Java Pair RDD

sortByKey (asc = true)

* Sort the RDD by key so that each partition contains sorted range of elements

ex : Val products = Assume K, V, K is price
V is product record

Val sorted by Price = products.sortByKey(); ASC
products.sortByKey(false) DESC

RDD Operations

Transformations - return RDD
- lazy evaluation

Actions - returns result to deliver
- immediately cause the previous op (transformations) to be executed.

RDD persistence:

- By default, each transformed RDD will be evaluated everytime an action gets subsequently executed
- we can persists an RDD in memory using persist() or cache() method in which case Spark will keep the RDD in memory around the cluster for much faster access the next time performs operations on this RDD.
- Caching is a key tool for iterative algorithms
- The cache() method allows only default storage level (mem-only)
- The persists() method allows to specify storage level

MEMORY_ONLY - Stored as serialized Java objects in JVM (Default)

MEMORY_AND_DISK - In addition to MEMORY_ONLY, if an RDD does not fit in available cluster memory, the RDD partitions that don't fit are stored on Disk.

MEMORY_ONLY_SER - Stored as serialized Scala obj but as one array of bytes (one array per partition)

MEMORY_AND_DISK_SER - Spills partitions that don't fit onto disk

DISK_ONLY - Stored on disk. (not in mem)

MEMORY_ONLY_2 - Keep 2 copies of each partition across nodes

MEMORY_AND_DISK_2 - Spill onto disk for partitions that don't fit

SPARK PROGRAM EXECUTION (SCALA)

Command Line (Interactive)

> spark-shell

Params

--master MASTER_URL	"local", "yarn", spark://host:port
--class CLASS_NAME	Application main class
--name "Name"	Application name
--jars JARS	CSV list of local jars to include in driver & executor class path.
--py-files PYFILES	CSV list of zip, egg or py files to place on PYTHON PATH for python apps
--files FILES	CSV list of files to be placed in working directories of each executor.
--conf PROP=VALUE	Arbitrary Spark-config property
--properties-file FILE	path to file from which to load extra properties. By default we /etc/conf/spark-defaul-

YARN only parameters

--archives ARCHIVES	CSV list of archives to be extracted into working directory of each executor.

Spark-Shell in interactive mode

* we don't specify --class & jar file.
* we can type in scala statements for spark or in general on the scala prompt

BATCH Mode

SBT (Scala Build Tool)

Common Steps

* Create Directory Structure

```
./
./simple.sbt
./src
./src/main/scala/yourprogram.scala
```

Simple.sbt

```
name := "Project Name"
version := "1.0"
scalaVersion := "2.10.4"
libraryDependencies += "org.apache.spark" %% "spark-co
                       %% "1.2.1"
```

YourProgram.scala

```
import org.apache.spark{SparkContext, SparkConf}
import org.apache.spark.SparkContext._
import org.apache.spark.rdd._
object YourProgram {
    def main (args: Array[String]) {
        yourStatement... val conf = new SparkConf()
                                .setAppName(" ")
    }
}
        val sc = new SparkContext(conf)
```

// Package the program into Jar
go to folder that contains simple.sbt

> sbt package → This will create a target folder

Run application using Spark-submit

> spark-submit --class "Your main class"
--master "local[4]" ← local mode with 4 threads
target/scala/xxx.jar
{location of your jar}

Batch Mode using Spark-shell

> spark-shell --class "Your main class"
--jars "path to jar"

* SPARK API (PySpark) *

Pyspark is a python API for spark.

Important classes & methods

* SparkContext

* Constructor : SparkContext (master = None, appName = None,
pyFiles = None, environment = None ----)

* accumulator (value, accum_param = None)
↓ helper object on how to add values

* addFile (path) - add files that gets sent with this job to every node where this job tasks run.

