shape.py
shapes.py
shape_movie.py
fish_tank.py

Lecture 07.03

by Marina Barsky

Reusing objects

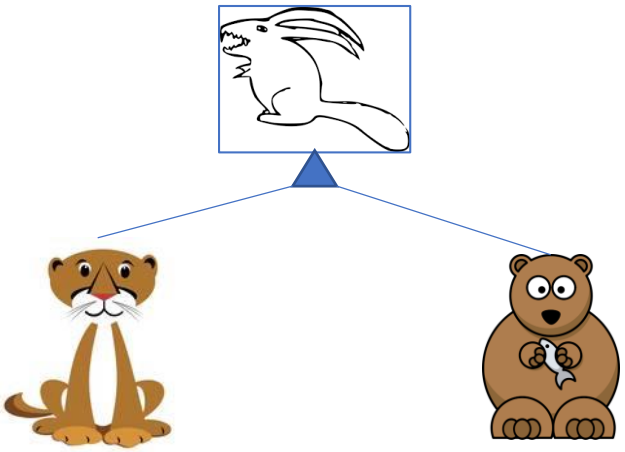
Two main approaches to reusing objects

- Inheritance
- Composition

I. Inheritance

Factoring out similarities

- If we define a set of new types (classes) we often find that there are similarities among them
- For example:
  - Class ***Lion*** and class ***Bear*** – both have a lot in common
  - We can factor out similarities and define them in a single class *Animal*

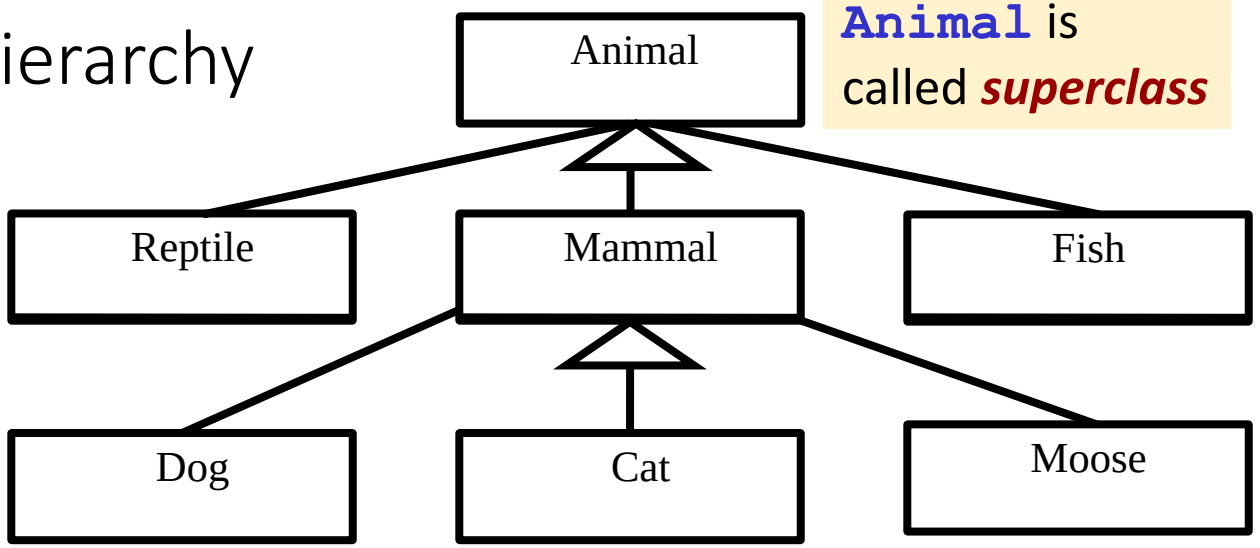


Lion **is a**n Animal Bear **is a**n Animal

Inheritance hierarchy

`Animal` is called ***superclass***

Subclass of class Animal



- Where there's inheritance, there's an *Inheritance Hierarchy* of classes
  - **`Mammal`** "*is an*" **`Animal`**
  - **`Cat`** "*is a*" **`Mammal`**
  - Transitive relationship: a **`Cat`** "*is an*" **`Animal`** too

- We can say:
  - **`Reptile, Mammal`** and **`Fish`** "*inherit from*" **`Animal`**
  - **`Dog, Cat,`** and **`Moose`** "*inherit from*" **`Mammal`**

