds, Clubs)
   - A rank (Ace, 2, 3, 4, …., 10, Jack, Queen, King)
   - A value (11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10)
   - A print method

Use the template ex_14.1.py from Brighspace

2. Create a deck class

   A deck is a collection of all possible combinations of cards (52 cards). A deck has one attribute to hold the 52 cards and the following methods:

   - Printing the deck
   - Shuffling the deck (use random.shuffle(list))
   - Dealing a card

Inheritance & Polymorphism

Inheritance

- Inheritance is a relationship between a more general class (superclass or base class) and a more specialized class (subclass)

- The subclass inherits all attributes and methods from the superclass

Inheritance

- Why might this be useful?
  - You might have a general class that is defined and has attributes and methods
  - You want to use it but you need extra attributes or methods
  - Or you might want to change some of the existing methods

Generic Example

- Think about a Vehicle class. There can be three different vehicle types:
  - Car
  - Motorcycle
  - Truck
- Regardless of the type, there are certain attributes that they all share:
  - Make
  - Model
  - Mileage
  - Price

```
class Vehicle:

    def __init__(self, make, model, mileage, price):
        self.__make = make
        self.__model = model
        self.__mileage = mileage
        self.__price = price

    def set_make(self, make):
        self.__make = make

    def get_make(self):
        return self.__make

    def set_model(self, model):
        self.__model = model

    def get_model(self):
        return self.__model

    def set_mileage(self, mileage):
        self.__mileage = mileage

    def get_mileage(self):
        return self.__mileage

    def set_price(self, price):
        self.__price = price

    def get_price(self):
        return self.__price
```

Generic Example

- However, there are a number of attributes that are type specific:
  - Cars and Trucks can have an attribute on number of doors
  - Motorcycle only has 2 tires and no doors
  - Trucks have loading capacity
- From the generic Vehicle class, we can create three sub-classes for each type

```
class Car(Vehicle):

    def __init__(self, make, model, mileage, price, doors):
        super().__init__(make, model, mileage, price)
        self.__doors = doors

    def set_doors(self, doors):
        self.__doors = doors

    def get_doors(self):
        return self.__doors

my_car = Car('BMW', 2010, 70000, 10000, 4)
print('Make:', my_car.get_make())
print('Model:', my_car.get_model())
print('Mileage:', my_car.get_mileage())
print('Price:', my_car.get_price())
print('Number of doors:', my_car.get_doors())
```

Defining a Car class that inherits from the Vehicle class

Car inherits all of Vehicle attributes and methods

This calls the constructor of the Vehicle class to inherit (and initialize) all attributes (optional)

Additional attribute of the Car class

Additional methods of the Car class

Methods from the superclass

Method from the subclass

```
class SuperCar(Car):
```

Defining a SuperCar class that inherits from the Car and the Vehicle class

```
    def __init__(self, make, model, mileage, price, doors, hp):
        super().__init__(make, model, mileage, price, doors)
        self.__hp = hp

    def set_hp(self, hp):
        self.__hp = hp

    def get_hp(self):
        return self.__hp

my_super_car = SuperCar('Ferrari', 2020, 1000, 250000, 2, 800)
print('Make:', my_super_car.get_make())
print('Model:', my_super_car.get_model())
print('Mileage:', my_super_car.get_mileage())
print('Price:', my_super_car.get_price())
print('Number of doors:', my_super_car.get_doors())
print('Horsepower:', my_super_car.get_hp())
```

Card and Deck class (ex_14.1.py)

1. Create a card class in OOP

   A card has the following attributes and methods:

   - A suit (Spades, Hearts, Diamonds, Clubs)
   - A rank (Ace, 2, 3, 4, …., 10, Jack, Queen, King)
   - A value (11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10)
   - A print method

2. Create a deck class

   A deck is a collection of all possible combinations of cards (52 cards). A deck has one attribute to hold the 52 cards and the following methods:

   - Printing the deck
   - Shuffling the deck (use random.shuffle(list))
   - Dealing a card

Think about the Deck class and how inheritance could be applied to it.