* broadcast (value) - broadcast read-only variable to the cluster

* defaultParallelism - default num of partitions for reduce tasks

* newAPIHadoopFile (path, inputFormatClass, keyClass, valueClass,
keyConverter = None, valueConverter = None,
conf = None)
Can provide lambda func here

keyClass = full qualified eg: org.apache.hadoop.io.Text
valueClass = full qualified eg: org.apache.hadoop.io.LongWritable
conf = Hadoop configuration (None by default)
inputFormatClass = org.apache.hadoop.mapreduce.lib.input.
TextInputFormat

parallelize (collection, numSlices = None)

→ distributes a local python collection to form an RDD.

sequenceFile (path, keyClass = None, valueClass = None,
keyConverter = None, valueConverter = None, minSplits = None

→ Reads a sequence file from HDFS or LFS and returns RDD of K, V

Stop() : shuts down the SparkContext

textFile (name, numPartitions = None, use-unicode = True)

→ reads a text file from HDFS or LFS and returns RDD where each record is a line from file.

→ if unicode = False, strings will be kept as str (UTF8) which is faster.

union (rdds) - build the union of a list of RDDs

Version - Spark Version

wholeTextFiles (path, minPartitions = None, use-unicode = True)

- reads a directory & returns RDD of K, V where K is filename, V = content of single file.

* lambda functions in python only support single statements

* RDD *

* aggregate (zeroValue, combinerOp, ReducerOp)
  - allows providing zero value
  - allows different logic for combiner & reducer
  - allows different data types between input records of RDD & output result
  - used for overall aggregations (max, min, sum, avg, count)

aggregateByKey (zeroVal, combineOp, ReducerOp), numPartitions = None)
  - per key aggregations. (rest are similar to aggregate() above)

eg: revenueAndOrderCountPerDay =

ordersByDate.aggregateByKey((0.0, 0), lambda acc, val: (acc[0] + value, acc[1] + 1), lambda acc, value: (acc[0] + val[0], acc[1] + val[1]))

- initial value is tuple
- combineOp generates tuple of (sum, count)
- ReducerOp adds up sums & counts from each tuple.

cache() - persists RDD with default storage level (MEMORY_ONLY_SER)

cartesian(Other) - returns cartesian product of this RDD with other

coalesce(numPartitions) - returns RDD that is reduced to numPartitions

cogroup(other, numPartitions = None)
  - for each key in this & other, return RDD that contains tuple with list of values for that key in self as well as other.

* (k, (value from this, value from other)) *

* collect() - returns a list that contains all elements in this RDD

collectAsMap() - returns a K, V RDD into a dictionary.

* combineByKey(zeroValFunc, combineOp, ReducerOp, numPartition = None)
  - Allows specifying a function that gets passed in the first record (from each partition) to generate zero value.
  - rest of this are same as aggregateByKey()

eg:

revAndOrderCountPerDay =

ordersByDate.combineByKey(lambda t: (t, 1), ← the return value of this function forms to Accumulator

lambda acc, val: (acc[0] + val, acc[1] + 1)

lambda acc, val: (acc[0] + val[0], acc[1] + val[1])

Context - the Spark context that this RDD was created on.

* count() - return number of elements in the RDD.

* countByKey(): returns number of unique values per key as a dictionary of (key, count(V))

eg: numItemsPerOrder = ordersKV.countByKey()

* distinct(numPartitions = None)
  - returns RDD containing distinct elements in this RDD
  - if the elements are tuples then each different tuple (ie any field different) is considered unique.

eg: cancelledOrders = orders.filter(lambda x: x[3].find("CANCELE... != -1)

* filter(Op) - return new RDD containing elements that match the Op. (ie Op returns true)
  - always filter as early as possible before any transformations

first() - returns first element in this RDD

* flatMap(Op, preservesPartitioning = False)
  - returns an RDD by first applying the function & flattening the results.

eg:

```
wordCount = book.flatMap
        (lambda line: line.split(" "))
        .map(lambda w: (w,1))
        .reduceByKey(lambda x,y: x+y)
```

flatMapValues(Op)
- passes the value of each key to the function which should return a collection which is then flattened with the key.
- usually used for sorting & ranking after groupByKey() or groupBy()

eg: productsCatGroup = products.groupBy(lambda x: x[1])

```
top3PriceByCat = productsCatGroup
        .flatMapValues
        (lambda x: sorted(x, key = lambda i: i[
                reverse = True)
        [0:3]
```

sorting using this function

These are all products (list) for a category

4 items in reverse order

slice to take first 3 elements

fold (zeroVal, op)

- aggregate elements of each partition & results of all partitions given the associative function
- Similar to reduce() but can specify a zero value
- used for overall aggregations.

