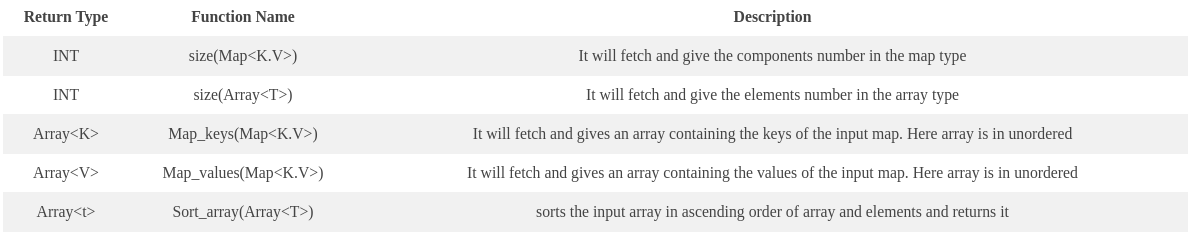
# Introduction to Hive Built-In Functions

In Hive, there are some built-in functions available. Basically, to use Hive built-in functions in our applications first we need to check the application requirement. However, it is possible to call these functions directly in our application.  
In addition, there are several types of Hive Built-in Functions available.

Now, let’s describe Hive Built-in Functions in detail:

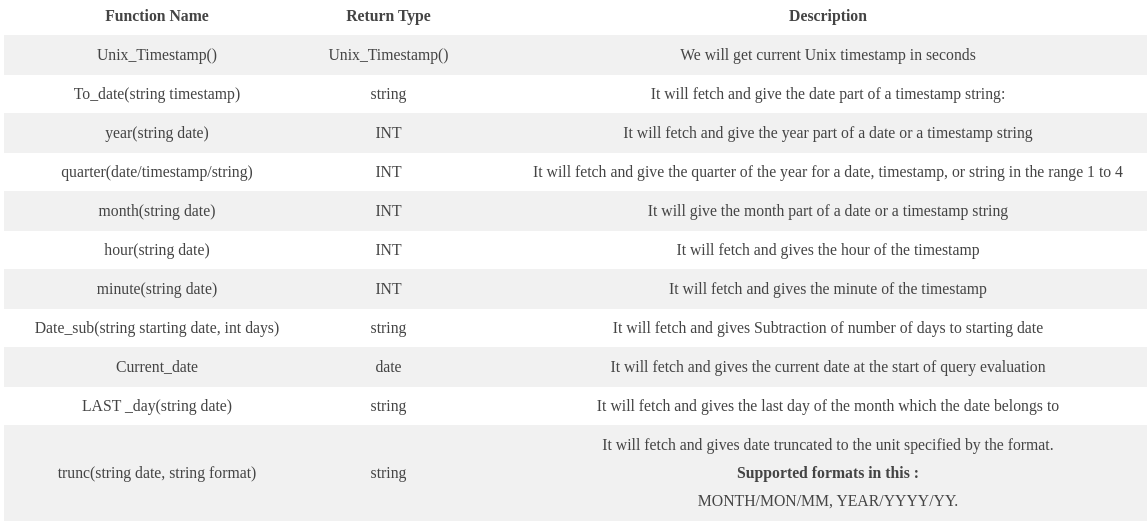
### Collection Functions

Basically, as par its name we use “Collection Functions” for collections. Here, collections are nothing but defined as a grouping of elements and returning single or array of elements depends on return type mentioned in the function name. Let’s discuss each Collection Functions- Hive Built-in functions in below table:



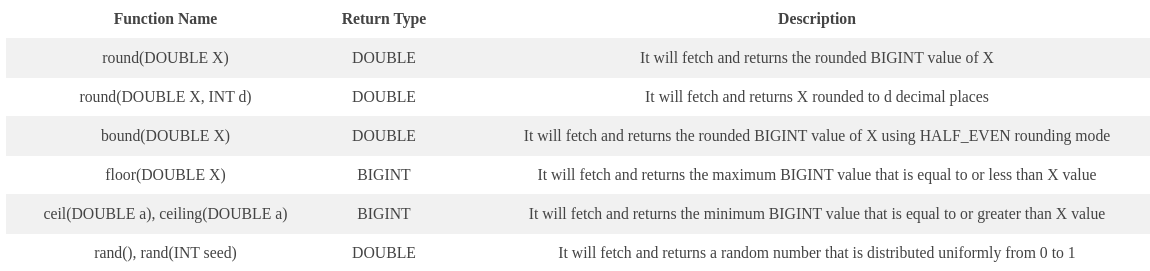
### Hive Date Functions

However, to perform date manipulations and conversion of date types from one type to another type we use Hive date Functions. Let’s discuss each Date Function – Hive Built-in Functions in below table:



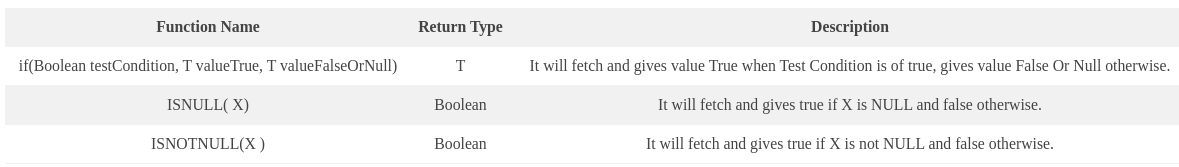
### Mathematical Functions

For Mathematical Operations in Hive, we use “Mathematical Functions”. Though, we have some inbuilt mathematical functions in Hive despite creating UDFs. Let’s discuss each Mathematical Function- Hive Built-in Functions in below table



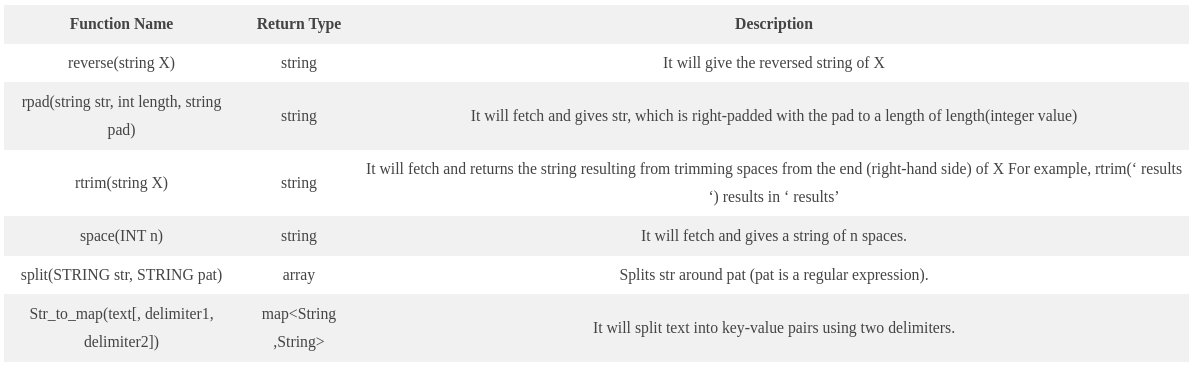
### Conditional Functions

While it comes to conditional values checks in Hive, we use “Conditional Functions”. Let’s discuss each Conditional Function- Hive Built-In Functions in below table:



### Hive String Functions

For String manipulations and string operations in Hive, we call Hive String Functions. Let’s discuss each String Function- Hive Built-In Functions in below table



# Hive UDF – User Defined Function with Example

## What is Hive UDF?

Basically, we can use two different interfaces for writing Apache Hive User Defined Functions.

1. Simple API
2. Complex API

As long as our function reads and returns primitive types, we can use the simple API (org.apache.hadoop.hive.ql.exec.UDF). In other words, it means basic **[Hadoop](https://data-flair.training/blogs/hadoop-features-and-design-principles/)** & Hive writable types. Such as Text, IntWritable, LongWritable, DoubleWritable, etc.  
Before we proceed, let’s discuss **[Hive Features & Limitations](https://data-flair.training/blogs/hive-features-and-limitations/)** in detail.  
So, let’s discuss each Hive UDF API in detail:

### a. Simple API

Basically, with the simpler UDF API, building a Hive User Defined Function involves little more than writing a class with one function (evaluate). However, let’s see an example to understand it well:  
****Simple API – Hive UDF Example****

class SimpleUDFExample extends UDF

{

public Text **evaluate**(Text input)