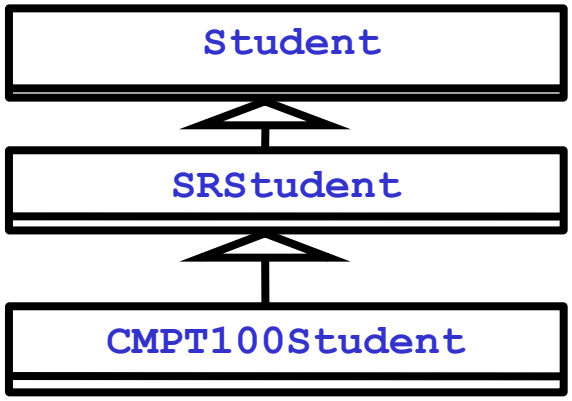
Inheriting properties (fields) and capabilities (methods)

- Subclass *inherits* all capabilities of its superclass
  - if **`Animal`**s eat and sleep, then **`Reptile`**s, **`Mammal`**s, and **`Fish`** eat and sleep
  - if **`Vehicle`**s move, then **`SportsCar`**s move!

- Subclass *specializes* its superclass
  - ***adding*** new fields and methods
  - ***overriding*** *(redefining)* existing methods

- **Superclass** *factors out* capabilities ***common*** among its subclasses
- **Subclasses** are defined by their ***differences*** from their superclass

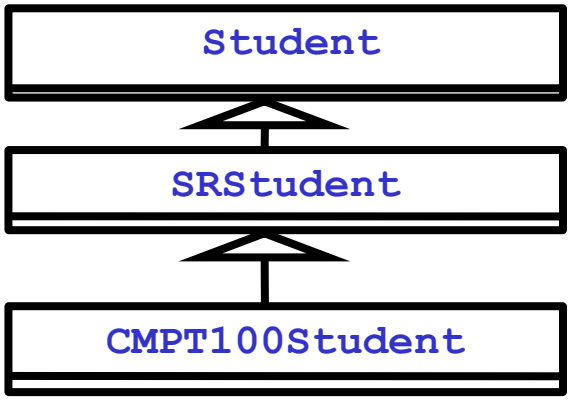
Inheritance Example: 1/3



- **`Student`** *inheritance hierarchy:*
  - **`Student`** is *base class*
  - **`SRStudent`** is **`Student`**'s *subclass*
  - **`CMPT100Student`** is *subclass* of **`SRStudent`**

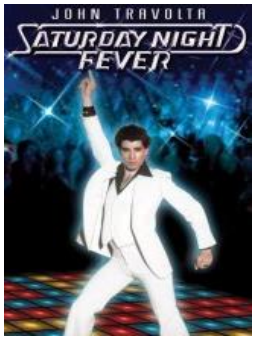
- **`Student`** has *a capability* (or *method*)
  - **`study()`** which works by:
    - going home, opening a book, and reading 50 pages.

Inheritance Example: 2/3



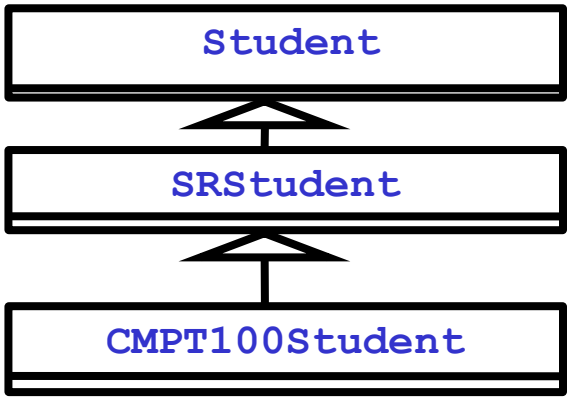
- **SRStudent** "is a" **Student**,
  so it inherits the **study()** method
  - but it *overrides* the method by:
    - reviewing lectures, and doing an assignment
  - note: it **doesn't have** to override this method!

- Finally, the **CMPT100Student** also knows how to **study()** (it **study()**s the same way a **SRStudent** does)
  - however, the **CMPT100Student** subclass adds two capabilities: **gitDown()** and **gitFunky()**




```
def gitDown():
  # Code to party

def gitFunky():
  # Code to do awesome CMPT100 dance
```

Inheritance Example (cont.)



