Learn to Code with Python

OOP Basics:Class Definitions and Instantiation

- Example
  - class Person (): // Creates a namespace
    - pass // A reserved that is a null operator used for stubs
  - class DatabaseConnection():
    - pass
  - Boris = Person() // Instantiates an object

OOP Basics:__init__ Method

- considered private methods
- class Guitar():
  - def __init__(self): // Set the initial state of each object. Run only once (Instantiate an object)
    - print(“ A new guitar is being crated”)
- guitar = Guitar() // will print “A new guitar is being created”
- self represent the object being created
  - 1st input to methods so you can access the attributes
  - Example – How the self works
    - obj = MyClass
    - obj.method() // Could ignore the syntactic sugar of the dot call syntax (object method) and pass the instance
      // object manually to get the same result
  - Example – pass the instance in manually
    - MyClas.mehtod(obj) // Pass the instance object manually to get the same result
  - Instance methods can access the class itself : self.__class__ attribute
    - Instance methods are powerful since they can modify state and class data

OOP Basics:Adding Attributes to Objects

- Are public by default and can be accessed by the dot operator
- Add an attribute to the instance after it has been defined
  - Example
    - acoustic = Guitar()
    - electric = Guitar()
    - acoustic.wood = “oak”
    - acoustic.strings = “6”
    - acoustic.year = 1990
    - electric.nickname = “Sound Viking 3000”
    - print(electric.year) would produce a name error since it was not defined
  - Anti Pattern → Since they are the same object they should share the same attributes. If two instances have the same object, but different attributes how do know if the instance will have the attribute that you need.
    - Without consistency it is hard to write sustainable productive programs.

OOP:Define Properties with the Property Method

- Why
  - Used to mask the complexities of data. Store it differently in the class that we would present it to the user
  - Define some validation on the attribute
- property accepts four arguments (setter method, getter method, deletor method and a doc string)
- If the property and attribute are the identical then there is a problem so either use different names or prefix the protected attribute with the underscore.
- Example
  - class Height():
    - def __init__(self, feet):
      - self._inches = feet * 12
    - def _get_feet():
      - return self._inches / 12
    - def _set_feet(self,feet):
      - If (feet >= 0):
        - self._inches = feet * 12
    - feet = property(_get_feet, _set_feet) // produces the appearance of a property
- h = Height(5)
- print(h.feet)
- h.feet = 6 // Set the self._inches = 60

OOP: Basics:Define Properties with the Property Method (Alternate Approach)

- Want a method to resemble a property to the user
- The setter and getter have the same name
- Example
  - class Currency():
    - def __init__(self, dollars):
      - self._cents = dollars * 100;
    - @property // A decorator
    - def dollars(self):
      - return self._cents / 100
    -
    - @dollars.setter
    - def dollars(self, dollars):
      - if (dollars > 0):
        - self._cents = dollars * 100
  - bank_account = Currency(50000)
  - print(bank_account.dollars) // Would print 50000. the dollars is seen as a proper because of the @property
  - bank_account.dollars = 100000 // Calls the method with the associated @dollars.set

OOP: Basics: Setting Object Attriburtes in the __init__ method

- Setting Objects Attributes in the __init__ method
  - Example
    - class Guitar():
      - def __init__(self, wood):
        - self.wood = wood
    - acoustic = Guitar(“Alder”)
    - electric = Guitar(“Mahogany”)
    - Baritone = Guitar() produces a Type Error

OOP: Basics: Default Values for Attributes

- Example
  - def Book():
    - def __init__(title, author, price = 14.00): // the 14.00 is a default price
      - self.title = title
      - self.author = author
      - self.price = price

OOP Attributes & Methods:Instance Methods

- Example
  - class Pokeman():
    - def __init__(self, name, speciality, health=100)
      - self.name = name
      - self.speciality = speciality
      - self.health = health
    - def roar(self):
      - print(“Raaaarr!”)
    - def take_damage(self, amount):
      - self.health -= amount
  - squirtle = Pokemon(“Squirtle”, “Water”)
  -
  - squirtle.roar();
  - squirtle.take_damage(30)
- All System methods take self as the 1st parameter which Python does automatically

OOP: Attributes & Methods:Protected Attributes and Methods

- Example
  - class SmartPhone():
    - def __init__(self):
      - self._company = “Apple”
      - self._firmaware = 10.0
    - def get_os_version(self):
      - return self._firmware
    - def update_firmware(self):
      - self._firmware += 1

