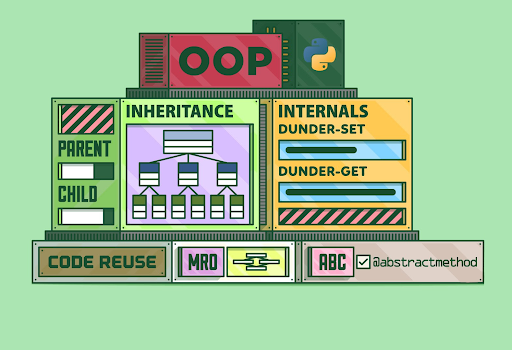
**Session 4**

Object-Oriented Programming Concepts

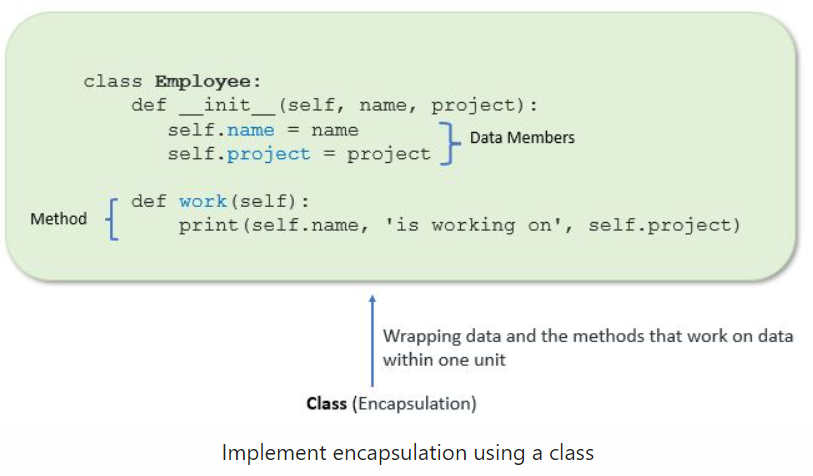
**OOP Concepts: ENCAPSULATION**, **INHERITANCE, AND POLYMORPHISMS**

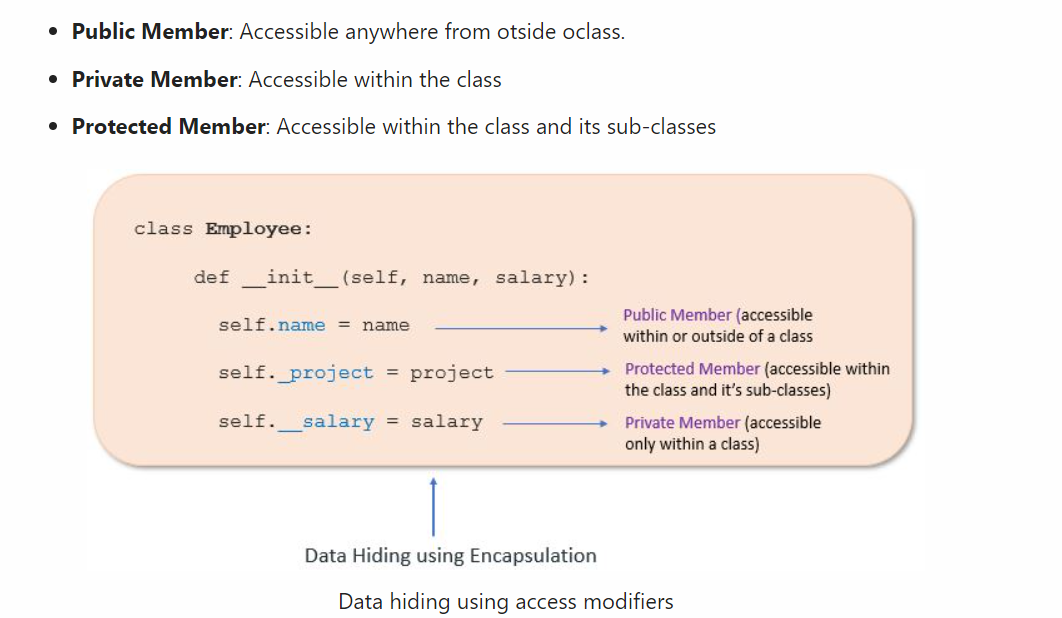
****

**Encapsulation**

What is Encapsulation in Python?