- Each subclass is a *specialization* of its superclass
  - `Student` knows how to `study()`, so all subclasses in hierarchy know how to `study()`
  - but the `SRStudent` does not `study()` the same way a `Student` does
  - and the `CMPT100Student` has some capabilities that neither `Student` nor `SRStudent` have (`gitDown()` and `gitFunky()`)

Inheritance: Classic example

Shape hierarchy

shape.py

shapes.py

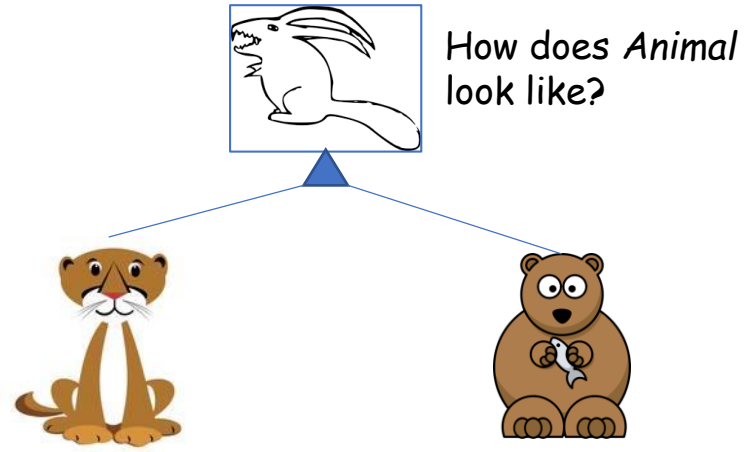
Superclass (Generic class): *Shape*

```
class Shape:

    def __init__(self, x=0, y=0, size=10, color="black"):
        self.x = x
        self.y = y
        self.size = size
        self.color = color
```

`__str__` can be defined once for all shapes

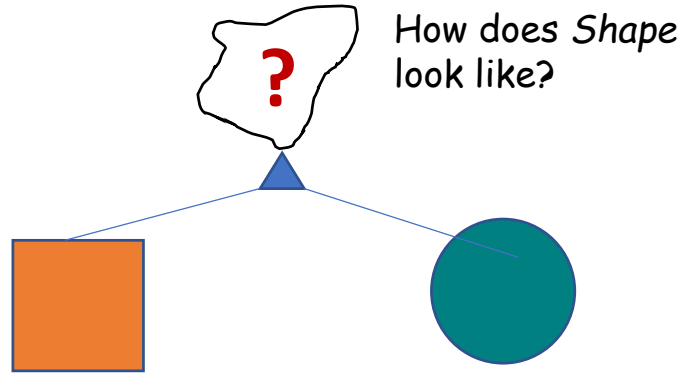
```
class Shape:
    def __str__(self):
        return "{} {} of size {}" \
            " located at ({},{})".format(
                self.color,
                self.__class__.__name__,
                self.size,
                self.x,
                self.y
            )

    def __repr__(self):
        return self.__str__()

    def get_area(self):
        return -1
```

Printing *shapes*




```
if __name__ == '__main__':
    shapes = []
    shapes.append(Shape(0, 0, 15, "red"))
    shapes.append(Shape(100, 50, 40, "green"))
    shapes.append(Shape(200, 20, 50, "blue"))

    print(shapes)
```

[red Shape of size 15 starts at 0,0,
green Shape of size 40 starts at 100,50,
blue Shape of size 50 starts at 200,20]



We can print properties of an abstract *Shape* but we cannot draw it and we cannot find its area

Extending abstract Shape: Square is a Shape

```
class Shape:
    def __init__(self, x=0, y=0, size=10, color="black"):
        self.x = x
        self.y = y
        self.size = size
        self.color = color
```

```
class Square (Shape):
    def get_area(self):
        return self.size ** 2
```

Square inherits all its properties and the constructor from Shape

Extending abstract Shape: Triangle is a Shape

```
class Shape:
    def __init__(self, x=0, y=0, size=10, color="black"):
        self.x = x
        self.y = y
        self.size = size
        self.color = color
```

```
class Triangle(Shape):
    def get_area(self):
        k = (3 ** 0.5)/4
        return k * self.size ** 2
```