Add the underscore to tell developers that they should not modify the attribute

OOP: Attributes & Methods:GetAttr and SetAttr

- Allow to get and set the value of an object without using the Dot Syntax
  - Used when we do not know the attribute we are read/writing in advance
    - may be provide by the user
    - passed by another part of the program
    - dynamically created on the fly
- SetAttr creates an attribute on the fly
- GetAttr retrieves the attribute and returs a default attribute if the attribute name is not found.
- Example
- stats = {
  - “name” : “BBQ Chicken”,
  - “price”: 19.99
  - “size”: “Extra Large”
  - “ingredients”: [“Chicken”, “Onions”, “BBQ Sauce”]
- }

OOP: Attributes & Methods:GetAttr and SetAttr

- Example (continued from the previous page)
  - class Pizza():
    - def __init__(self,stats):
      - for key,value in stats.items(): // If self.key was used then it would have created the attribute key instead of
        - setattr(self, key, value) // the real name such size or price.
  - bbq = Pizza(Stats)
  - print(bbq.size)
  - for attr in [“price”, “name”, “diameter”, “discounted”]
    - print(getattr(bbq, attr, “Unknown”,))
  - Output
    - 19.99
    - BBQ Chicken
    - Unknown
    - Unknown

OOP: Attributes & Methods:The hasattr and deleteattr Functions

- Use the pizza example from section “GetAttr” and “SetAttr”
- hasattr will return true if the attribute exist on the object
- delattr will delete an attribute on the object
- Example – Remember working from the previous example (from Section “GetAttr” and “SetAttr”)
  - stats_to_delete = [“size”, “diameter”, “spiceness”, “ingrediants”]
  - for stat in stats_to_delete:
    - if (has_attr(bbq, stat)):
      - delattr(self, stat)
- print(bbq.size) // AttributeError ‘Pizza’ object has no attribute size
-

OOP Attributes & Methods:Class Methods

- Methods on the class itself and cannot modify object instance stae
- Use Cases
  - Create preconfigured Object (ex. builder pattern)
- Example
  - class SushiPlatter():
    - def __init__(self, salmon, tuna, shrimp, squid):
      - self.salmon = salmon
      - self.tuna = tuna
      - self.shrimp = shrimp
      - self.squid = squid
    - @classmethod // annotation to make it a class method and cls is parameter that is the // class itself
    - def lunch_special_A(cls):
      - return cls(salmon=2, tuna = 2, shrimp = 2, squid = 0) // Instantiate the class
  - Boris = SushiPlatter(salmon = 8, tuna =4 , shrimp =5 squid = 10)
  - print(Boris.salmon) // Would print 8
  - lunch_eater = SushiPlatter.lunch_special_A():
  - print(lunch_eater.tuna) // result = 4

OOP Attributes & Methods:Class Methods

- Example of a Class Variable
  - class A:
    - a = “Hello Chuck”
  - x = A()
  - x.a, A.a would produce Hello Chuck
- To change an class attribute use the classname and attribute Name

OOP: Attributes & Methods:Class Attributes

- Class Attributes are attributes that are part of the class not the instance
- obj.classMethod() calls the method visible to the object
- Used to store a piece of data that is not tied to the state of a specific instance such as globals, number of instances
- Example
  - class Counter():
    - count = 0
    - def __init__(self):
      - Counter.count += 1
    - def create_two(cls):
      - two_counters = [cls(), cls()]
      - print(f"New Number of Counter Objects create {cls.count}")
      - return tow_counters
  - print(Counter.counter) // Will print out 0
  - c1 = Counter()
  - print(Counter.counter) // Will print out 1
  - c2, c3 = Counter.create_two()
  - print(Counter.count) // Will print out 3
-

OOP: Attributes & Methods:Class Attributes Attribute Lookup Order

- Class attributes are shared among all instance.
  - Only one attribute will be created for each instance
- Example – Ran the previous example
  - print(c1.count) // will print 3
  - print(c2.count) // will print 3
  - print(c3.count) // will print 3
- Attribute Lookup Order
  - Example
    - class Example():
      - data = “Class Attrtibute”
    - e1 = example()
    - e2 = example()
    - print(e1.data) // It does not have an instance attribute, then it uses the class attribute
    - e1.data = “Instance Attribute”
    - print(e1.data) // print the instance attribute
    - print(e2.data) // print the class attribute
    - del(e1.data) // Remove the data instance
    - print(e1.data) // prints out the class attribute