~~foreach(op)~~

foldByKey (zeroVal, func)

- similar to fold but operates on list of values per key.

foreach (func) - applies a function to all elements of RDD
- its an Action which does not return anything
- used mainly for side effects (printing on screen)

eg: products.foreach(lambda x : print(x))

getNumPartitions() - gets the ~~number of partitions for this RDD~~ ~~device of the file to which this RDD~~

getStorageLevel()

glom() - returns RDD created by coalescing all elements within each partition into a list.

* groupBy (func, numPartitions = None)
  - returns a RDD with K, V where K is the value returned by the function & value is the element of the RDD.

eg: productsCatGrp = products.groupBy(lambda x : x[1])

Group by 2nd element of tuple resulting is (key, (tuple))

* groupByKey (numPartitions = None)
  - groups a K, V RDD to return an RDD of (K, List(V))

eg:
productsCatGrp = products.map(lambda x : (x[1], x)).groupByKey();

groupWith (other, *others)
  - cogroup with multiple RDDs

id() - unique id for this RDD.

keyBy (func)

- generates K, V RDD ~~from~~ for each element in source RDD by applying the function

eg:

```
ordersKV = ordersRDD.keyBy(lambda x : x[0])
```

↑ generate (k, v) from v where k is orderid (x[0])

~~map~~ keys() - return RDD with the keys of a this K, V RDD

lookup (key) - returns list of values in the RDD for the given key.

map (func, preservesPartitioning = False)

- transforms element V of this RDD into V'.
- returns an RDD(V')
- This is PUSH model (elements are pushed one by one into f)

// Transforms Orders Tuple to OrdersKV while removing x[4]

ordersKV = orders.map(lambda x: (x[0], (x[1], x[2], x[3]))

mapPartitions (func, preservesPartitioning = False)

- the function func is applied to each partition
- The elements of each partition are accessible to the func as iterator.
- This is PULL model.
- useful if we want to execute piece of code only once per partition. (map() will execute the func per element).

// Transforms Orders Tuple to OrdersKV

Sum of orderItemSubtotals per order

mapPartitionsWithIndex (index, func, preservesPartitioning = False)

- Similar to mapPartitions() but the index of original partition is also available to func.

mapValues(func)

- passes the V of a K,V RDD to the func
- The func can return single value (in which case the result RDD will be flat structure (K,V)
  but it can also return collection (in which case the resulting RDD will be nested structure (K, list(V))

eg: someRDD.mapValues(lambda v : v * v) v is single value
Input - Flat Output - Flat V is single value (v*v)
transform

someRDD.mapValues(lambda v : [v, v]) v is single value
Input - Flat Output - Nested V is list of values
transform/explode

someRDD.mapValues(lambda v : sum(v)) v is iterator (collection)
Input - Nested Output - Flat V is single value
aggregation

someRDD.mapValues(lambda v : for x in v : x+x) v is iterator
Input - Nested Output - Nested V is list.
Transform.

max(key = None)

- Find max item in this RDD according to the function

eg: highestProductPrice = products.max(lambda x : x[4])

min(key = None)

sum()

persist(StorageLevel)

- sets this RDD storage level.

~~perfects~~

* reduce (func)

- overall aggregation
- no zero value can be specified
- the func serves as both combine & reduce op.

* reduceByKey (func, numPartitions = None)

- similar to reduce() but does per key aggregation
- values for key are grouped together & the values are passed one by one into the func.

eg:

```
wordcount = words.map(lambda x : (x, 1))
            .reduceByKey(lambda acc, val : acc + val)
```

repartition (numPartitions)

- returns new RDD that has exactly numPartitions

setName (name)

- Assign a name to this RDD

* sortBy(func, asc = True, numPartitions = None)
  - sorts this RDD according to the given func
  - used for global sorting. ~~& ranking.~~

eg: // sort by product price
priceAsc = products.sortBy(lambda x : x[4])
priceDesc = products.sortBy(lambda x : x[4], False)

* sortByKey(asc = True, numPartitions = None)
  - similar to sortBy but requires K, V & sorts by key.
  - used for global sorting.

priceAsc = products.map(lambda x : (x[4], x))
.sortByKey()
priceDesc = products.map(lambda x : (x[4], x))
.sortByKey(false)

* takeOrdered(num, key = None)
  - used for global ranking.
  - sorts the elements in Asc and picks only top "num"
  - can specify a sort key function.

