Massive Information & Knowledge Engineering

Python OOP

part II

204113 Computer & Programming

Dr. Arnon Rungsawang
Dept. of computer engineering
Kasetsart University

https://mike.cpe.ku.ac.th/204113

Version 2024

Inheritance

Inheritance Concept

- In Object-oriented programming, inheritance is an important aspect. The main purpose of inheritance is the reusability of code because we can use the existing class to create a new class instead of creating it from scratch.
- Process of inheriting the properties of the parent class into a child class is called inheritance.
- The existing class is called a base class or parent class and the new class is called a subclass or child class or derived class.
- In inheritance, the child class acquires all the data members, properties, and functions from the parent class.
- Also, a child class can also provide its specific implementation to the methods of the parent class.

Inheritance Syntax

```
class BaseClass:
  Body_of_base_class
class DerivedClass(BaseClass):
  Body_of_derived_class
```

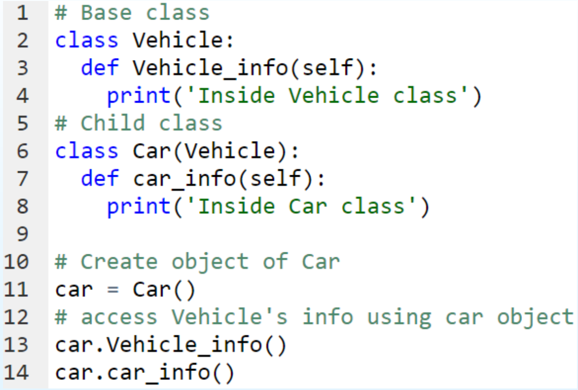
- For example, In the real world, `Car` is a sub-class of a `Vehicle` class.
- We can create a `Car` by inheriting the properties of a `Vehicle` such as `Wheels`, `Colors`, `Fuel_tank`, `engine`, and add extra properties in `Car` as required.

Type of Inheritance

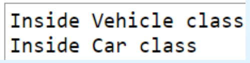
- In Python, based upon the number of child and parent classes involved, there are five types of inheritance.
- The type of inheritance are listed below:
  - Single inheritance
  - Multiple Inheritance
  - Multilevel inheritance
  - Hierarchical Inheritance
  - Hybrid Inheritance

Single Inheritance

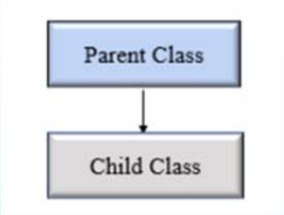
- A child class inherits from a single-parent class. Here is one child class and one parent class.



```python
# Base class
class Vehicle:
  def Vehicle_info(self):
    print('Inside Vehicle class')
# Child class
class Car(Vehicle):
  def car_info(self):
    print('Inside Car class')

# Create object of Car
car = Car()
# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```

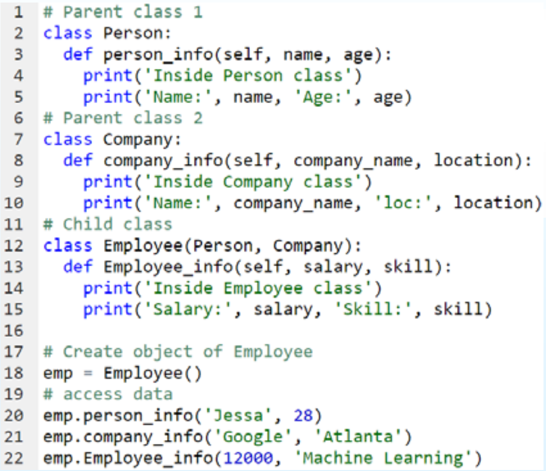


```
Inside Vehicle class
Inside Car class
```

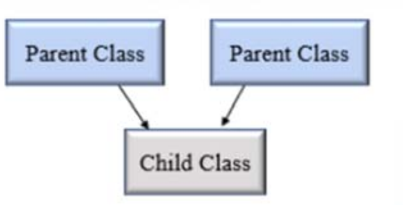


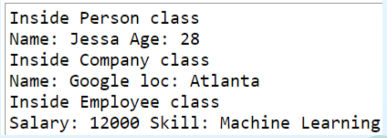
Multiple Inheritance

- One child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.



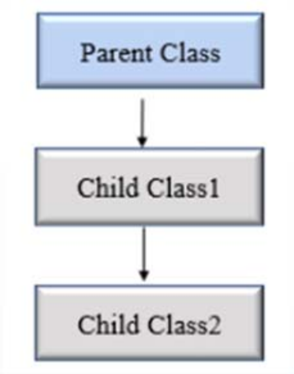
```python
# Parent class 1
class Person:
  def person_info(self, name, age):
    print('Inside Person class')
    print('Name:', name, 'Age:', age)
# Parent class 2
class Company:
  def company_info(self, company_name, location):
    print('Inside Company class')
    print('Name:', company_name, 'loc:', location)
# Child class
class Employee(Person, Company):
  def Employee_info(self, salary, skill):
    print('Inside Employee class')
    print('Salary:', salary, 'Skill:', skill)

# Create object of Employee
emp = Employee()
# access data
emp.person_info('Jessa', 28)
emp.company_info('Google', 'Atlanta')
emp.Employee_info(12000, 'Machine Learning')
```





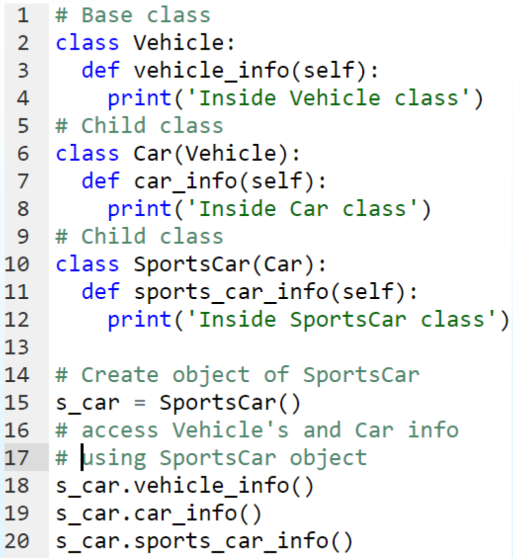
```
Inside Person class
Name: Jessa Age: 28
Inside Company class
Name: Google loc: Atlanta
Inside Employee class
Salary: 12000 Skill: Machine Learning
```

Multilevel Inheritance



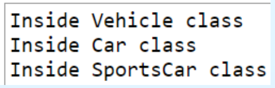
- A class inherits from a child class or derived class.
- Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B.
- In other words, we can say a chain of classes is called multilevel inheritance.

Multilevel Inheritance (2)