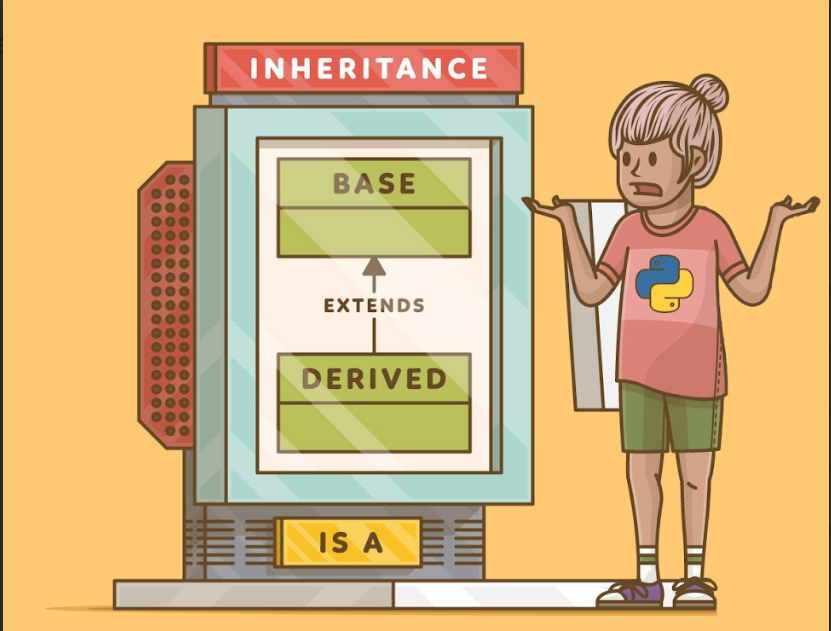
Encapsulation is one of the fundamental concepts in object-oriented programming (OOP), including abstraction, inheritance, and polymorphism. This lesson will cover what encapsulation is and how to implement it in Python.



**Access Modifiers:  
**

## INHERITANCE

* Inheritance allows us to define a class that inherits all the methods and properties from another class.
* The parent class is the class being inherited from, also called the base class.
* The child class is the class that inherits from another class, also called the derived class



Now let's say want to create different types of employees

* manager
* developers

What we can do is create two different types of classes one for managers and another for developers. but that is not what OOP is about. **OOP is about code reuse.**

We break the entire problem into 3 separate classes. One class would have all the attributes common to both the employee and developer classes. and the next 2 classes for managers and developers with the things that are not common

-- This is called subclassing.

-- The subclasses inherit from the base class.

-- So in our case the the **base class** is the **employee class**

-- And the **subclasses** are the **manager and developers class**.

**Super ():**

* Use of super() keyword.
* using super() we can directly pass attributes without passing the self keyword.
* but when using the class name to initialize we must pass the self keyword.

**class** Employee():  
   
 raise\_amount = 1.04  
   
 **def** \_\_init\_\_(self,first,last,pay):  
 self.first\_name = first  
 self.last\_name = last  
 self.salary = pay  
 self.email = first+'.'+last+'@companymail.com'  
   
 **def** full\_name(self):  
 **return** self.first\_name+' '+self.last\_name  
   
 **def** apply\_raise(self):  
 self.salary = int(self.salary \* self.raise\_amount)  
   
**class** Developer(Employee):  
 raise\_amount = 1.1  
   
 **def** \_\_init\_\_(self,first,last,pay,lang):  
 super().\_\_init\_\_(first,last,pay)  
 self.lang = lang  
   
**class** Manager(Employee):  
   
 **def** \_\_init\_\_(self,first,last,pay,designation):  
 super().\_\_init\_\_(first,last,pay)  
 self.designation = designation  
   
  
dev1 = Developer('test1', 'user',3000, 'python')  
dev2 = Developer('test2', 'user',6000, 'java')  
  
mang1 = Manager('mang1', 'super1',30000,' sales')  
  
print(mang1.salary)  
print(dev1.lang)

# Inheritance in PyGame

Let's try to create 2 different types of balls.