Triangle has its own special get_area()

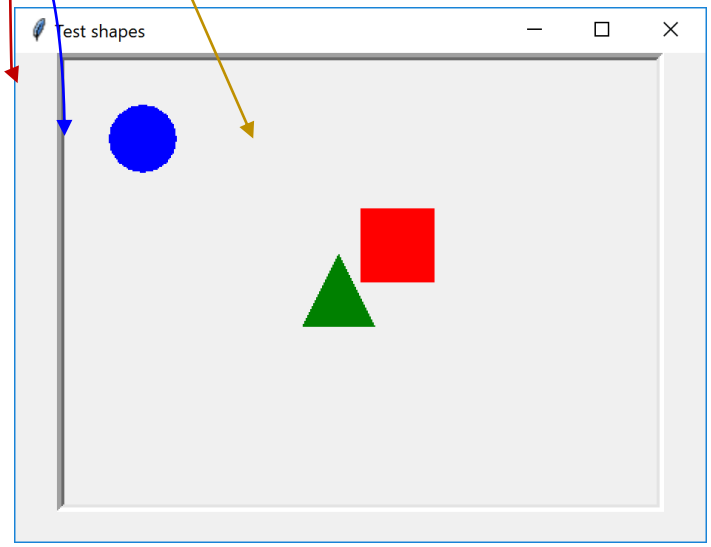
Extending abstract Shape: Circle is a Shape

```
class Shape:
    def __init__(self, x=0, y=0, size=10, color="black"):
        self.x = x
        self.y = y
        self.size = size
        self.color = color
```

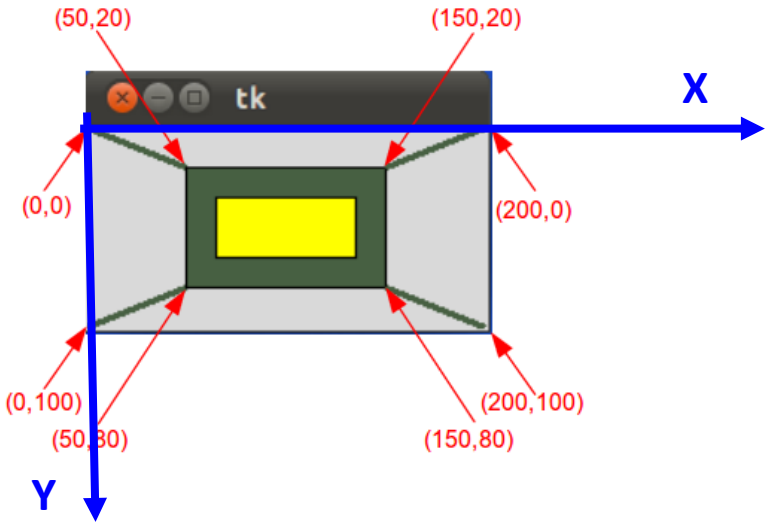
```
class Circle(Shape):
    def get_area(self):
        pi = 3.14
        return pi * self.size ** 2
```

Drawing shapes: tkinter *canvas*

```
window = tk.Tk()
window.title("Test shapes")

frame = tk.Frame(window)
frame.pack()

canvas = tk.Canvas(frame)
canvas.pack()
```



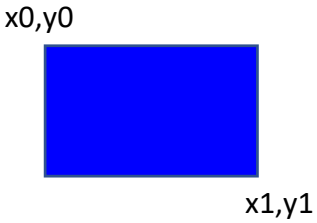
canvas inside *frame* inside *window*



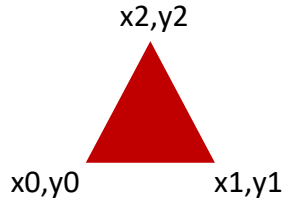
canvas coordinate system

Asking canvas object to hold a shape for us

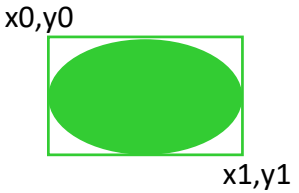
```
class Square (Shape):
    def set_shape(self, canvas):
        self.shape = self.canvas.create_rectangle(
            self.x, self.y,
            self.x + self.size,
            self.y + self.size,
            fill=self.color,
            outline=""
        )
```




```
class Triangle(Shape):
    def set_shape(self, canvas):
        self.shape = self.canvas.create_polygon(
            self.x, self.y,
            self.x + self.size, self.y,
            self.x + self.size/2, self.y - self.size,
            fill=self.color,
            outline=""
        )
```




```
class Circle(Shape):
    def set_shape(self, canvas):
        self.canvas = canvas
        self.shape = self.canvas.create_oval(
            self.x, self.y,
            self.x + self.size, self.y + self.size,
            fill=self.color,
            outline=""
        )
```