```
# Base class
class Vehicle:
  def vehicle_info(self):
    print('Inside Vehicle class')
# Child class
class Car(Vehicle):
  def car_info(self):
    print('Inside Car class')
# Child class
class SportsCar(Car):
  def sports_car_info(self):
    print('Inside SportsCar class')

# Create object of SportsCar
s_car = SportsCar()
# access Vehicle's and Car info
# using SportsCar object
s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

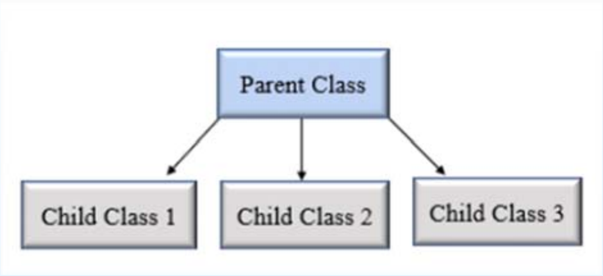
- Here, we can see there are three classes named `Vehicle`, `Car`, `SportsCar`.
- `Vehicle` is the superclass, `Car` is a child of `Vehicle`, `SportsCar` is a child of `Car`. So, we can see the chaining of classes.



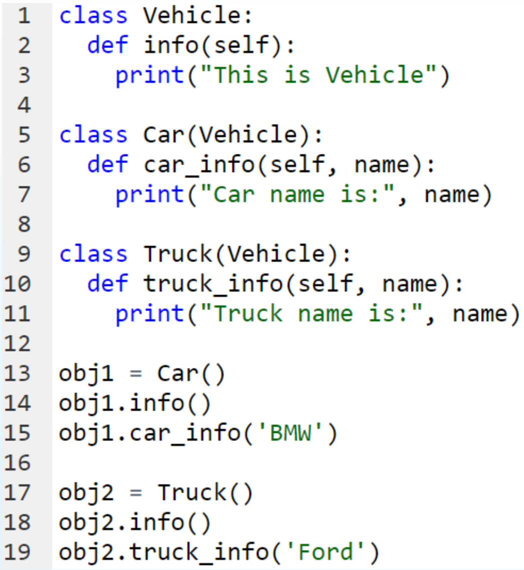
```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```

Hierarchical Inheritance

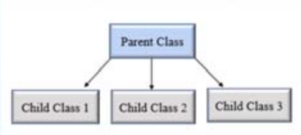
- More than one child class is derived from a single parent class.
- In other words, we can say one parent class and multiple child classes.

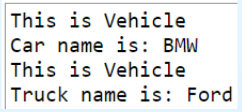


Hierarchical Inheritance (2)



```
class Vehicle:
  def info(self):
    print("This is Vehicle")

class Car(Vehicle):
  def car_info(self, name):
    print("Car name is:", name)

class Truck(Vehicle):
  def truck_info(self, name):
    print("Truck name is:", name)

obj1 = Car()
obj1.info()
obj1.car_info('BMW')

obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

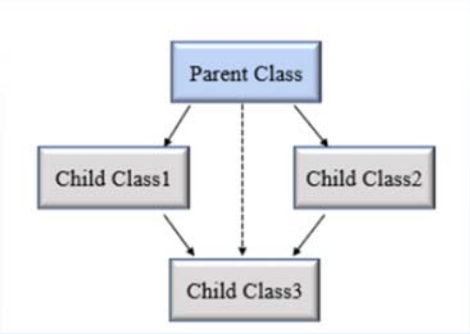




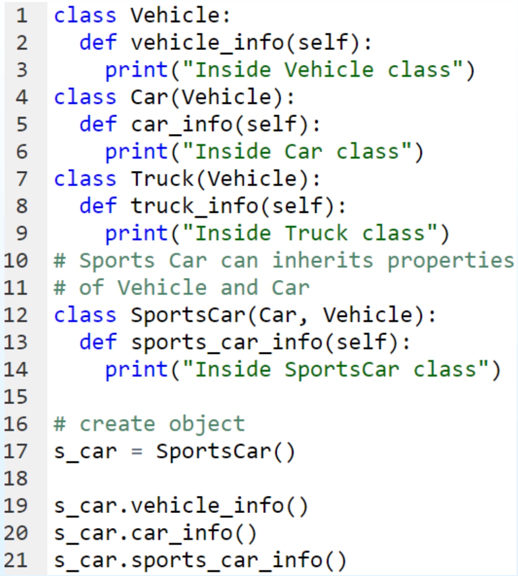
```
This is Vehicle
Car name is: BMW
This is Vehicle
Truck name is: Ford
```

Hybrid Inheritance

- When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.

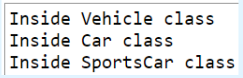


Hybrid Inheritance (2)



```
class Vehicle:
  def vehicle_info(self):
    print("Inside Vehicle class")
class Car(Vehicle):
  def car_info(self):
    print("Inside Car class")
class Truck(Vehicle):
  def truck_info(self):
    print("Inside Truck class")
# Sports Car can inherits properties
# of Vehicle and Car
class SportsCar(Car, Vehicle):
  def sports_car_info(self):
    print("Inside SportsCar class")

# create object
s_car = SportsCar()

s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

- Note here that hierarchical and multiple inheritance exists.
- Here we created, parent class Vehicle and two child classes named Car and Truck this is hierarchical inheritance.
- Another is SportsCar inherit from two parent classes named Car and Vehicle. This is multiple inheritance.

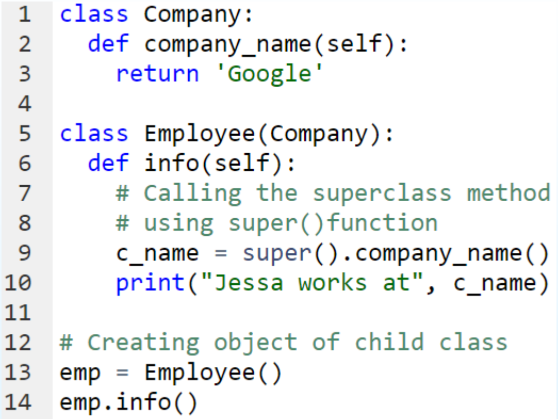


