Difference between OLTP and OLAP

====================================

| **Sr. No.** | **Key** | **OLAP** | **OLTP** |
| --- | --- | --- | --- |
| 1 | Basic | It is used for data analysis | It is used to manage very large number of online short transactions |
| 2 | Database Type | It uses data warehouse | It uses traditional DBMS |
| 3 | Data Modification | It manages all insert, update and delete transaction | It is mainly used for data reading |
| 4 | Response time | Processing is little slow | In Milliseconds |
| 5 | Normalization | Tables in OLAP database are not normalized. | Tables in OLTP database are normalized. |

OLAP stands for On-Line Analytical Processing. It is used for analysis of database information from multiple database systems at one time such as **sales** **analysis and forecasting, market research, budgeting** etc. Data Warehouse is the example of OLAP system.

OLTP stands for On-Line Transactional processing. It is used for maintaining the online transaction and record integrity in multiple access environments. OLTP is a system that manages very large number of short online transactions for example, **ATM**.

===============================================================================

The **goal of linear regression** is to create a model that predicts the value accurately and consequently has the lowest (SSE) sum of squared errors (also known as least squares). Such a model is called as the **best fit model**.

========================

**True Positives (TP)** – These are the correctly predicted positive values. It implies that the value of the actual class is yes and the value of the predicted class is also yes.

**True Negatives (TN)** – These are the correctly predicted negative values. It implies that the value of the actual class is no and the value of the predicted class is also no.

**False positives and false negatives**, these values occur when your actual class contradicts with the predicted class.

**Now,**  
**Recall,** also known as Sensitivity is the ratio of true positive rate (TP), to all observations in actual class – yes  
Recall = TP/(TP+FN)

**Precision** is the ratio of positive predictive value, which measures the amount of accurate positives model predicted viz a viz number of positives it claims.  
Precision = TP/(TP+FP)

**Accuracy** is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations.  
Accuracy = (TP+TN)/(TP+FP+FN+TN)

**F1 Score** is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. Intuitively it is not as easy to understand as accuracy, but F1 is usually more useful than accuracy, especially if you have an uneven class distribution. Accuracy works best if false positives and false negatives have a similar cost. If the cost of false positives and false negatives are very different, it’s better to look at both Precision and Recall.

=====================

To build a model in machine learning, you need to follow few steps:

1. Understand the business model
2. Data acquisitions
3. Data cleaning
4. Exploratory data analysis
5. Use machine learning algorithms to make a model
6. Use unknown dataset to check the accuracy of the model

==================================

**Collinearity** is a linear association **between** two predictors. **Multicollinearity** is a situation where two or more predictors are highly linearly related.

**Inheritance** is the capability of one class to derive or inherit the properties from another class. The benefits of inheritance are:

1. It represents real-world relationships well.
2. It provides **reusability** of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

===========================================================================

**What is Polymorphism :** The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being uses for different types.

==========================================

data points which lie on the margin lines and control the margins and consequently the position of the separating hyperplane, are known as **Support Vectors.**

**======================**

**Oops**

**Creating Behaviors**

We can create behavior in a class by adding functions in a class. However, such functions should have a special parameter called self as the first parameter.

Such functions which describe the behavior are also called as methods. We can invoke the methods using the dot operator as shown.

Even though purchase() is accepting a parameter called self, we need not pass it when we invoke it.

**class Mobile:**

**def \_\_init\_\_(self):**

**print("Inside constructor")**

**def purchase (self):**

**print("Purchasing a mobile")**

**mob1=Mobile()**

**mob1.purchase()**

**print(mob1)**

Reference variables :

* Reference variables hold the objects
* We can create objects without reference variable as well
* An object can have multiple reference variables
* Assigning a new reference variable to an existing object does not create a new object

**==========================**

**Diff Between Attribute and Method :**

Attribute == characteristics. Method == operations/ actions.

For example, Let's describe a cat (meow!).

What are the attributes(characteristics) of a cat? It has different breed, name, color, whether they have spots...etc.

What are methods (actions) of a cat? It can meow, climb, scratch you, destroy your laptop, etc.

Notice the difference, attributes define characteristics of the cat.

Methods, on the other hand, defines action/operation (verb).

Now, putting the above definition in mind, let's create an object of class 'cat'...meowww

**class Cat():**

To create attributes, use def init(self, arg1, arg2) - (as shown below).

The 'self' keyword is a reference to a particular instance of a class.

**def \_\_init\_\_(self, mybreed, name):**

**# Attributes**

**self.breed = mybreed**

**self.name = name**

**# Operations/actions --> methods**

**def kill\_mouse(self):**

**print('Insert some method to kill mouse here')**

Notice (above) 'mybreed' is an input argument that the user need to specify, whereas self.breed is an attribute of the instance assigned to 'mybreed' argument. Usually, they're the same (e.g. breed for both, self.breed = breed). Here, it's coded differently to avoid confusion.