1. Fastball which is hollow from the inside
2. Slow ball which is solid

This requires us to rethink the entire structure of the OOP's approach. since both the balls have a few things in common

**Attributes**

1. x and y coordinates
2. radius
3. color

**Methods**

1. move()

#### Task 1:

* **So our approach will be to create a base/parent class that has these attributes and then inherit from this class and create child classes.**
* **This will automatically give us the common attributes and then we can make the necessary modifications/additions to get the desired output.**

*# Creating the Base/Parent circle class*  
**class** Circle():  
 **def** \_\_init\_\_(self):  
 self.x = randint(0,500)  
 self.y = randint(0,500)  
 self.r = randint(10,50)  
 self.color = (randint(0,255),randint(0,255),randint(0,255))  
 self.x\_speed = randint(-2,2)  
 self.y\_speed = randint(-2,2)  
   
 **def** move(self):  
 self.x = self.x+self.x\_speed  
 self.y = self.y+self.y\_speed

Once the base class is created with the attributes common to both the classes we can inherit from this class and create the child classes.

#### Task 2 Creating the FastCircle Class

*# Inheriting from the parent circle class and creating the child class for FasCtCircle*  
**class** FastCircle(Circle):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_()  
   
 *# redefinig the move() method*  
 **def** move(self):  
 self.x = self.x + (self.x\_speed\*2)  
 self.y = self.y + (self.y\_speed\*2)  
   
 *# Adding a new method for drawing circles with thin edges*  
 **def** draw(self):  
 pygame. draw.circle(screen, self. color, (self. x, self. y), self. r,2)

#### Task 3: Creating the SlowCircle Class

*# Inheriting from the parent circle class and creating the child class for SlowCircle*  
**class** SlowCircle(Circle):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_()  
   
 *# There is no need to define the move method as we can use the one from the parent class*  
   
 *# Adding a new method for drawing circles filled with a color*  
 **def** draw(self):  
 pygame. draw.circle(screen, self. color, (self. x, self. y), self. r)

### Putting it all together

import pygame  
from random import randint  
  
pygame.init()  
  
screen = pygame.display.set\_mode([500, 500]) *# creates a screen with the said size*  
  
clock = pygame.time.Clock()  
  
  
*# Creating the Base/Parent circle class*  
**class** Circle():  
 **def** \_\_init\_\_(self):  
 self.x = randint(0,500)  
 self.y = randint(0,500)  
 self.r = randint(10,50)  
 self.color = (randint(0,255),randint(0,255),randint(0,255))  
 self.x\_speed = randint(-2,2)  
 self.y\_speed = randint(-2,2)  
   
 **def** move(self):  
 self.x = self.x+self.x\_speed  
 self.y = self.y+self.y\_speed  
   
   
*# Inheriting from the parent circle class and creating the child class for FasCtCircle*  
**class** FastCircle(Circle):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_()  
   
 *# redefinig the move() method*  
 **def** move(self):  
 self.x = self.x + (self.x\_speed\*2)  
 self.y = self.y + (self.y\_speed\*2)  
   
 *# drawing circles with thin edges*  
 **def** draw(self):  
 pygame. draw.circle(screen, self. color, (self. x, self. y), self. r,2)  
   
   
*# Inheriting from the parent circle class and creating the child class for SlowCircle*  
**class** SlowCircle(Circle):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_()  
   
 *# There is no need to define the move method as we can use the one from the parent class*  
   
 *# drawing circles with thin edges*  
 **def** draw(self):  
 pygame. draw.circle(screen, self. color, (self. x, self. y), self. r)  
  
   
   
*# Creating circle objects using the FastCircle class*  
c1 = FastCircle()  
c2 = FastCircle()  
c3 = FastCircle()  
  