```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```

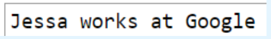
super() function

- When a class inherits all properties and behavior from the parent class, in such a case, the inherited class is a subclass, and the latter class is the parent class.
- In child class, we can refer to parent class by using the super() function.
- The super() function returns a temporary (reference to) object of the parent class that allows us to call a parent class method inside a child class method.
- Benefits of using the super() function
  - We are not required to remember or specify the parent class name to access its methods.
  - We can use the super() function in both single and multiple inheritances.
  - The super() function support code reusability as there is no need to write the entire function.

super() function (2)



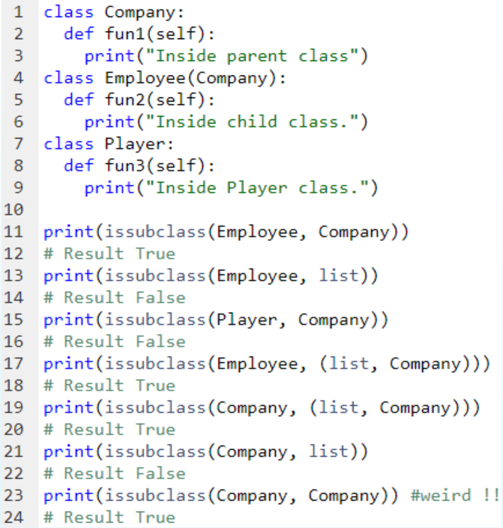
```
class Company:
  def company_name(self):
    return 'Google'

class Employee(Company):
  def info(self):
    # Calling the superclass method
    # using super()function
    c_name = super().company_name()
    print("Jessa works at", c_name)

# Creating object of child class
emp = Employee()
emp.info()
```



```
Jessa works at Google
```

- Here, we create a parent class Company and child class Employee.
- In Employee class, we call the parent class method by using a super() function.

issubclass() function



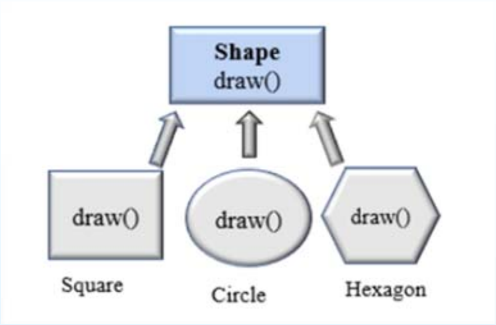
```
class Company:
  def fun1(self):
    print("Inside parent class")
class Employee(Company):
  def fun2(self):
    print("Inside child class.")
class Player:
  def fun3(self):
    print("Inside Player class.")

print(issubclass(Employee, Company))
# Result True
print(issubclass(Employee, list))
# Result False
print(issubclass(Player, Company))
# Result False
print(issubclass(Employee, (list, Company)))
# Result True
print(issubclass(Company, (list, Company)))
# Result True
print(issubclass(Company, list))
# Result False
print(issubclass(Company, Company)) #weird !!
# Result True
```

- In Python, we can verify whether a particular class is a subclass of another class.
- For this purpose, we can use Python built-in function issubclass(), which returns True if the given class is the subclass of the specified class. Otherwise, it returns False.
- Syntax:

`issubclass(class, classinfo)`

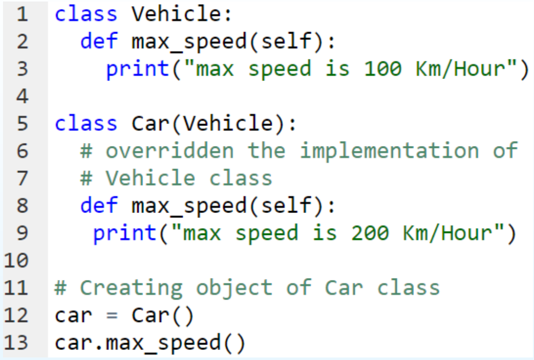
  - class: class to be checked
  - classinfo: a class, type, or a tuple of classes or data types.

Method Overriding

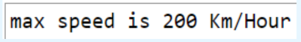


- In inheritance, all members available in the parent class are by default available in the child class.
- If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class. This concept is called method overriding.
- When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to override the method in the parent class.

Method Overriding (2)



```
class Vehicle:
  def max_speed(self):
    print("max speed is 100 Km/Hour")

class Car(Vehicle):
  # overridden the implementation of
  # Vehicle class
  def max_speed(self):
   print("max speed is 200 Km/Hour")

# Creating object of Car class
car = Car()
car.max_speed()
```



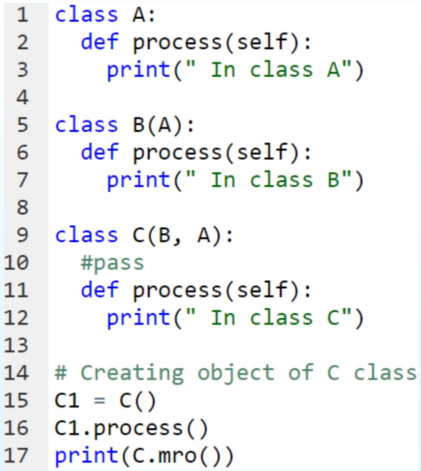
```
max speed is 200 Km/Hour
```

- Here, we create two classes named Vehicle (Parent class) and Car (Child class).
- The class Car extends from the class Vehicle so, all properties of the parent class are available in the child class.
- In addition to that, the child class redefined the method max_speed()..

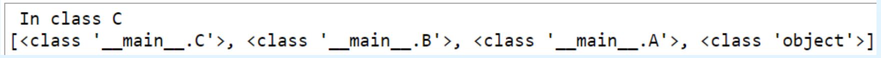
Method Resolution Order

- Method Resolution Order (MRO) is the order by which Python looks for a method or attribute.
- First, the method or attribute is searched within a class, and then it follows the order we specified while inheriting.
- This order is also called the linearization of a class, and a set of rules is called MRO.
- The MRO plays an essential role in multiple inheritances as a single method may found in multiple parent classes.
- In multiple inheritance, the following search order is followed:
  1. First, it searches in the current parent class, if not available then searches in the parent's class specified while inheriting (that is left to right.)
  2. We can get the MRO of a class. For this purpose, we can use either the mro attribute or the mro() method.