{

return new **Text**("Hello " + input.**toString**());

}

}

#### ****i. TESTING SIMPLE Hive UDF****

Moreover, we can test it with regular testing tools, like JUnit, since the Hive UDF is simple one function.

public class SimpleUDFExampleTest

{

@Test

public void **testUDF**()

{

SimpleUDFExample example = new **SimpleUDFExample**();

Assert.**assertEquals**("Hello world", example.**evaluate**(new **Text**("world")).**toString**());

}

}

### b. Complex API

However, to write code for objects that are not writable types. Like struct, map and array types. Hence the org.apache.hadoop.hive.ql.udf.generic. GenericUDF API offers a way.  
In addition, for the function arguments, it needs us to manually manage object inspectors. Also, to verify the number and types of the arguments we receive. To be more specific, an object inspector offers a consistent interface for underlying object types. Hence, that different object implementation can all be accessed in a consistent way from within hive. For example, we could implement a struct as a Map so long as you provide a corresponding object inspector.  
Read about **[Apache Hive Built-in functions](https://data-flair.training/blogs/hive-built-in-functions/)** in detail.  
Moreover, with this API we need to implement three methods:

// this is like the evaluate method of the simple API. It takes the actual arguments and returns the result

abstract Object **evaluate**(GenericUDF.DeferredObject[] arguments);

// Doesn't really matter, we can return anything but should be a string representation of the function.

abstract String **getDisplayString**(String[] children);

// called once, before any evaluate() calls. You receive an array of object inspectors that represent the arguments of the function

// this is where you validate that the function is receiving the correct argument types and the correct number of arguments.

abstract ObjectInspector **initialize**(ObjectInspector[] arguments);

To understand this properly,  let’s take an example.  
****Complex API – Apache Hive UDF Example****  
Basically, here the creation of a function called containsString. However, it takes two arguments:

1. A list of Strings:
2. A String

Further, it returns true/false on whether the list contains the string that we offer, for example:  
Let’s learn**[Apache Hive Operators](https://data-flair.training/blogs/apache-hive-operators/)**in detail.  
containsString(List(“a”, “b”, “c”), “b”); // true  
containsString(List(“a”, “b”, “c”), “d”); // false

#### i. GenericUDF API

However, the GenericUDF API needs a little more boilerplate, unlike with Hive UDF API:

class ComplexUDFExample extends GenericUDF

{

ListObjectInspector listOI;

StringObjectInspector elementOI;

@Override

public String **getDisplayString**(String[] arg0)

{

return "arrayContainsExample()"; // this should probably be better

}

@Override

public ObjectInspector **initialize**(ObjectInspector[] arguments) throws UDFArgumentException

{

**if** (arguments.length != 2)

{

throw new **UDFArgumentLengthException**("arrayContainsExample only takes 2 arguments: List<T>, T");

}

// 1. Check we received the right object types.

ObjectInspector a = arguments[0];

ObjectInspector b = arguments[1];

**if** (!(a instanceof ListObjectInspector) || !(b instanceof StringObjectInspector))

{

throw new **UDFArgumentException**("first argument must be a list / array, second argument must be a string");

}

this.listOI = (ListObjectInspector) a;

this.elementOI = (StringObjectInspector) b;

// 2. Check that the list contains strings

**if**(!(listOI.**getListElementObjectInspector**() instanceof StringObjectInspector))

{

throw new **UDFArgumentException**("first argument must be a list of strings");

}

// the return type of our function is a boolean, so we provide the correct object inspector

return PrimitiveObjectInspectorFactory.javaBooleanObjectInspector;

}

@Override

public Object **evaluate**(DeferredObject[] arguments) throws HiveException

{

// get the list and string from the deferred objects using the object inspectors

List<String> list = (List<String>) this.listOI.**getList**(arguments[0].**get**());

String arg = elementOI.**getPrimitiveJavaObject**(arguments[1].**get**());

// check for nulls

**if** (list == null || arg == null)

{

return null;

}

// see if our list contains the value we need

**for**(String s: list)

{

**if** (arg.**equals**(s)) return new **Boolean**(true);

}

return new **Boolean**(false);

}

}