Create list of shapes, set it up for drawing

```
shapes = []

o = Circle(30, 30, 45, "blue")
shapes.append(o)

r = Square(200, 100, 50, "red")
shapes.append(r)

t = Triangle(160, 180, 50, "green")
shapes.append(t)

print(shapes)

for i in range(len(shapes)):
    shapes[i].set_shape(canvas)
```

Draw shapes

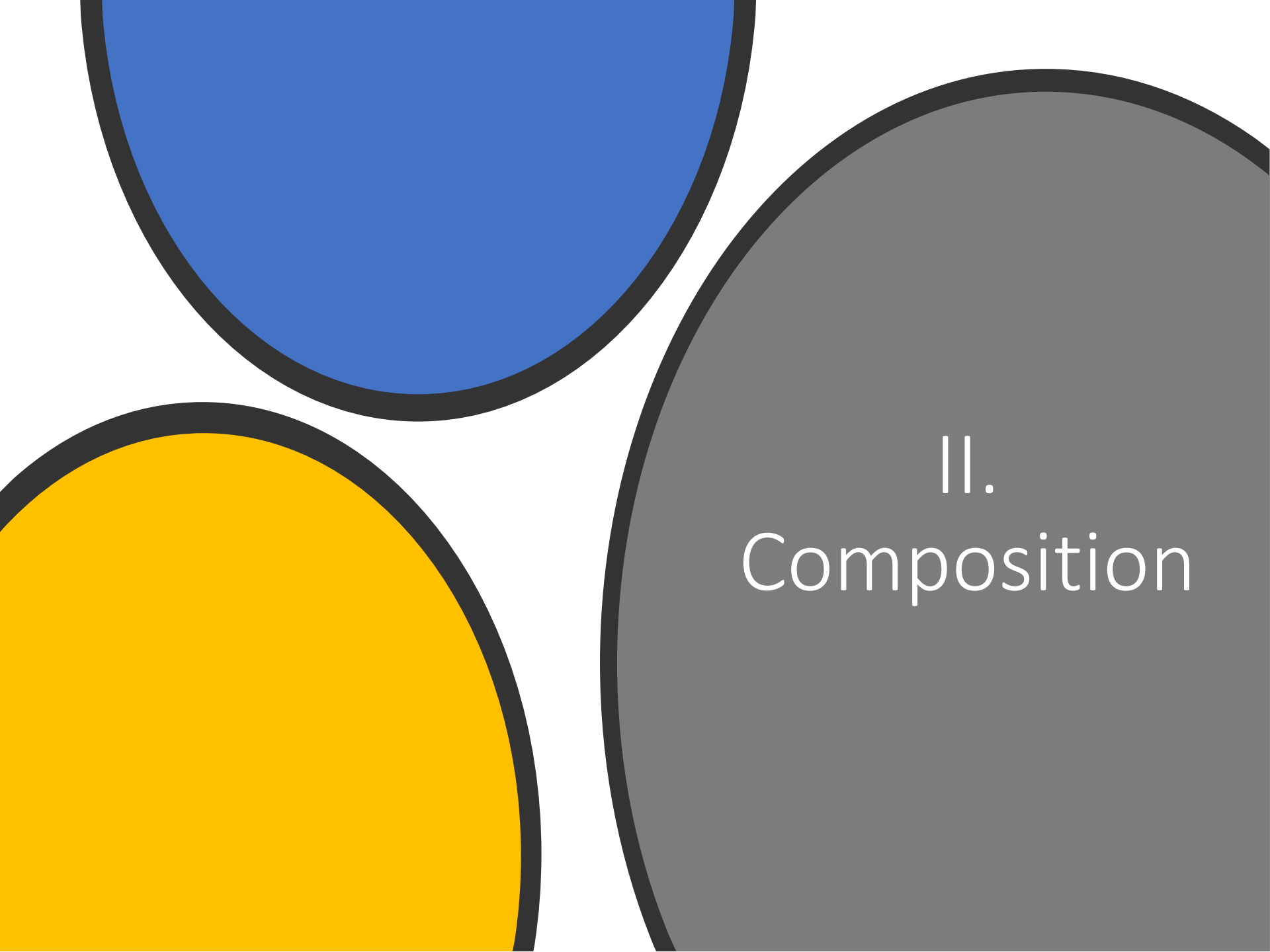
```
window.update()   # fix geometry

for i in range(len(shapes)):
    shapes[i].set_shape(canvas)

try:
    while True:
        window.update_idletasks()   # redraw
        window.update()   # process events
except tk.TclError:
    pass  # to avoid errors when the window is closed
```