Method Resolution Order (2)



```
class A:
  def process(self):
    print(" In class A")

class B(A):
  def process(self):
    print(" In class B")

class C(B, A):
  #pass
  def process(self):
    print(" In class C")

# Creating object of C class
C1 = C()
C1.process()
print(C.mro())
```



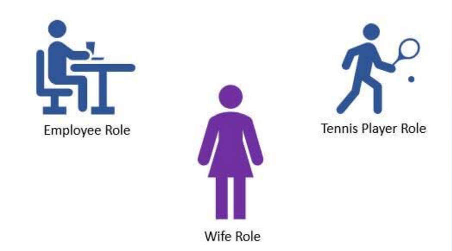
```
 In class C
[<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

- Here, we create three classes named A, B and C. Class B is inherited from A, class C inherits from B and A. When we create an object of the C class and calling the process() method, Python looks for the process() method in the current class in the C class itself.
- Then search for parent classes, namely B and A, because C class inherit from B and A. that is, C(B, A) and always search in left to right manner.

Polymorphism

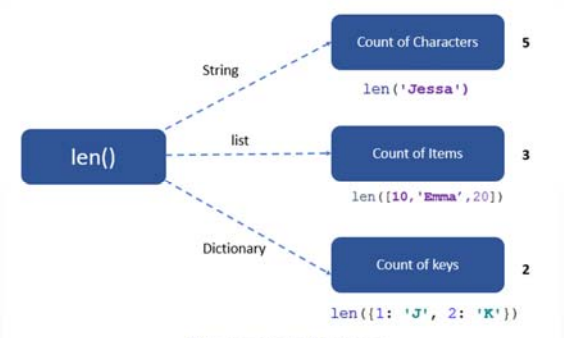
What's Polymorphism

- Polymorphism in Python is the ability of an object to take many forms.
- In simple words, polymorphism allows us to perform the same action in many different ways.
  - For example, Jessa acts as an employee when she is at the office. However, when she is at home, she acts like a wife. Also, she represents herself differently in different places. Therefore, the same person takes different forms as per the situation.
- In polymorphism, a method can process objects **differently** depending on the class type or data type.



Jessa takes different forms as per the situation

Polymorphism in Built-in function len()

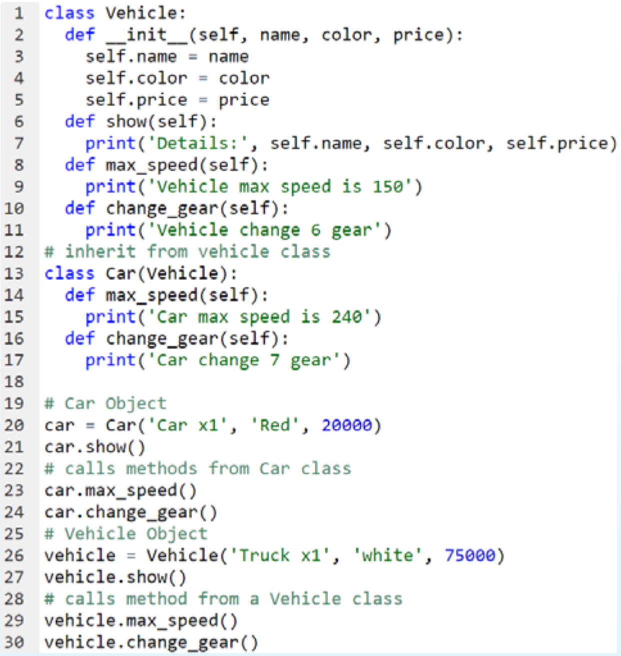


Polymorphic len() function

- The built-in function `len()` calculates the length of an object depending upon its type.
- If an object is a string, it returns the count of characters, and if an object is a list, it returns the count of items in a list.
- The `len()` method treats an object as per its class type.

Polymorphism with Inheritance

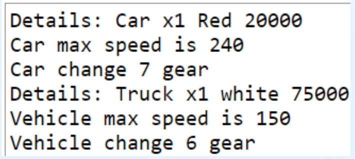
- Polymorphism is mainly used with inheritance.
- In inheritance, child class inherits the attributes and methods of a parent class.
- Using method overriding, polymorphism allows us to defines methods in the child class that have the same name as the methods in the parent class. This process of re-implementing the inherited method in the child class is known as Method Overriding.
- Advantage of method overriding
  - It is effective when we want to extend the functionality by altering the inherited method. Or the method inherited from the parent class doesn't fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way.
  - Method overriding is useful when a parent class has multiple child classes, and one of that child class wants to redefine the method. The other child classes can use the parent class method. Due to this, we don't need to modification the parent class code



```
class Vehicle:
  def __init__(self, name, color, price):
    self.name = name
    self.color = color
    self.price = price
  def show(self):
    print('Details:', self.name, self.color, self.price)
  def max_speed(self):
    print('Vehicle max speed is 150')
  def change_gear(self):
    print('Vehicle change 6 gear')
# inherit from vehicle class
class Car(Vehicle):
  def max_speed(self):
    print('Car max speed is 240')
  def change_gear(self):
    print('Car change 7 gear')

# Car Object
car = Car('Car x1', 'Red', 20000)
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()
# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.change_gear()
```

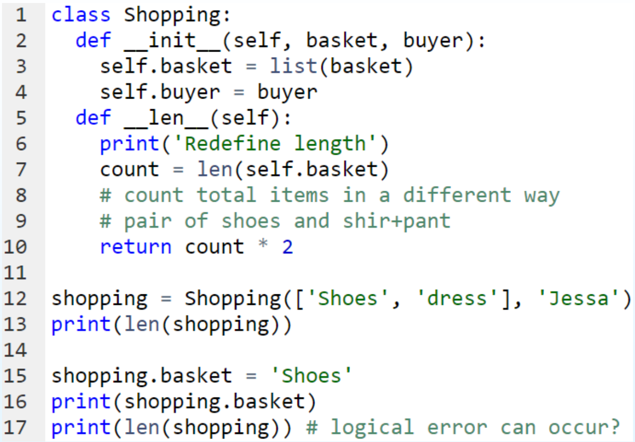
Polymorphism with Inheritance (2)

- As we can see, due to polymorphism, the Python interpreter recognizes that the max_speed() and change_gear() methods are overridden for the car object. So, it uses the one defined in the child class (Car).
- On the other hand, the show() method isn't overridden in the Car class, so it is used from the Vehicle class.

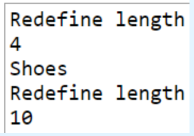


```
Details: Car x1 Red 20000
Car max speed is 240
Car change 7 gear
Details: Truck x1 white 75000
Vehicle max speed is 150
Vehicle change 6 gear
```

Override Built-in Functions