// get the 3 least priced products
bottom5price = products.takeOrdered(3, key = lambda x : x[4])
OR using K, V
bottom5price = products.map(lambda x : (x[4], x))
.takeOrdered(3)

top(num, key = None)
  - works exactly opposite of takeOrdered. default DESC & picks "num"
  - used for global ranking.

top5price = products.top(3, lambda x : x[4])
top5price = products.map(lambda x : (x[4], x)).top(3)

* take (num)
- takes first num elements in RDD.

unpersist ()
- mark RDD as non-persistent

values () - returns an RDD of V from K, V RDD

union (other) - return union of this RDD with other

intersection (other) - return intersection of this RDD with other

* join (other, numPartitions = None)
- returns RDD contains pair of matching elements (ie elements with matching keys.
- this & other must be a K, V RDD.
- returned RDD as (k, (v1, v2)) tuple

```
ordersKV = orders.map(lambda x : (x[0], x))
orderItemsKV = orderItems.map(lambda x : (x[1], x))

ordersJoin = orderKV.join(orderItemsKV)
```

leftOuterJoin (other, numPartition = None)

fullOuterJoin (other, numPartition = None)

rightOuterJoin (other, numPartition = None)

saveAsNewAPIHadoopFile (~~conf~~ ~~path~~, keyConverter = None
valueConverter = None)

saveAsNewAPIHadoopFile (path, outputFormatClass, keyClass = None
valueClass = None, keyConverter = None
valueConverter = None)

outputFormatClass = org.apache.hadoop.mapreduce.lib.output.
SequenceFileOutputFormat

keyClass = org.apache.hadoop.io.Text

valueClass = org.apache.hadoop.io.IntWritable.

saveAsSequenceFile (path, compressionCodecClass = None)

saveAsTextFile (path)

SparkConf

contains(key)

get(key, defaultValue = None)

getAll() - returns list of K, V pairs config settings

* setAppName(name) - set App Name
* setMaster(value) - set master URL

PACKAGE pyspark.sql

Class StructField(name, datatype, nullable = True)

* class SQLContext(sparkContext, sqlContext = None)
  - main entry point for Spark sql functionality
  - SchemaRDD - RDD with schema (Row objects)

applySchema(rdd, schema)
→ applies the given schema to an RDD.

```
schema = StructType([StructField("field1", IntegerType(), False),
                     StructField("field2", StringType(), False)])

rdd = sc.parallelize([(1, "One"), (2, "Two")])

SchemaRDD = sqlContext.applySchema(rdd, schema)
sqlContext.registerRDDAsTable(schemaRDD, "table1")
```

This is method on sqlContext to create table

```
sqlContext.sql("Select * from table1").collect()
```

cacheTable (tableName) - cache table in mem.

inferSchema (rdd)

- infer & apply schema to an RDD of Row

```
OrdersRDD = orders.map(lambda x :
                Row(order_id = int(x[0]),
                    order_date = x[1],
                    order_cust_id = int(x[2]),
                    order_status = x[3]))
ordersSchemaRDD = ~~ordersRDD~~ sqlContext.inferSchema(ordersRDD)
ordersSchemaRDD.registerTempTable("orders")
sqlContext.sql("select * from orders")
```

This method is on SchemaRDD to create a table

* jsonFile (path, schema = None)

```
eg:  deptJSON = sqlcontext.jsonFile(path)
     deptJSON.registerTempTable("deps")

depRDD = sqlContext.sql("select * from deps")
```

parquetFile (path)

- loads a parquet file & return result as SchemaRDD

registerRDDAsTable (rdd, tablename)

- registers an RDD as temp table in catalog

sql (sqlquery)

- returns an SchemaRDD representing results of the given query.