*# Creating circle objects using the SlowCircle class*  
c4 = SlowCircle()  
c5 = SlowCircle()  
c6 = SlowCircle()  
  
  
run = True  
**while** run:  
 **for** event **in** pygame. event.get():  
 **if** event.type == pygame.QUIT:  
 run = False  
   
 screen.fill((255,255,255))  
   
   
  
 *# Creating individual circle objects using the draw method*  
 *# creating Fastcircles*  
 c1.draw()  
 c2.draw()  
 c3.draw()  
 *# Creating SlowCircles*  
 c4.draw()  
 c5.draw()  
 c6.draw()  
   
 *# Moving the circle objects*  
 *# Moving FastCircles*  
 c1.move()  
 c2.move()  
 c3.move()  
 *# Moving SlowCircles*  
 c4.move()  
 c5.move()  
 c6.move()  
   
   
   
 pygame. display.flip()  
   
 clock.tick(30)  
   
   
pygame.quit()

pygame 2.2.0 (SDL 2.0.22, Python 3.11.1)  
Hello from the pygame community. https://www.pygame.org/contribute.html

->Adding OOPs does help in keeping all the attributes and methods associated with an object in a single place.

-> It also helps us reduce the code size a lot by reusing the code by inheriting.

-> But still we have to call the methods on individual objects which is not so efficient

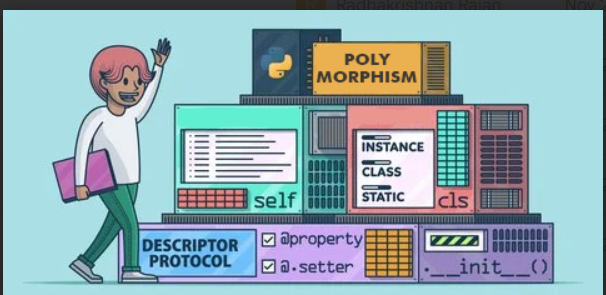
Instead of assigning each object a name, we can directly add the items to the list and call individual items by indexing them.

import pygame  
from random import randint  
  
pygame.init()  
  
screen = pygame.display.set\_mode([500, 500]) *# creates a screen with the said size*  
  
clock = pygame.time.Clock()  
  
  
*# Creating the Base/Parent circle class*  
**class** Circle():  
 **def** \_\_init\_\_(self):  
 self.x = randint(0,500)  
 self.y = randint(0,500)  
 self.r = randint(10,50)  
 self.color = (randint(0,255),randint(0,255),randint(0,255))  
 self.x\_speed = randint(-2,2)  
 self.y\_speed = randint(-2,2)  
   
 **def** move(self):  
 self.x = self.x+self.x\_speed  
 self.y = self.y+self.y\_speed  
   
   
*# Inheriting from the parent circle class and creating the child class for FasCtCircle*  
**class** FastCircle(Circle):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_()  
   
 *# redefinig the move() method*  
 **def** move(self):  
 self.x = self.x + (self.x\_speed\*2)  
 self.y = self.y + (self.y\_speed\*2)  
   
 *# drawing circles with thin edges*  
 **def** draw(self):  
 pygame.draw.circle(screen, self.color, (self.x, self.y), self.r,2)  
   
   
*# Inheriting from the parent circle class and creating the child class for SlowCircle*  
**class** SlowCircle(Circle):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_()  
   
 *# There is no need to define the move method as we can use the one from the parent class*  
   
 *# drawing circles with thin edges*  
 **def** draw(self):  
 pygame.draw.circle(screen, self.color, (self.x, self.y), self.r)  
  
   
   
*# Creating circle objects using the FastCircle class*  
cir =[]  
  
**for** i **in** range(5):  
 cir.append(FastCircle())  
  
**for** i **in** range(5):  
 cir.append(SlowCircle())  
  
run = True  
**while** run:  
 **for** event **in** pygame. event.get():  
 **if** event.type == pygame.QUIT:  
 run = False  
   
 screen.fill((255,255,255))  
   
   
  
 *# Creating individual circle objects using the draw method*  
 *# creating Fastcircles*  
 cir[1].draw()  
*# c2.draw()*  
*# c3.draw()*  
*# # Creating SlowCircles*  
*# c4.draw()*  
*# c5.draw()*  
*# c6.draw()*  
   