```
class Shopping:
  def __init__(self, basket, buyer):
    self.basket = list(basket)
    self.buyer = buyer
  def __len__(self):
    print('Redefine length')
    count = len(self.basket)
    # count total items in a different way
    # pair of shoes and shir+pant
    return count * 2

shopping = Shopping(['Shoes', 'dress'], 'Jessa')
print(len(shopping))

shopping.basket = 'Shoes'
print(shopping.basket)
print(len(shopping)) # logical error can occur?
```



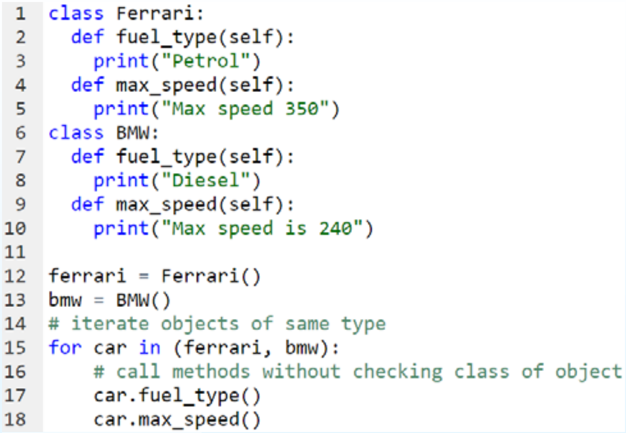
```
Redefine length
4
Shoes
Redefine length
10
```

- In Python, we can change the default behavior of the built-in functions.
- For example, we can change or extend the built-in functions such as len(), abs(), or divmod() by redefining them in our class.

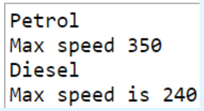
Polymorphism in Class methods

- Polymorphism with class methods is useful when we group different objects having the same method. We can add them to a list or a tuple, and we don't need to check the object type before calling their methods.
- Instead, Python will check object type at runtime and call the correct method.
- Thus, we can call the methods without being concerned about which class type each object is. We assume that these methods exist in each class.

Polymorphism in Class methods (2)



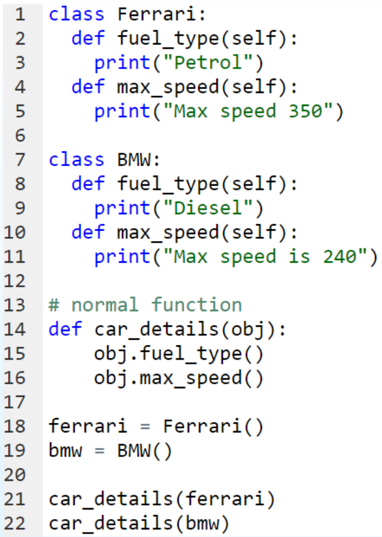
```
class Ferrari:
  def fuel_type(self):
    print("Petrol")
  def max_speed(self):
    print("Max speed 350")
class BMW:
  def fuel_type(self):
    print("Diesel")
  def max_speed(self):
    print("Max speed is 240")

ferrari = Ferrari()
bmw = BMW()
# iterate objects of same type
for car in (ferrari, bmw):
    # call methods without checking class of object
    car.fuel_type()
    car.max_speed()
```



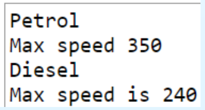
```
Petrol
Max speed 350
Diesel
Max speed is 240
```

- Python allows different classes to have methods with the same name.
  - Let's design a different class in the same way by adding the same methods in two or more classes.
  - Next, create an object of each class
  - Next, add all objects in a tuple.
  - In the end, iterate the tuple using a for loop and call methods of an object without checking its class.
- In the example, fuel_type() and max_speed() are the instance methods created in both classes.

Polymorphism with Functions and Objects



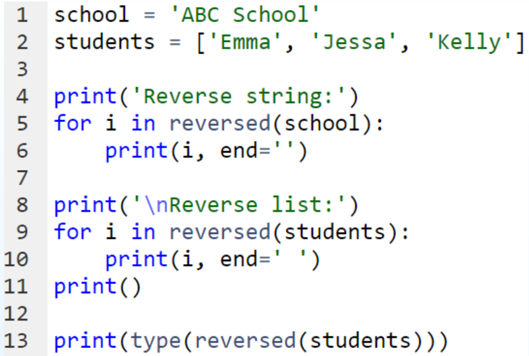
```python
class Ferrari:
  def fuel_type(self):
    print("Petrol")
  def max_speed(self):
    print("Max speed 350")

class BMW:
  def fuel_type(self):
    print("Diesel")
  def max_speed(self):
    print("Max speed is 240")

# normal function
def car_details(obj):
    obj.fuel_type()
    obj.max_speed()

ferrari = Ferrari()
bmw = BMW()

car_details(ferrari)
car_details(bmw)
```



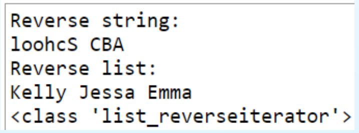
```
Petrol
Max speed 350
Diesel
Max speed is 240
```

- We can create polymorphism with a function that can take any object as a parameter and execute its method without checking its class type.
- Using this, we can call object actions using the same function instead of repeating method calls.

Polymorphism in Built-in Methods



```python
school = 'ABC School'
students = ['Emma', 'Jessa', 'Kelly']

print('Reverse string:')
for i in reversed(school):
    print(i, end='')

print('\nReverse list:')
for i in reversed(students):
    print(i, end=' ')
print()