* SQL Context is used to define a structure on top of data in HDFS
It is not Hive aware

* Hive Context knows the structure from hive metastore

* Class SchemaRDD

cache(), count(), distinct(), intersection(), limit(), persist(), take(), saveAsParquetFile().

registerTempTable(name)
- registers the RDD as temporary table with given name

limit(num)
- limits the result to specified number

toJSON()
- Convert schemaRDD to MappedRDD of JSON documents

* Row - used to create SchemaRDD.

Constructor

Row(name = "Alice", age = 21)

Class HiveContext

- Variant of SQLContext that is aware of Hive metastore
- ie query Hive tables & execute hive commands like show table, describe formatted etc.
- Also can set hive properties.

eg: hiveContext.sql("set hive.execution.engine = mr")

hiveContext.sql("use retail_db")
hiveContext.sql("select * from departments")

Spark Execution Python

① Interactive Mode - directly type python statements at prompt

```
>pyspark .
```

--master	MASTER_URL	"yarn", "local", spark://host:port
--name	AppName	
--py-files	PY_FILES	CSV list of eg: zip or .py files to place on PYTHONPATH
--files	FILES	CSV list of files to be placed in working directory of each executor
--conf	prop=Value	
--properties-file	FILE	path of file to load extra properties default: /etc/spark/conf/spark-defaults.conf
--version		

② Batch mode - create python script & submit for execution

SimpleApp.py

```
if __name__ == "__main__":
    def myfunc(s):
        words = s.split(" ")
        return len(words)

    sc = SparkContext(...)
    sc.textFile("file.txt").map(myfunc)
```

```
> spark-submit --master local[4] SimpleApp.py .
```

Spark Configuration

Spark.app.name

Spark.master

Spark.driver.extraClassPath

spark.driver.extraLibraryPath

Spark.executor.extraClasspath

Spark.executor.extraLibraryPath

Spark.shuffle.compress - true - Wether to compress map outpu

Spark.ui.port 4040 Port for Application Dashboard

Spark.rdd.compress false Whether to compress serialized partitions

Spark.io.compression.codec Snappy

Spark.default.parallelism

Spark.task.maxFailures 4

Spark.speculation False Speculative execution

SPARK SQL

Spark.sql.shuffle.partitions 200 Num partitions when shuffling data.

* Types are always implicitly determined
* Everything is obj.
* Pass by reference default

SCALA FOR SPARK

Important class & functions.

i : Int = 1

Val i = 1 OR Val ~~i [Int]~~ = 1

Var str = " " OR Var str : String = "ABC"

Val - immutable

Var - mutable

// Formatted

* printf ("%d %.2f", i, j, k)

// Unformatted

* println (i, j, k)

// Arrays are fixed size but mutable (ie values within the array can be changed)

```
var myArr = Array (1, 2, 3, 4, 5)
// Access
myArr (2)
```

// List - size can change (add/remove elements) but cannot change existing value

```
var myList1 = List (1, 2, 3, 4, 5)
myList1 (2) = 5 // Invalid
myList1 = List (7, 8, 9) // Valid - changing object ref.
```

// Mutable List

```
val myList2 = Collection.mutable.MutableList (1, 2, 3, 4)
myList2 (3) = 100 // Valid
myList2 = (7, 8, 9) // Invalid since val type
```

* for loops

```
for (i ← myList1) {
    println (i)
}
```

// excludes last number

* for (i ← 0 until str.length()) {
    print (str(i) + " ")
  }

* for (i ← 0 to 5) { // 0 to 5 inclusive
    print(i)
  }

// Yield -

* Val evenList = for { i ← 1 to 20 if i % 2 == 0 } yield i ;
  // gets all even values between 1 & 20.