This adds a drawing to canvas

This is an ***event loop***: redraws the shapes and listens to events

Why use inheritance

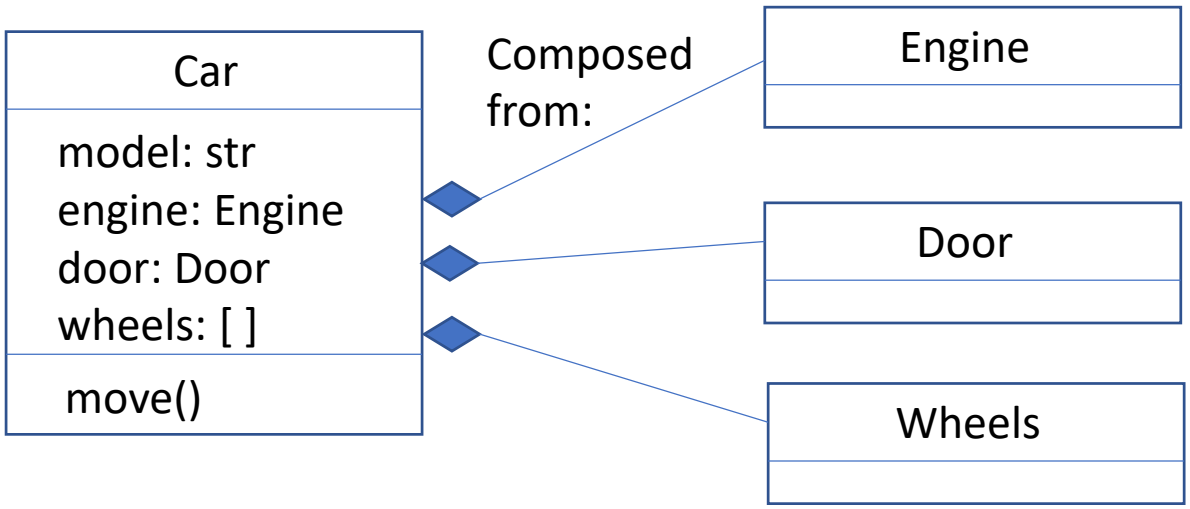
- Get rid of duplicate code by abstracting out the common behavior.
- Modify in one place, and the change is ‘magically’ carried out to all subclasses
- Add new subclasses easily, and they have some methods and properties right away


II.
Composition

Objects as building blocks

- Object fields (attributes) can be of any type: they can also be of a new custom type (class)
- This way we can build complex objects which contain simpler objects within them
- The method of constructing a program by incorporating smaller objects inside a larger one is called ***composition***
- This is **the most useful and widely used** approach in Object-Oriented Programming

Composing with objects



People who build engines do not have to know how to make wheels

- Combining elementary objects to build a more complex object ensures that we can **abstract** only important properties and capabilities of an elementary object, and concentrate on correct implementation of each small piece
- We can divide work among many programmers

Shape Movie class: contains shape objects and moves them on the screen

```
class ShapeMovie:
    def __init__(self, shape_list, canvas):
        self.shapes = shapes
        self.canvas = canvas

    def animate(self):
        canvas_w = int(self.canvas.cget("width"))
        for s in self.shapes:
            if s.shape:
                s.x += (s.direction*s.speed)
                self.canvas.move(s.shape,
                                 s.direction*s.speed, 0)
```