OOP: Attributes & Methods: Static Method

- used for utility operations that affect neither the class or instance
- Serve as a way to namespace your methods (Need to research)
- Good for functions that don’t modify instance nor class variables which is good about communication for your class structure and they are easier to test
- Example
  - class Weather():
    - def __init__(temperatures):
      - self.temperatures = temperates
    - @staticmethod
    - def convert_from_fahrenheit_to_celsius(fahr): // Does not need any access to the instance or class variable or methods
      - calculation = (5/9) * (fahr - 32)
      - return round(calculation,0)
    - def in_celcius(self):
      - return [self.convert_from_fahrenheit_to_celsius(temp) for temp in self.temperatures]
  - wf = WeatherForecast([100, 90. 80, 70. 60])
  - print(wf.in_celsius())
- obj.staticmethod()

OOP: Magic Methods:Intro to Magic Methods

- Make our classes behave just like python classes
  - Must use the specific name. Example to convert your object to a class use __str__
  - Make all the classes consistent and have no surprises
  - Operator Overloading → To use common python operators against your own class
- Thundar Method the following pattern “__” + <name> + “__” Example __str__
  - Act as hooks for our classes
- A hook is a procedure that intercepts a process at some point in execution
- A magic method is a hook that is called at the implicity by Python behind the scence at the right moment
- Developers don’t call a magic method directly
  - Example print(3.3 + 4.4) which python implements print(3.3.__add__(4.4))
  - Example print(len[1,2,3]) which python implements print([1,2,3].__len__())
  - Example print(“h” in “hello”) which python implements print(“hello”.__contains__(‘h”))
  - Example print([“a”, “b”, “c”][2]) which python implements print([“a”, “b”, “c”].__getitem__[2]))
- An example magic method is __add__(self, other)

OOP Magic Methods:String Representation with the __str__ and __repr__

- Example
  - class Card():
    - def __init__(self, rank, suit):
      - self.rank = rank
      - self.suit = suit
    - c = Card(“Ace”, “Spades”)
    - print(c) // Display the class and the memory location <__main__.Card object at 0x1035a01d0>
- __str__ A very high level explanation of the object
- Example
  - class Card():
    - def __init__(self, rank, suit):
      - self.rank = rank
      - self.suit = suit
    - def __str__(self, rank, suit):
      - return f” The card is { self.rank} of {self.suit}”
    - c = Card(“Ace”, “Spades”)
    - print(c) // Displays : The card is Ace of Spades

OOP Magic Methods:String Representation with the __str__ and __repr__

- __repr__
  - A very technical explanation of the object
  - If possible it should look live a valid python expression used to recreate the object with the same value
- Example
  - class Card():
    - def __init__(self, rank, suit):
      - self.rank = rank
      - self.suit = suit
    - def __repr__(self):
      - rerturn f'Card"{self.rank }", "{self.suit}")'
  - c = Card("Ace", "Spades")
  - print(repr(c)) would produce Card("Ace", "Spades")
- Order : searches for a user defined __str__ then the default __str__, searches for a user defined __str__ then the default __str__
- print(c.__repr__()) is the same as print(repr(c))

Mixin

- Mixins take various forms depending on the language, but at the end of the day they encapsulate behavior that can be reused in other classes.
- Difference between multiple inheritance and mixins come down to a mixin is independent enough that it does not feel the same as a parent class
- Example LoggerMixin
  - import logging
  - class LoggerMixin(object):
  - @property
  - def logger(self):
  - name = '.'.join([
  - self.__module__,
  - self.__class__.__name__
  -])
  - return logging.getLogger(name)

Mixin

- Example – adding a mixin
- class EssentialFunctioner(LoggerMixin, object):
- def do_the_thing(self):
- try:
- ...
- except BadThing:
- self.logger.error('OH NOES')
-
- class BusinessLogicer(LoggerMixin, object):
- def __init__(self):
- super().__init__()
- self.logger.debug('Giving the logic the business...')