And attributes are usually written as 'self.attribute\_name' (as shown above).

Now, methods are more like actions, or operations, where you define a function inside the body of a class to perform some operation, for example, killing a mouse. A method could also utilize the attributes that you defined within the object itself.

Another key difference between a method and attribute is how you call it.

For example, let's say we create an instance using the above class we defined.

**my\_cat = Cat()**

To call an attribute, you use

**my\_cat.name**

**or**

**my\_cat.breed**

For methods, you call it to execute some action. In Python, you call method with an open and close parenthesis, as shown below:

my\_cat.kill\_mouse()

**=======================================================**

**OOP Summary**

**--------------------------**

* OOP is a style of programming which allows us to club data and behavior together.
* This is more suited for coding real life scenarios.
* Objects are real world entities
* Class is just a classification. It is just a concept.
* Class is a description of attributes and behavior that objects of that classification should possess.
* Attributes are created in a special function called \_\_init\_\_ and behaviors are created using functions called methods.
* Objects can be created using ClassName() or using object literals for some of the built in classes
* Attributes are created using reference\_variable.attribute\_name = value syntax.
* Behavior is created by defining a function inside the class having a special parameter called self.

---------------------------------------------------------------------------------------------------------------------------

**Abstraction:**

When we invoke the purchase() on a mobile object, we don’t have to know the details of the method to invoke it.

This ability to use something without having to know the details of how it is working is called as abstraction.

**class Mobile:**

**def \_\_init\_\_(self, brand, price):**

**print("Inside constructor")**

**self.brand = brand**

**self.price = price**

**def purchase(self):**

**print("Purchasing a mobile")**

**print("This mobile has brand", self.brand, "and price", self.price)**

**print("Mobile-1")**

**mob1=Mobile("Apple", 20000)**

**mob1.purchase()**

**print("Mobile-2")**

**mob2=Mobile("Samsung",3000)**

**mob2.purchase()**

**-------------------------------------------------------------------------------------------------------------------------------**

**Encapsulation :**

**----------------------------------------**

**class Customer:**

**def \_\_init\_\_(self, cust\_id, name, age, wallet\_balance):**

**self.cust\_id = cust\_id**

**self.name = name**

**self.age = age**

**self.wallet\_balance = wallet\_balance**

**def update\_balance(self, amount):**

**if amount < 1000 and amount > 0:**

**self.wallet\_balance += amount**

**def show\_balance(self):**

**print ("The balance is ",self.wallet\_balance)**

**c1=Customer(100, "Gopal", 24, 1000)**

**c1.wallet\_balance = 10000000000**

**c1.show\_balance()**

**--------------------------------------------**

Output:

The balance is 10000000000

------------------------------------

Adding a double underscore makes the attribute a private attribute. Private attributes are those which are accessible only inside the class. This method of restricting access to our data is called **encapsulation**.

-------------------------------------

**class Customer:**

**def \_\_init\_\_(self, cust\_id, name, age, wallet\_balance):**

**self.cust\_id = cust\_id**

**self.name = name**

**self.age = age**

**self.\_\_wallet\_balance = wallet\_balance**

**def update\_balance(self, amount):**

**if amount < 1000 and amount > 0:**

**self.\_\_wallet\_balance += amount**

**def show\_balance(self):**

**print ("The balance is ",self.\_\_wallet\_balance)**

**c1=Customer(100, "Gopal", 24, 1000)**

**print(c1.\_\_wallet\_balance)**

---------------

When we put a double underscore in front of the attribute name, python will internally change its name to **\_Classname\_\_attribute.**

This is why we get an error when we try to access a private attribute.

--------------------

So change the last line in above code to :

print(c1.\_Customer\_\_wallet\_balance)

---------------------------------------------------------------------------------

Why encapsulation is used then? If we can access the private variable outside ?

-----------------------------------

Any lock can be broken by a determined thief. Similarly, just because you make your code private, does not mean it is not accessible to other developers. When a developer sees a private variable, it’s a gentleman's agreement not to access it directly. It is used to only prevent accidental access.

Thus in python encapsulation is more like a caution sign than a lock. A caution sign is there so that you don’t accidentally break a rule. But if you still want to break it you can, with consequence ;)

**Quick summary** Encapsulation**:**

**-------------------------**

Encapsulation is preventing access to a data outside the class

Adding a \_\_ in front of a attribute makes it private

In python, adding a \_\_ changes the name of the attribute to **\_Classname\_\_attribute**

**--------------------------------**

**Static variables – Introduction:**

We can create shared attributes by placing them directly inside the class and not inside the constructor. And since this attribute is not owned by any one object, we don’t need the self to create this attribute. Such variables which are created at a class level are called static variables. Here discount is a static value.

**class Mobile:**

**discount = 50**

**def \_\_init\_\_(self, price, brand):**

**self.price = price**

**self.brand = brand**

----------------------------------------------------------------------------------------------------------------------

Since static variable is object independent, we need a way to access the getter setter methods without an object. This is possible by creating static methods. Static methods are those methods which can be accessed without an object. They are accessed using the class name.