Inheritance in Python

```
class A
class B(A)
class C(B)
```

Polymorphism

- Polymorphism allows a subclass to redefine a superclass method

- Let's look at another example:
  - Assume we are creating an animal class
  - One generic type of animals is Animal
  - A dog is an Animal
  - A cat is also an Animal

Polymorphism

- The Animal class has:
  - An attribute representing the species type
  - A method that prints the species type
  - A method that prints the sound the animal makes
- Now, let's create the Animal Class and then:
  - Create a Dog subclass
  - Create a Cat subclass

```
class Animal:
    def __init__(self, species='Unknown'):
        self.__species = species

    def show_species(self):
        print('I am a', self.__species)

    def make_sound(self):
        print('Grrrrrrr')

class Dog(Animal):
    def __init__(self):
        super().__init__('dog')

class Cat(Animal):
    def __init__(self):
        super().__init__('cat')
```

```
class Animal:
    def __init__(self, species='Unknown'):
        self.__species = species

    def show_species(self):
        print('I am a', self.__species)

    def make_sound(self):
        print('Grrrrrrr')

class Dog(Animal):
    def __init__(self):
        super().__init__('dog')

class Cat(Animal):
    def __init__(self):
        super().__init__('cat')

dog = Dog()
cat = Cat()

dog.show_species()      I am a dog
dog.make_sound()        Grrrrrrr

cat.show_species()      I am a cat
cat.make_sound()        Grrrrrrr
```

This calls the constructor of the Animal class to inherit all attributes

Notice how the sound of the animals are not correct
A dog sound should be Wof Wof
A cat sound should be Meow

The make_sound() method is defined in the Animal class
The dog and cat subclasses both inherit this method from the Animal class

```
class Animal:
    def __init__(self, species='Unknown'):
        self.__species = species

    def show_species(self):
        print('I am a', self.__species)

    def make_sound(self):
        print('Grrrrrrr')

class Dog(Animal):
    def __init__(self):
        super().__init__('dog')

    def make_sound(self):
        print('Wof Wof')

class Cat(Animal):
    def __init__(self):
        super().__init__('cat')

    def make_sound(self):
        print('Meow')
```

Overload the make_sound() method of the superclass

```
dog = Dog()
cat = Cat()

dog.show_species()    I am a dog
dog.make_sound()      Wof Wof

cat.show_species()    I am a cat
cat.make_sound()      Meow
```

Instance of a Class

- For example, if we define a **function** to print all the information of an Animal

```
def showAnimal(creature):
    creature.show_species()
    creature.make_sound()

dog = Dog()
cat = Cat()

showAnimal(dog)
showAnimal(cat)
showAnimal('Hello World')
```

Expects an object as the argument!

Error

Instance of a Class

```
def showAnimal(creature):
    if isinstance(creature, Animal):
        creature.show_species()
        creature.make_sound()
    else:
        print ('Argument is not an Animal')

showAnimal(dog)
showAnimal(cat)
showAnimal('Hello World')
```

Is not an Animal

Breakout session II:
Inheritance and Polymorphism

Card, Deck and Hand class (ex_14.1.py)

1. Create a card class in OOP

   A card has the following attributes and methods:

   - A suit (Spades, Hearts, Diamonds, Clubs)
   - A rank (Ace, 2, 3, 4, …., 10, Jack, Queen, King)
   - A value (11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10)
   - A print method

2. Create a deck class

   A deck is a collection of all possible combinations of cards (52 cards). A deck has one attribute to hold the 52 cards and the following methods:

   - Printing the deck
   - Shuffling the deck (use random.shuffle(list))
   - Dealing a card

Create an empty Hand class that inherits from the Deck class, create 2 players (hands) and deal 2 cards for each from the Deck class