Updates X position of a Shape object

Moves the shape along X axis

shape_movie.py

Shape movie: simple animation

```
class ShapeMovie:
    def __init__(self, shape_list, canvas):
        self.shapes = shapes
        self.canvas = canvas

    def animate(self):
        canvas_w = int(self.canvas.cget("width"))
        for s in self.shapes:
            if s.shape:
                s.x += (s.direction*s.speed)
                self.canvas.move(s.shape,
                                 s.direction*s.speed, 0)

            if s.x < 0 or s.x + s.size > canvas_w:
                s.direction = - s.direction
```

If reached the end of canvas – change direction

shape_movie.py

Movie time!

```
movie = ShapeMovie(shapes, canvas)

window.update()  # fix geometry

try:
    while True:
        movie.animate()
        window.update_idletasks()  # redraw
        window.update()  # process events
except tk.TclError:
    pass  # to avoid errors when the window is closed
```

Moves shapes a little in each iteration of the event loop

fish_tank.py

Class *Fish*: composed of shapes

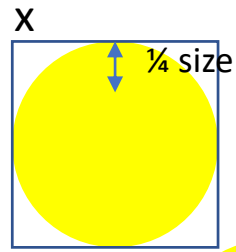
```
class Fish:
    def __init__(self, x, y, size, color,
                 direction=-1, speed=0):
        …

    def set_shape(self, canvas):
        self.canvas = canvas
        self.body = Circle(self.x, self.y, self.size,
                           self.color)
        self.body.set_shape(self.canvas)
```

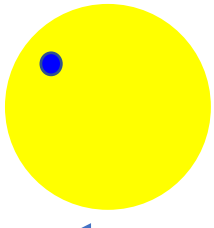
First, the body

Our Fish is round!

Class Fish: eye

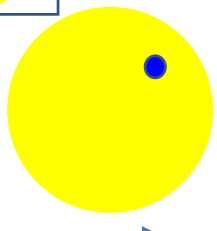
```
class Fish:
    …
    def set_shape(self, canvas):
        self.canvas = canvas
        self.body = Circle(self.x, self.y, self.size,
                           self.color)
        self.body.set_shape(self.canvas)

        self.eye = Circle(self.x + self.size/2
               + self.direction * (self.size / 4),
               self.y + self.size / 4, self.size / 6, "blue")
        self.eye.set_shape(self.canvas)
```





direction = -1

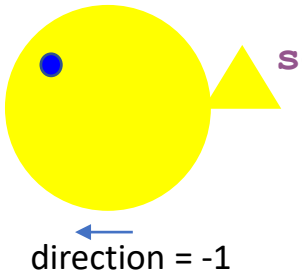
X coordinate of the eye is ¼ of size from the middle



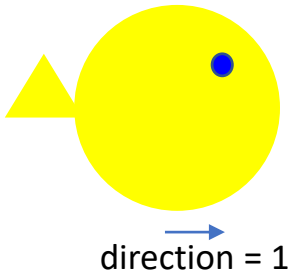
direction = 1

Class Fish: tail

```
class Fish:
    …
    def set_shape(self, canvas):
        …
        tail_x = self.x - self.size / 2
        if self.direction < 0:
            tail_x = self.x - self.direction * self.size
        self.tail = Triangle(tail_x, self.y + self.size / 2,
                             self.size / 2, self.color)
        self.tail.set_shape(self.canvas)
```



direction = -1



direction = 1

Fish tank animation: works exactly as *shape movie*

```
def animate(self):
    canvas_w = int(self.canvas.cget("width"))
    for fish in self.fish_list:
        if fish.body.shape:
            fish.x += (fish.direction * fish.speed)
            self.canvas.move(fish.body.shape,
                             fish.direction * fish.speed, 0)
            self.canvas.move(fish.eye.shape,
                             fish.direction * fish.speed, 0)
            self.canvas.move(fish.tail.shape,
                             fish.direction * fish.speed, 0)
```

Move those body parts

Fish tank animation: change direction of each fish

```
if fish.x < 0 or fish.x + fish.size > canvas_w:
    # reposition tail
    move_x = fish.direction*(fish.size + fish.size/2)
    self.canvas.move(fish.tail.shape, move_x, 0)

    # reposition eye
    self.canvas.move(fish.eye.shape,
                     -fish.direction*fish.size/2, 0)

    fish.direction = -fish.direction
```