// Printing Simple String or Expressions

* println (s "Hello $name") // Where Val name = "Derek"

// Printing with Formatted String or Expressions

~~println (f$ "I am $age~~

println (f "I am ${age + 1} and weigh $weight %.2f")

printf ("I am %d and weight %.2f", age, weight)

Strings : same as Java - fancy. string

Val str = " Something important "

* str.length
* str.concat (" for me")

* str.equals ("test other string")
* str.contains (" something")
* str.startsWith ( )
* str.endsWith ( )
* String. format ("%d, %f", age, name)

str.indexOf (

* str.replace (old, new)
* str.split (", ")
* str.substring (beginIndex., endIndex)
* str.trim ()

Arrays :

Val friends = Array ("Bob", "Tom")

// Access friends(0)

val friends2 = ArrayBuffer[String] (); ArrayBuffer can vary in size

friends2.insert (0, "Mark") // insert

friends2 += "Phil" // append

friends2 ++= Array("Susy", "Ben") // Append an array

friends2.insert (1, "Mike", "Sally", "Sam"); // insert multiple elements at 1

friends2.remove (2, 3) // remove elements from 2 to 3

friends2.foreach (println) // print

// Transform.

val friendsF3 = for (f <- friends2) yield f.toUpper();

// Functions available on Arrays

val nums = Array(1, 2, 3, 4, 5)

* nums.max
* nums.min
* nums.sum

// Sorting

* nums.sortWith(_ > _) // Desc
  nums.sortWith(_ < _) // Asc

// to string in CSV

* nums.mkString(", ")

// Mixed Arrays (Any type)

val arr = Array("a", 1, true)

array.find{x => x == 4}.getOrElse(10);
// return 4 is found else 10.

List Immutable (ie all operations return a new List object)

```
val list1 = List(1, 2, 3, 4)
val list2 = List(7, 8, 9, 10)

* list1 ++ list2          // concatenate

// declare Empty list  var list1 : List[Int] = List()
var list1 = List[Int]();   // define list of Integers.
* list1 ::= 5             // append 5
* list1 = 6 :: list1      // append 6
```

returns new list object which is assigned to list1

```
list1.drop(2)   // drop 2 elements from front
list.init       // retrieve everything except last
* list.head     // first element
* list.last     // last element
* list.slice(2, 5)   // retrieves from 2 (inclusive) and 5 (exclusive)
list.splitAt(4)     // split at index 4.
* list.take(5)      // take first 5 elements

* list.contains(6)          // True if 6 is present
* list.startsWith(Array(1, 2))
* list.endsWith(Array(6, 7, 8))
* list.isEmpty              // whether empty
list.indexOf(5)             // index of element 5
list.indexOf(5, 2)          // index of element 5 from index 2
* list.distinct             // returns list of unique values
* list.reverse              // reverse the list.

* list.min
* list.max
* list.sum
```

* list.map{x => x.length} // Transform
* list.filter{x => x%2==0} // filter returns even number list
* list.count{x => x%2==0} // count of even numbers in list

* list.exists(x => x%3==0) // whether multiples of 3 are present

* list.flatMap(x => List.fill(x)(x)) // flattens the nested collection

* list.foreach(x => print(x + ":"))

list.sortWith(f) → use f for sorting
eg: list.sortWith(_ > _) //Desc.

Apache SPARK

> spark-shell (Scala command line)
> pyspark (python CLI)

$Spark Context
Sql Context - Using sql to create, load & store tables
Hive Context - Use hive metastore to get Hive data.
- Directly run Hive statements
- Instead of running MR Hive queries will be translated to Spark code.
- Copy or create Softlink to /etc/hive/conf/hive-site.xml in /etc/spark/conf directory

> Spark-sql - Allows hive queries directly to be executed from CLI
- Similar to Hive CLI
- Does not require Scala or Python knowledge

To connect to RDBMS using JDBC syntax of spark
> spark-shell --driver-class-path /usr/share/--- /mysql-connector.jar
import org.apache.spark.sql.SQLContext
sqlContext.load("jdbc", Map(
"url" → url,
"dbtable" → "departments"))
.collect().foreach(println);

→ Spark can run in local mode with being aware of Hadoop

Spark 1.2.1 does work properly with YARN
to start Spark 1.2.1 use:
> spark-shell --master local

Scala App
① Write scala file (Spark code)
② Generate jar file using SBT (Scala build tool)
③ use spark-submit command to submit Spark job to cluster