print(type(reversed(students)))
```

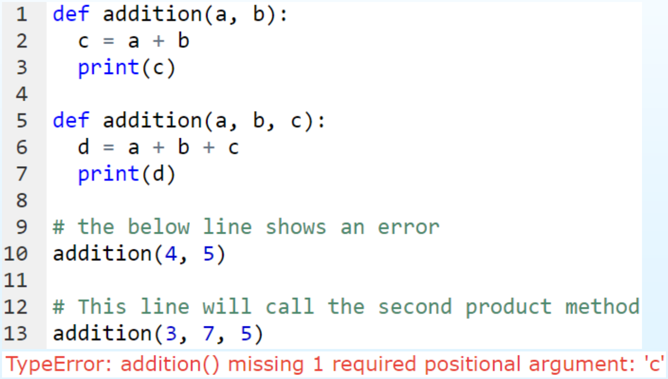


```
Reverse string:
loohcS CBA
Reverse list:
Kelly Jessa Emma
<class 'list_reverseiterator'>
```

- The word polymorphism is taken from the Greek words poly (many) and morphism (forms). It means a method can process objects differently depending on the class type or data type.
- The built-in function `reversed(obj)` returns the iterable by reversing the given object.
- For example, if you pass a string to it, it will reverse it. But if you pass a list of strings to it, it will return the iterable by reversing the order of elements (it will not reverse the individual string).

Method Overloading

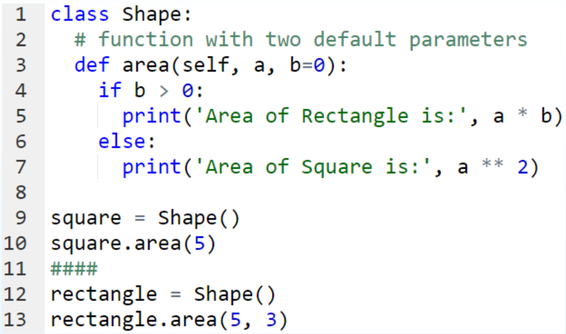
- The process of calling the same method with different parameters is known as method overloading.
- Python does not support method overloading.
- Python considers only the latest defined method even if you overload the method. Python will raise a `TypeError` if you overload the method.



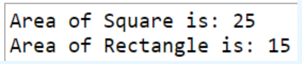
```python
def addition(a, b):
  c = a + b
  print(c)

def addition(a, b, c):
  d = a + b + c
  print(d)

# the below line shows an error
addition(4, 5)

# This line will call the second product method
addition(3, 7, 5)
```

```
TypeError: addition() missing 1 required positional argument: 'c'
```

Method Overloading (2)



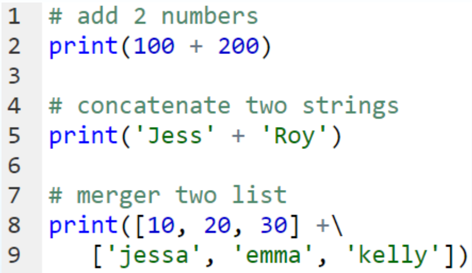
```python
class Shape:
  # function with two default parameters
  def area(self, a, b=0):
    if b > 0:
      print('Area of Rectangle is:', a * b)
    else:
      print('Area of Square is:', a ** 2)

square = Shape()
square.area(5)
####
rectangle = Shape()
rectangle.area(5, 3)
```



```
Area of Square is: 25
Area of Rectangle is: 15
```

- To overcome the above problem, we can use different ways to achieve the method overloading. In Python, to overload the class method, we need to write the method's logic so that different code executes inside the function depending on the parameter passes.
- Let's assume we have an `area()` method to calculate the area of a square and rectangle. The method will calculate the area depending upon the number of parameters passed to it.
  - If one parameter is passed, then the area of a square is calculated
  - If two parameters are passed, then the area of a rectangle is calculated.

Operator Overloading in Python

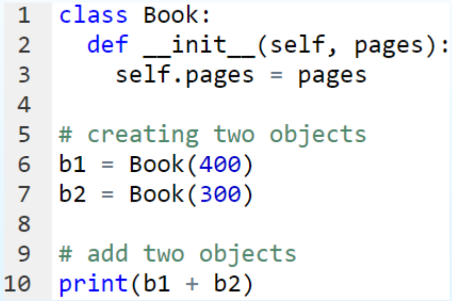


```
# add 2 numbers
print(100 + 200)

# concatenate two strings
print('Jess' + 'Roy')

# merger two list
print([10, 20, 30] +\
   ['jessa', 'emma', 'kelly'])
```

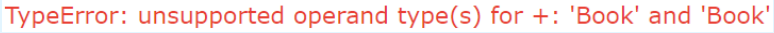
- Operator overloading means changing the default behavior of an operator depending on the operands (values) that we use. In other words, we can use the same operator for multiple purposes.
- For example, the + operator will perform an arithmetic addition operation when used with numbers. Likewise, it will perform concatenation when used with strings.
- The operator + is used to carry out different operations for distinct data types.
- This is one of the simplest occurrences of polymorphism in Python.

Overloading + operator for custom object

- Suppose we have two objects, and we want to add these two objects with a binary + operator.
- However, it will throw an error if we perform addition because the compiler doesn't add two objects.



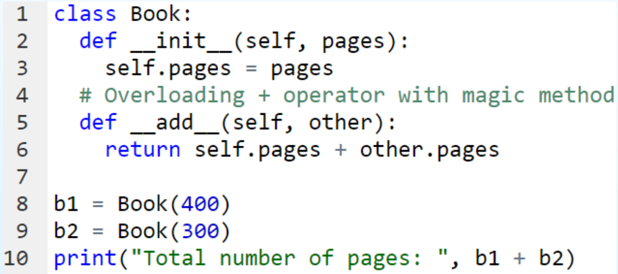
```
class Book:
  def __init__(self, pages):
    self.pages = pages

# creating two objects
b1 = Book(400)
b2 = Book(300)

# add two objects
print(b1 + b2)
```



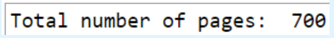
```
TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

Overloading + operator for custom object (2)

- We can overload + operator to work with custom objects also. Python provides some special or magic function that is automatically invoked when associated with that particular operator.
- For example, when we use the + operator, the magic method __add__() is automatically invoked.
- Internally + operator is implemented by using __add__() method. We have to override this method in our class if you want to add two custom objects.

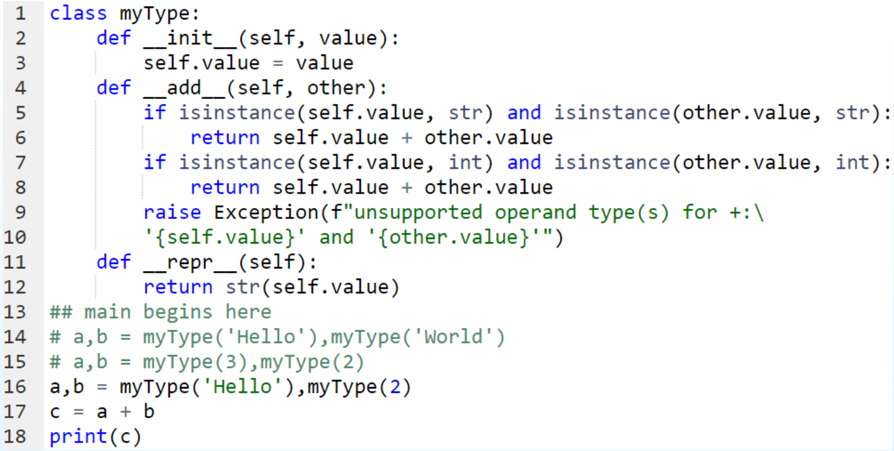


```
class Book:
  def __init__(self, pages):
    self.pages = pages
  # Overloading + operator with magic method
  def __add__(self, other):
    return self.pages + other.pages

b1 = Book(400)
b2 = Book(300)
print("Total number of pages: ", b1 + b2)
```