*# # Moving the circle objects*  
*# # Moving FastCircles*  
*# c1.move()*  
*# c2.move()*  
*# c3.move()*  
*# # Moving SlowCircles*  
*# c4.move()*  
*# c5.move()*  
*# c6.move()*  
   
   
   
 pygame. display.flip()  
   
 clock.tick(30)  
   
   
pygame.quit()

pygame 2.0.1 (SDL 2.0.14, Python 3.6.8)  
Hello from the pygame community. <https://www.pygame.org/contribute.html>

## POLYMORPHISM

* Polymorphism in Python refers to how different object classes can share the same method name



**class** Dog():  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
   
 **def** speak(self):  
 **return** '{} says woff'.format(self.name)  
**class** Cat():  
 **def** \_\_init\_\_(self,name):  
 self.name = name  
   
 **def** speak(self):  
 **return** '{} says meow'.format(self. name)  
   
dog1 = Dog('Richard)  
cat1 = Cat('isis')  
  
print(dog1.speak())  
print(cat1.speak())

Richard says woff  
isis says meow

## POLYMORPHISM IN PYGAME

As we have objects from 2 different classes and both of them have methods with the same name doing different things we can take advantage of polymorphism to get all the methods called using a single for loop.

import pygame  
from random import randint  
  
pygame.init()  
  
screen = pygame.display.set\_mode([500, 500]) *# creates a screen with the said size*  
  
clock = pygame.time.Clock()  
  
  
*# Creating the Base/Parent circle class*  
**class** Circle():  
 **def** \_\_init\_\_(self):  
 self.x = randint(0,500)  
 self.y = randint(0,500)  
 self.r = randint(10,50)  
 self.color = (randint(0,255),randint(0,255),randint(0,255))  
 self.x\_speed = randint(-2,2)  
 self.y\_speed = randint(-2,2)  
   
 **def** move(self):  
 self.x = self.x+self.x\_speed  
 self.y = self.y+self.y\_speed  
   
   
*# Inheriting from the parent circle class and creating the child class for FasCtCircle*  
**class** FastCircle(Circle):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_()  
   
 *# redefinig the move() method*  
 **def** move(self):  
 self.x = self.x + (self.x\_speed\*2)  
 self.y = self.y + (self.y\_speed\*2)  
   
 *# drawing circles with thin edges*  
 **def** draw(self):  
 pygame.draw.circle(screen, self.color, (self.x, self.y), self.r,2)  
   
   
*# Inheriting from the parent circle class and creating the child class for SlowCircle*  
**class** SlowCircle(Circle):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_()  
   
 *# There is no need to define the move method as we can use the one from the parent class*  
   
 *# drawing circles with thin edges*  
 **def** draw(self):  
 pygame.draw.circle(screen, self.color, (self.x, self.y), self.r)  
  
*# List to store all the circles*   
cir = []  
  
*# Creating FastCircles and adding them to the list*  
**for** i **in** range(5):  
 cir.append(FastCircle())  
  
*# Creating SlowCircles and adding them to the list*  
**for** i **in** range(5):  
 cir.append(SlowCircle())  
  
  
run = True  
**while** run:  
 **for** event **in** pygame. event.get():  
 **if** event.type == pygame.QUIT:  
 run = False  
   
 screen.fill((255,255,255))  
   
   
  
 *# Creating individual circle objects using the draw method*  
 **for** i **in** range(5):  
 cir[i].draw()  
   
 *# Moving the circle objects*  
 **for** i **in** range(5):  
 cir[i].move()  
   
   
   
 pygame. display.flip()  
   
 clock.tick(30)  
   
   
pygame.quit()

pygame 2.0.1 (SDL 2.0.14, Python 3.9.1)  
Hello from the pygame community. https://www.pygame.org/contribute.html

## HOMEWORK

1. Make changes to the final code so that no circle is stationary. *Note: Go through the Random module documentation and try to find a function that will help with the above task.* <https://docs.python.org/3/library/random.html>