There are two rules in creating such static methods:

The methods should not have self

@staticmethod must be written on top of it

**@staticmethod**

**def get\_discount():**

**return Mobile.\_\_discount**

**@staticmethod**

**def set\_discount(discount):**

**Mobile.\_\_discount=discount**

**-------------------------------------------------------------**

Static attributes are created at class level.

Static attributes are accessed using ClassName.

Static attributes are object independent. We can access them without creating instance (object) of the class in which they are defined.

The value stored in static attribute is shared between all instances(objects) of the class in which the static attribute is defined.

----------------------------------------------------------------

Common types of OOP Relationship:

-----------------------------------------------

|  |  |  |
| --- | --- | --- |
| **Relationship** | **Description** | **Example** |
| Inheritance | When one object is a type of another object | Mobile is a Product |
| Aggregation | When one object owns another object, but they both have independent life cycle | Customer has an Address. Even if the Customer is no more, there may be other customers in that address. So Address continues to exist even after a customer is no more |
| Composition | When one object owns another object, but they both have same life cycle | College has a department. If the college closes, the department is also closed |

Also, each object may relate with multiple objects at the same time. For example, Shoe is also a Product. A Customer may have many addresses. A department may have many employees. A child may have many siblings, etc.   
In this course, we will be looking at Inheritance and Aggregation alone.

---------------------------------------------------------------------------------------------------------------------------------------

**Polymorphism:**

Sometimes a child may not want to use what it has inherited from the parent. The same holds true for OOP as well. If the child class does not want to use a method inherited from the parent class then it may create its own method with the same name.

When the child has a method with the same name as that of the parent, it is said to override the parent’s method. This is called as **Method** **Overriding**. Method overriding is also called as **Polymorphism**.

-------------------------------------------------------------------------------------------------------------------

**class Phone:**

**def \_\_init\_\_(self, price, brand, camera):**

**print ("Inside phone constructor")**

**self.\_\_price = price**

**self.brand = brand**

**self.camera = camera**

**def buy(self):**

**print ("Buying a phone")**

**def return\_phone(self):**

**print ("Returning a phone")**

**class FeaturePhone(Phone):**

**pass**

**class SmartPhone(Phone):**

**def buy(self):**

**print ("Buying a smartphone")**

**s=SmartPhone(20000, "Apple", 13)**

**s.buy()**

**super() - Invoke overridden method :**

**Even though the child class may override the methods of the parent class, it might still decide to use the parent class overridden method. To invoke anything belonging to the parent class, the child class needs to use the super() function, as shown below:**

class Phone:

def \_\_init\_\_(self, price, brand, camera):

print ("Inside phone constructor")

self.\_\_price = price

self.brand = brand

self.camera = camera

def buy(self):

print ("Buying a phone")

def return\_phone(self):

print ("Returning a phone")

class FeaturePhone(Phone):

pass

class SmartPhone(Phone):

def buy(self):

print ("Buying a smartphone")

super().buy()

s=SmartPhone(20000, "Apple", 13)

s.buy()

**O/P :**

**Inside phone constructor**

**Buying a smartphone**

**Buying a phone**

To access the parent class constructor we can use super(). Thus, the data is passed to the child class constructor, from there the data is sent to the parent class constructor and thus the attributes of the parent class get inherited.

super() function can be used to access the constructor or methods of the parent class, but not the attributes. Also super() function can be used only inside a class and not outside it

**----------------------------**

**class Phone:**

**def \_\_init\_\_(self, price, brand, camera):**

**print ("Inside phone constructor")**

**self.\_\_price = price**

**self.brand = brand**

**self.camera = camera**

**def buy(self):**

**print ("Buying a phone")**

**def return\_phone(self):**

**print ("Returning a phone")**

**class FeaturePhone(Phone):**

**pass**

**class SmartPhone(Phone):**

**def \_\_init\_\_(self, price, brand, camera, os, ram):**

**super().\_\_init\_\_(price, brand, camera)**

**self.os = os**

**self.ram = ram**

**print ("Inside smartphone constructor")**

**def buy(self):**

**print ("Buying a smartphone")**

**s=SmartPhone(20000, "Samsung", 12, "Android", 2)**

**print(s.os)**

**print(s.brand)**

**---------------- O/P : -----------------------**

Inside phone constructor

Inside smartphone constructor

Android

Samsung

-----------------------------------------------------------------------------------------------

**Summary of Inheritance:**

**--------------------------------------------**

A class can inherit from another class.

Inheritance improves code reuse

Constructor, attributes, methods get inherited to the child class

The parent has no access to the child class

Private properties of parent are not accessible directly in child class

Child class can override the attributes or methods. This is called method overriding

super() is an inbuilt function which is used to invoke the parent class methods and constructor

--------------------------------------------------