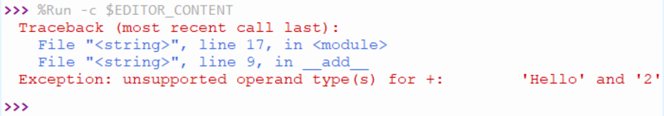


```
Total number of pages:  700
```

Overloading + operator for custom object (3)

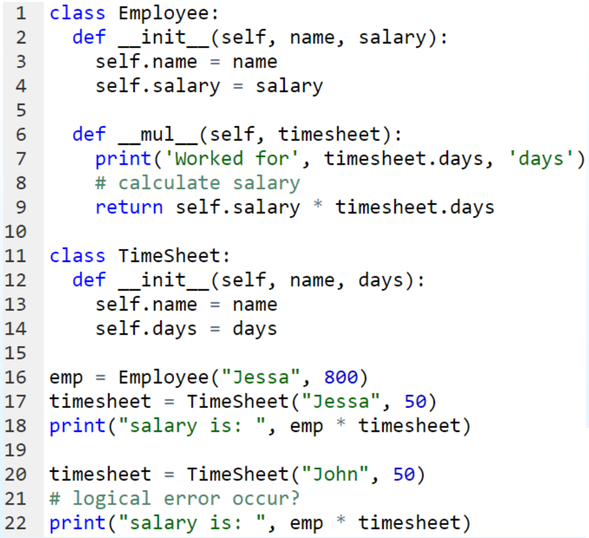


```
1  class myType:
2      def __init__(self, value):
3          self.value = value
4      def __add__(self, other):
5          if isinstance(self.value, str) and isinstance(other.value, str):
6              return self.value + other.value
7          if isinstance(self.value, int) and isinstance(other.value, int):
8              return self.value + other.value
9          raise Exception(f"unsupported operand type(s) for +:\
10         '{self.value}' and '{other.value}'")
11     def __repr__(self):
12         return str(self.value)
13 ## main begins here
14 # a,b = myType('Hello'),myType('World')
15 # a,b = myType(3),myType(2)
16 a,b = myType('Hello'),myType(2)
17 c = a + b
18 print(c)
```



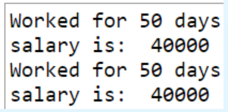
```
>>> %Run -c $EDITOR_CONTENT
 Traceback (most recent call last):
   File "<string>", line 17, in <module>
   File "<string>", line 9, in __add__
 Exception: unsupported operand type(s) for +:        'Hello' and '2'

>>>
```

Overloading the * operator



```python
class Employee:
  def __init__(self, name, salary):
    self.name = name
    self.salary = salary

  def __mul__(self, timesheet):
    print('Worked for', timesheet.days, 'days')
    # calculate salary
    return self.salary * timesheet.days

class TimeSheet:
  def __init__(self, name, days):
    self.name = name
    self.days = days

emp = Employee("Jessa", 800)
timesheet = TimeSheet("Jessa", 50)
print("salary is: ", emp * timesheet)

timesheet = TimeSheet("John", 50)
# logical error occur?
print("salary is: ", emp * timesheet)
```

- The * operator is used to perform the multiplication.
- Let’s see how to overload it to calculate the salary of an employee for a specific period.
- Internally * operator is implemented by using the `__mul__()` method.



```
Worked for 50 days
salary is:  40000
Worked for 50 days
salary is:  40000
```

Magic Methods

Operator Name	Symbol	Magic Method
Addition	+	`__add__(self, other)`
Subtraction	-	`__sub__(self, other)`
Multiplication	*	`__mul__(self, other)`
Division	/	`__div__(self, other)`
Floor Division	//	`__floordiv__(self,other)`
Modulus	%	`__mod__(self, other)`
Power	**	`__pow__(self, other)`
Increment	+=	`__iadd__(self, other)`
Decrement	-=	`__isub__(self, other)`
Product	*=	`__imul__(self, other)`
Division	/+	`__idiv__(self, other)`
Modulus	%=	`__imod__(self, other)`

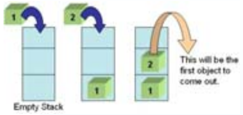
Magic Methods (2)

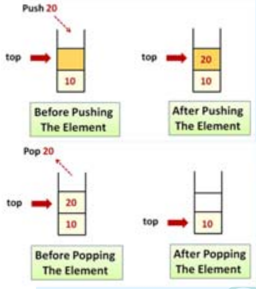
Operator Name	Symbol	Magic Method
Power	**=	`__ipow__(self, other)`
Less than	<	`__lt__(self, other)`
Greater than	>	`__gt__(self, other)`
Less than or equal to	<=	`__le__(self, other)`
Greater than or equal to	>=	`__ge__(self, other)`
Equal to	==	`__eq__(self, other)`
Not equal	!=	`__ne__(self, other)`

Sample Problem Solving

OOP Stack (LIFO) Implementation

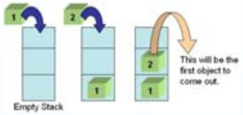
```python
class Stack:
  def __init__(self):
    self.items = []
  def push(self, item):
    self.items.append(item)
  def pop(self):
    if not self.is_empty():
      return self.items.pop()
    else:
      return "Cannot pop from an empty stack."
  def is_empty(self):
    return len(self.items) == 0
  def size(self):
    return len(self.items)
  def peek(self):
    if not self.is_empty():
      return self.items[-1]
    else:
      return "Empty stack."
```




Push 20
top
10
top
20
10
Before Pushing The Element
After Pushing The Element
Pop 20
top
20
10
top
10
Before Popping The Element
After Popping The Element

OOP Stack (LIFO) Implementation (2)

```python
# Example usage
stack = Stack()
stack.push(0)
stack.push(1)
stack.push(2)
stack.push(3)
stack.push(4)

print("Stack size:", stack.size())
print("Top element:", stack.peek())
popped_item = stack.pop()
print("\nPopped item:", popped_item)
print("\nStack size:", stack.size())
print("Top element:", stack.peek())
#---------------------------------------
stack1 = Stack()
print("\nStack size:", stack1.size())
popped_item = stack1.pop()
print("\nPopped item:", popped_item)
```