OOP Magic Methods:Equality with the __eq__ Method

- When comparing two custom object Python has no clue which object are important to equality
- Example
  - class student():
    - def __init__(self, math, history, writing):
      - self.math = math
      - self.history = history
      - self.writing = writing
    - @property
    - def grades(self):
      - return self.math + self.history + self.writing
    - def __eq__(self, other_student): self I the current student and is on the left hand side of the ==
      - return (self.grades == other_students.grades)
  - bob = Student(math = 90, history = 90, writing = 90)
  - moe = Student(math = 100, history = 90, writing = 80)
  - joe = Student(math = 40, history = 45, writing = 50)
  - print(bob == moe) // Would return true
  - print(bob == joe) // Would return false
  - print (bob != job) // Would return true

OOP Magic Methods:Magic Method s for Comparison Operations

- Also convenience for __add__, __sub__ among others for mathematical operations
- class student():
  - def __init__(self, math, history, writing):
    - self.math = math
    - self.history = history
    - self.writing = writing
  - @property
    - def grades(self):
    - return self.math + self.history + self.writing
    - def __gt__(self, other_student):
      - return (self.grades > other_students.grades)
    - def__le__(self, other_student):
      - return (self.grades <= other_students.grades)
  - bob = Student(math = 90, history = 90, writing = 90)
  - moe = Student(math = 100, history = 90, writing = 80)
  - joe = Student(math = 40, history = 45, writing = 50)
  - print(bob > joe) // Would return true
  - print(bob <= moe) // Would return true

OOP Magic Methods:Doc Strings

- A DocString is a regular python string that creates documentation for a piece your program
- Triple Quotes allows for the multiple lines of text
- A doc string should be the first line of the module, function, class or anywhere it is used
- Example of a Doc Strings in sushi.py
  - “””
  - A module relate to the joy of sushi
  - No Fishy code found here
  - The first lines of a module
  - “””
  - def fish():
    - “”
    - determines if fish is a good meal choice
    - Always returns true because it always is
    - “”
  - class Salmon():
    - “””
    - Blueprint for Salmon Object
    - “””

OOP Magic Methods:Doc Stirngs

- In another file
  - import Sushi
  - print(sushi.__doc__)
  - print(sushi.fish.__doc__)
  - print(sushi.Salmon.__doc__)
- __doc__ is automatically set by Python using the Doc String. Don’t set it yourself
- help(sushi) will group all the module information , the description, the class and more information

OOP: Magic Methods:The Truthiness with the Bool Method

- Example
  - class Emotion():
    - def _init(self,positivity, negativity):
      - self.positivity = positivity
      - self.negativity = negativity
    - def __bool__(self): // Determines if the state (properties) is true or not
      - return self.positivity > negativity
    - my_emotional_state = Emotion(positivity = 50, negativity = 75)
    - if my_emotional_state: // call __bool__ since we are generating a boolean result
      - print(“This will not print because I have more negativity”)
    - my_emotional_state.positivity = 100
    - if my_emotional_state:
      - print(“This will print since positivity is 100 and negativity is 75”)

OOP: Magic Methods:The namedTuple Object

- Object with attributes, but no methods
  - Example Database Record
- Example
  - import collections
  - Book = collections.namedtuple(“Book”, [“title”, “author”)) // 1 parameter is name of object then rest are the attributes
  - Book = collections.namedtuple(“Book”, “title author”)) // Use a string to defined the attributes
  - animal_farm = Book(“Animal Farm”, “George Orwell”) // An new book instantiated using Book
  - print(animal_farm[0]) // Would print “Animal Farm”
  - print(animal_farm.title) // Would print “Animal Farm

OOP: Magic Methods:The Length with the __len__

- Allows the object to have a concept of len
- Example
  - Book = collections.namedtuple(“Book”, [“title”, “author”))
  - animal_farm = Book(“Animal Farm”, “George Orwell”)
  - gatsby = Book(title = “The Great Gatsby”, author = “F. Scott Fritzgerald”)
  - class Library():
    - def __init__(self, *books):
      - self.books = books
      - self.librarians = []
    - def __len__(self):
      - return len(self.books)
  - l1 = Library(animal_farm)
  - l2 = Library(animal_farm, gatsby)
  - print(len(l1)) // prints 1
  - print(len(l2)) // prints 2

OOP: Magic Methods:index with the __get_item__ and __set_item__ Methods

- Allow us to define the index logic for our own items
- Examples
  - pillows = { “soft”: 79.99, “hard”: 99.99 }
  - print(pillows[“soft”]) and print(pillows.__getitem__(“soft”))
- Example
  - class CrayonBox():
    - def __init__(self):
      - self.crayons = []
    - def add(self, crayon):
      - self.crayons.append(crayon)
  - cb = CrayonBox()
  - cb.add(“Blue”)
  - cb.add(“Red”)
  - print(cb[0]) // Get error CrayonBox object is not subscriptable
-

OOP: Magic Methods:index with the __getitem__ and __setitem__ Methods

- Example – Adding indexing
  - class CrayonBox():
    - def __init__(self):
      - self.crayons = []
    - def add(self, crayon):
      - self.crayons.append(crayon)
    - def __getitem__(self, index): // index is what is used to access the object. If we had a class of 10 list the index becomes important
      - return self.crayons[index]
    - def __setitem__(self, index, value):
      - self.crayons[index] = value
- cb = CrayonBox()
- cb.add(“Blue”)
- cb.add(“Red”)
- print(cb[0]) // 0 will be passed into the __getitem__(self, 0) and the value printed would be blue
- cb[0] = “Yellow” // __setitem__(self,0, “Yellow”)
- __getitem__ can be used to iterate over the items in your group

OOP: Magic Methods:index with the __getitem__ and __setitem__ Methods

- Example
  - __getitem__ can be used to iterate over the items in your group
  - for crayon in cb: // (see previous example)
    - print(crayon)

OOP: Magic Methods: The __del__ Method

- Invoked when an instance is not longer being referenced or used.
- Example
  - class Garbage():
    - def __del__(self):
      - print(“This is my last breath”)
  - g = Garbage() // Prints This is my last breath since the interpreter has reached the last line
  -
  - Example
    - g = garbage()
    - g = [1,2,3] // Prints This is my last breath since there are no more references to g
  -

OOP Inheritance:Define a subclass

- Example
  - class Store():
    - def __init__(self):
      - self.owner = “Boris”
    - def exclaim(self):
      - returns “Lots of stuff to buy, come inside!”
  - class CoffeeShop(Store): // Store is the superclass of CoffeeShop because it is in the ()
    - pass
  - starbucks = CoffeShop()
  - print(starbucks.owner) // print “Boris” which is a variable from the superclass
  - print(starbucks.exclaim) // print Lots of stuff to buy, come inside
  -

OOP Inheritance:New Methods on Subclass

- Example
  - class Employe()
    - def do_work(self):
      - print(“I am working)
  - class Manager(Employee):
    - def waste_time(self):
      - print(“Wow, this YouTube video looks fun”)
  - class Director(Manager): // Inherits from Manager which inherits from Employee
    - def fire_employee(self):
      - print(“You are fired”)
  - e = Employee()
  - m = manager()
  - e.do_work() // prints I am working
  - e.waste_time() // Get the Error AttributeError Employee object has no attribute waste_time
  - m.waste_time() // Wow, this YouTube video looks fun
  - d = Director()
  - d.do_work() // prints I am working
  - d.fire_employee() // prints You are fired

OOP Inheritance:Override an Inherited method on a subclass

- To define a different implementation on a subclass
- Example
  - class Teacher():
    - def teach_class(self):
      - print(“Teaching stuff...”)
    - def grab_lunch(self):
      - print(“Yum yum yum”)
    - def grade_tests(self):
      - print(“F! f! f!”)
  - class CollegeProfessor(Teacher):
    - def publish_book(self):
      - print(“Hooray, I’m an author”)
    - def grade_tests(self):
      - print(“A! A! A!”)

OOP Inheritance:Override an Inherited method on a subclass

- Example
  - teacher = Teacher()
  - professor = CollegeProfessor()
  - teacher.teach_class()
  - professor.teach_class()
  - professor.grade_tests() // prints A! A! A! due to method overriding. The grade_tests is defined in Proffer Class so it is used

OOP Inheritance:The Super Function

- Every single class in Python
- Example
  - class Animal():
    - def __init__(self, name):
      - self.name = name
    - def eat(self,food):
      - return f"{self.name} is enjoying the {food}"
  - class dog(Animal):
    - def __init__(self, name, breed):
      - super().__init__(name) // Calls any function from the parent class
      - Animal.__init__(name) // The Magic Method to call the parent does the same as the line above
      - self.breed)
  - watson = dog("Watson", "Golden Retriever") // The first parameter is need so the __init__ from Animal can have a value name
  - watson.eat("bacon")
  - print(watson.name) // Will print Watson
- Animal.__init__(name) problem with this line is if we change the name of the class then we need to change it here too.