Palindrome check using Stack

```python
#### Palidrome check using Stack
myStack = Stack()
#s = 'A SANTA LIVED AS A DEVIL AT NASA'
s = 'Civic'
tmp, textInput = [i for i in s if i!=' '], ''
for i in tmp:
    textInput += i.lower()

for c in textInput:
    myStack.push(c)

revTextInput = ''
while not myStack.is_empty():
    revTextInput += myStack.pop()

if textInput == revTextInput:
    print(f'[{s}] is a palindrome.')
else:
    print(f'[{s}] is not a palindrome.')
```

```
[Civic] is a palindrome.
```

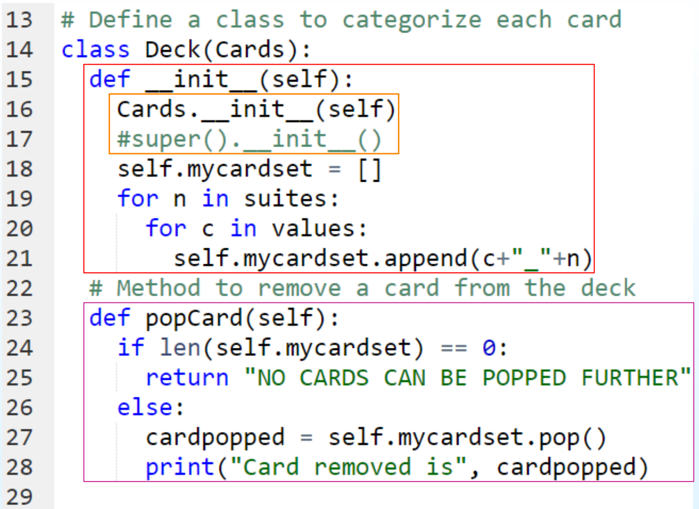
Shuffle a deck of cards using a Python random module

```python
from random import shuffle

# Define a class to create all type of cards
class Cards:
  global suites, values
  suites = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
  values = ['A', '2', '3', '4', '5', '6', '7', '8']
  values += ['9', '10', 'J', 'Q', 'K']
  def __init__(self):
    pass
```

https://www.geeksforgeeks.org/shuffle-a-deck-of-card-with-oops-in-python/

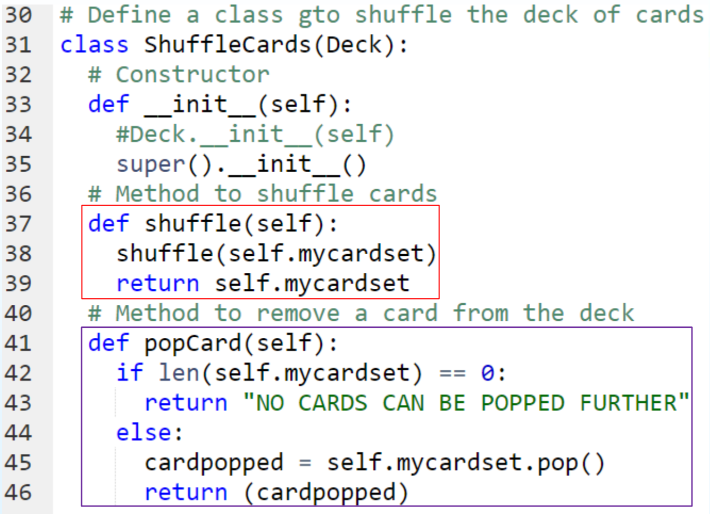
https://en.wikipedia.org/wiki/Shuffling

Shuffle a deck of cards using a Python random module (2)



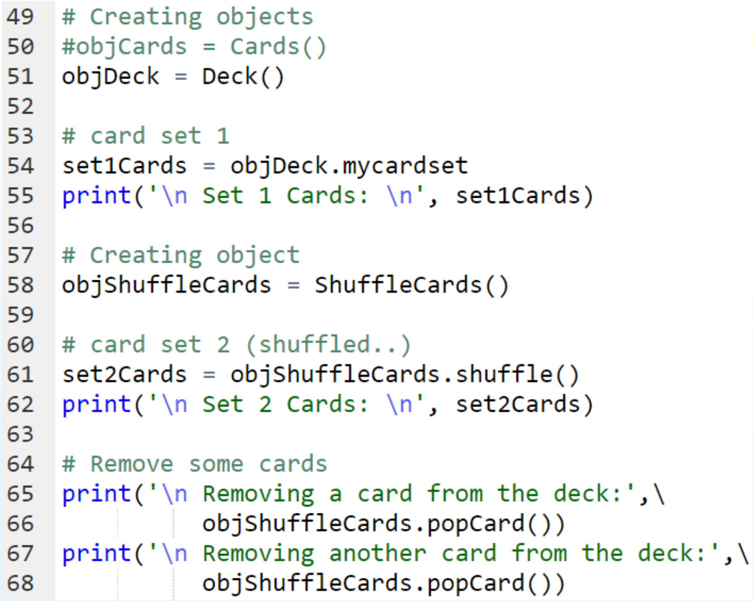
```
# Define a class to categorize each card
class Deck(Cards):
 def __init__(self):
  Cards.__init__(self)
  #super().__init__()
  self.mycardset = []
  for n in suites:
   for c in values:
    self.mycardset.append(c+"_"+n)
 # Method to remove a card from the deck
 def popCard(self):
  if len(self.mycardset) == 0:
   return "NO CARDS CAN BE POPPED FURTHER"
  else:
   cardpopped = self.mycardset.pop()
   print("Card removed is", cardpopped)

```

Shuffle a deck of cards using a Python random module (3)



```
# Define a class gto shuffle the deck of cards
class ShuffleCards(Deck):
 # Constructor
 def __init__(self):
  #Deck.__init__(self)
  super().__init__()
 # Method to shuffle cards
 def shuffle(self):
  shuffle(self.mycardset)
  return self.mycardset
 # Method to remove a card from the deck
 def popCard(self):
  if len(self.mycardset) == 0:
   return "NO CARDS CAN BE POPPED FURTHER"
  else:
   cardpopped = self.mycardset.pop()
   return (cardpopped)
```

Shuffle a deck of cards using a Python random module (4)



```
# Creating objects
#objCards = Cards()
objDeck = Deck()

# card set 1
set1Cards = objDeck.mycardset
print('\n Set 1 Cards: \n', set1Cards)

# Creating object
objShuffleCards = ShuffleCards()

# card set 2 (shuffled..)
set2Cards = objShuffleCards.shuffle()
print('\n Set 2 Cards: \n', set2Cards)

# Remove some cards
print('\n Removing a card from the deck:',\
     objShuffleCards.popCard())
print('\n Removing another card from the deck:',\
     objShuffleCards.popCard())
```

To be continue..

つづく