OOP Inheritance:Polymorphism 1 and 2

- Different Object can use the same method name that are different internally
- Duck Typing → An object type does not matter, as much as the methods it can respond to.
- Example of polymorphism in Python → len function
  - As long as the __len__ is implemented it can calculate the length
- example
  - class Person():
    - def __init__(self, name, height):
      - self.name = name
      - self.height = height
    - def __len__(self)
      - return self.height()
  - print(len(Person(name = “Boris”, height = 71)) // For len() does not matter about type, but the len() message can be sent

OOP Inheritance:Polymorphism 1 and 2

- Example
  - import random
  - def __init__(self, games_played, victories):
    - self.games_played = games_played
    - self.victories = victories
  - @property
  - def win_ratio(self):
    - return self.victories / self.games_played
  - class HumanPlayer(Player):
    - def make_move(self):
      - print(“Let player make the decision”)
  - class ComputerPlayer(Player):
    - def make_move(self):
      - print(“Run advanced algorihtm to calculate best move!”)
  -

OOP Inheritance:Polymorphism 1 and 2

- Example
  - hp = HumanPlayer(games_played = 30, victories = 15)
  - cp = ComputerPlayer(games_played = 1000, victories = 999)
  - print(hp.win_ratio) // .5
  - print(cp.win_ratio) // .999
  - games_players = [hp,cp]
  - starting_player = random.choice(game_players) // Select an item from a list randomly
  - starting_player.make_move(); // starting_player could be either hp or cp and work each time
- When you prioritize behavior over type your program will become better designed

OOP Inheritance:Name Mangling for Privacy

- private attributes and instance methods do not exist.
  - There is no way we can create an attribute or instance method using dot syntax.
- Problem you may define a method in your subclass that they define in their superclass. This may prevent the entire subclass from functioning
- Name Mangling → Decrease the probability of name collisions between name defined in superclass and names in subclasses.
  - Still not fully private
- An attribute will be if it begins with a double underscore.
  - Only use it when is the possibility that the subclass will overwrite something and break the api
- Example
  - class Nonsense():
    - def __init__(self):
      - self.__some_attribute = “Hello” // the double underscore tells the python interpreter to mange the name
    - def __some_method():
      - print(“This is coming from some method”)
  - class SpecialNonsense(Nonsense):
    - pass

OOP Inheritance:Name Mangling for Privacy

- Example
  - n = Nonsense()
  - sn = SpecialNonsense()
  - print(n.__some_attribute) // Nonsense object has no attribute __some_attribute
  - print(sn.__some_attribute) // Nonsense object has no attribute __some_attribute
  - print(n.__some_attribute) // Nonsense object has no attribute __some_method
  - print(sn.__some_attribute) // Nonsense object has no attribute __some_method
- The interpreter will rename the function/attributes that start with __ underscore
  - Python is making hard to access them so that name collisions can be avoided.
- In order to access : print (sn._Nonsense__some_attribute)
- In order to access : print(sn._Nonsense__some_method())

OOP Inheritance:Multiple Inheritance I: Method Resource Order

- Example
  - class FrozenFood():
    - def thaw(self, minutes):
      - print(“Thawing for {minutes} minutes
    - def store(self)
      - print (“Putting in the Freezer”)
  - class Dessert():
    - def add_weight(self):
      - print(“Putting on the pounds!”)
    - def store(self):
      - print(“Putting in the refrigerator”)
  - classIceCream(FrozenFood, Dessert):
    - pass

OOP Inheritance:Multiple Inheritance I: Method Resource Order

- ic = IceCream()
- ic.add_weight()
- ic.thaw(5) // prints thawing for 5 minutes
- ic.store() // print “Putting in the freezer”
- Which store() function will be invoked.
  - Look for store() in the order of the class listed in the ()
    - class IceCreate(FrozenFood, Dessert)
    - unless the IceCream has its own store method
- print(IceCream.rmo()) // Get Back the search order of the Class which is IceCream, FrozenFood, Dessert
  - mro (M)ethod (R)esoution (O)rder
  - can invoke on the class itself.

OOP Inheritance:Multiple Inheritance : Breath First Search and Depth First Search

- Example
  - class Restaurant():
    - def make_reservations(self, party_size):
      - print(f”Booked a table {party_size}”)
  - class Steakhouse(Restaurant):
    - pass
  - class Bar():
    - def make_reservation(self, party_size):
      - print(f”Booked a lounge for {party_size}”)
  - class BarAndGrill(Steakhouse, Bar):
    - pass
  - bag = BarAndGrill()
  -

OOP Inheritance:Multiple Inheritance : Breadth First Search and Depth First Search

- bag.make_reservation(2) // Which make_reservation will Python Invoke the method in Restaurant
- problem
  - Since Steakhouse is defined first in the inheritance list will it search Restaurant next (the parent class of Steakhouse or Search Bar)
- Two ways to Search an inheritance tree (Breadth, Depth First Search)
  - Breadth Search the level first (horizontal)
  - Depth First Search each level to the terminal node is reached.
- How does Python search through the Inheritance Tree
  - Uses by default Depth First Search
  - It will prioritize look at Steak House and then the Parent Restaurant before looking at Bar
  - Search Order Bar And Grill then the first object in the Inheritance list (class BarAndGrill (Steakhouse, Bar)) then to Restaurant where it find it.
- can invoke BarAndGrill.mro and shwo the search class

OOP Inheritance:Multiple Inheritance : Diamond-Shaped Inheritance

- Example
  - class FilmMaker():
    - def give_interview(self):
      - print(“I love making movies)
  - class Director(FilmMaker):
    - pass
  - class ScreenWriter(FilmMaker):
    - def give_interview(self):
      - print(“I love writing scripts!”)
  - class JackOfAllTrades(Director, ScreenWriter) // Inheriting from two supper that inherit from the same parent class
  - stallone = JackOfAllTrades()
  - stallone.give_interview() // output I love writing scripts
- Python will still search using Depth First Algorithm
- If the same class (FilmMaker) occurrs multiple time then Python will remove all earlier occurrences of the class
  - Does this for efficiency. Will Search Jack Of All Trades, Director, Screen Writer , FilmMaker (instead of searching for it twice)
  - Moves all duplicate class to the end

OOP Inheritance:The isinstance Function and issubclass Function

- isinstance return true if the first argument is an instance of the second argument
  - first argument is instance
  - second argument is class
- example
  - print(isinstance(1,int)) // Returns True
  - print(({ “a”:1}, dict)) // Returns True
  - print([], int) // Returns False
  - print(isinstance(1, object)) // Returns True Note that sub is a parent class of int
  - print([], (list, dict)) // Returns True If the instance is create from a list or dict
- example
  - class Person():
    - pass
  - class Superhero(Person)
    - pass
  - arnold = Person()
  - boris = Superhero()
  - print(isinstance(boris, Superhero)) // Returns True
  - print(isinstance(boris, Person)) // Returns True

OOP Inheritance:The isinstance Function and issubclass Function

- Example
  - print issubclass(Superhero, Person) Returns True
  - print issubclass(Person, Superhero) Returns False

OOP Inheritance Composition

- Object delegates responsible to a nested class
  - If you believe your class has too many responsibilities then extract the new functionality to the class
- Creates separation between objects and they can be more adaptable.
  - Example Can change the brief case to contain the paper a Dictionary and the Layer Class will not care
  - less changes of errors when change classes
- Create small Lightweight Classes usually a good sign that the class has few responsibilities
- Example
  - class Paper():
    - def __init__(self, text, case):
      - self.text = text
      - self.case = case
  - def Briefcase():
    - def __init__(self,price):
      - self.price = price
      - self.papers = papers
    - def add_paper(self, paper):
      - self.papers.append(paper)
    - def view_notes(self):
      - return [paper.txt for paper in self.papers]

OOP Inheritance: Composition

- Example
  - class Lawyer():
  - def __init__(self, name, briefcase):
    - self.name = name
    - self.briefcase
  - def writeNote(self, text, case):
    - paper = Paper(test,case)
    - self.briefcase.add
  - def viewNotes(self, viewNotes):
    - print(self.briefcase.view_notes())
  - cheap_briefcase = Briefcase(price = 19.99)
  - vinny = Lawyer(name = “Vincent”, briefcase = cheap_briefcase)
  - vinny.writeNote(“My client is innocent”, “AS-2ZK1”)
  - vinny.view_notes()

Delicious Pizza Factories With @Classmethod

- Example
  - class Pizza:
  - def __init__(self, ingredients):
  - self.ingredients = ingredients
  - def __repr__(self):
  - return f'Pizza({self.ingredients!r})
  - @classmethod
  - def margherita(cls): // Called a factory function
  - return cls(['mozzarella', 'tomatoes'])
  - @classmethod
  - def prosciutto(cls):
  - return cls(['mozzarella', 'tomatoes', 'ham'])