* Sudo yum -y install sbt
* Env Variable: ~/.bash_profile

Important classes -

org.apache.spark.SparkContext
sql context
Hive Context

org.apache.spark.executor

Need to also know

Hadoop filesystem API
hadoop.io - package

Name spaces

* org.apache.spark [SC, S]
* org.apache.spark.rdd [RDDs]

org.apache.spark.io
↳ compression

org.apache.spark.sql
↳ SQL context

org.apache.spark.sql.hive
↳ Hive Context

SPARK

RDD :

Driver Program : runs the main() function and executes various parallel operations on a cluster

(Resilient Distributed Datasets)

RDD : main abstraction in Spark.

→ its a collection of elements partitioned across nodes of a cluster that can be operated on parallel.

→ RDDs automatically recover from node failures

Shared Variables : → can be used by parallel operations

→ By default Spark function runs in parallel across nodes it ships a copy of each variable used in the function to each task.

→ Sometimes we need to share variables across tasks. & hence the shared variables are used.

→ Two types

① Broadcast variables : used to cache a value in memory of all nodes.

→ created by calling broadcast function on spark context

val bcVar = sc.broadcast (Array (1, 2, 3))

→ After creation it should be used instead of the object passed in to the broadcast function.

② Accumulators : Variables that are only added to such as counters or sums.

→ Created with initial value (v) by using function

val accum = sc.accumulator (v);

→ Tasks running on the cluster can earlier add() method or += operator

Important Spark API

SparkConf : Configuration for Spark application
: created using new SparkConf()

get() : get value for a configuration key

getAll() : get all configs settings as K, V

getDouble(), getLong() - type getter methods for configurations

set(String key, String value) : SparkConf
- set config parameter

setAppName(String) - name of our spark application

setJars(Array[String]) - set jar files to distribute to the cluster.

setMaster(String) : Master URL to connect to.
ex: "local" to run locally in one thread
"local[4]" to run locally in 4 cores.
"yarn" to run on YARN

SparkContext :

→ Main entry point for Spark functionality
→ Represents the connection to a spark cluster
→ Used to create RDD, Accumulators & Broadcast variables
→ Only one SparkContext should be active in a single JVM.

* new SparkContext (appName, SparkConf)

* addFile (path) - add file to be downloaded with this Spark Job on every node.

* addJar (path) - adds a dependency JAR for all tasks to be executed on this context in the future

* appName

* broadcast (value: T) — create broadcast variables

* getConf

* newAPIHadoopFile(path, class[InputFormat], class[K], class[Value])

ex: import org.apache.hadoop.mapreduce.lib.input._
val depseq = sc.newAPIHadoopFile[IntWritable, Text, SequenceFileInputFormat[IntWritable, Text]]
(outputpath, classOf[SequenceFileInputFormat[IntWritable, Text]], classOf[IntWritable], classOf[Text]);

* sequenceFile (path, class[K], class[V]): reads sequence file.

ex: val depseq = sc.sequenceFile[Int, String]
("/user/cloudera/output/ITversity/Spark/simple-app.seq");

* textFile (path) - reads textfile from HDFS/LFS

* wholeTextFiles (path) - Reads directory of text files from HDFS/LFS
returns RDD[String, String]
↓ file path ↓ file content

✗ A pair is a type of tuple with 2 fields (K, V)

RDD operations

Transformations - are operations that return another RDD.
- it does not evaluate the statement immediately.
- statements are executed only on Action operations.

cache() - Persists the RDD with default storage level (mem only)

coalesce(n) - returns a new RDD with ~~spec~~ specified number of partitions
- used to reduce number of partitions after a filter operations when the dataset become sufficiently small

collect(f:) - returns new RDD by applying specified functions
collect() - returns RDD containing all element of this RDD.

context - returns Spark context that this RDD is created on.

count() - number of elements in this RDD.

countByValue() (Ordering)
- returns count of each unique value in this RDD as a map.
- for single fields its unique value.
- for tuples its unique combination of fields

~~dist~~ distinct(n): n = num Partitions
- return RDD containing distinct elements

filter(f:(T) => Boolean) : returns RDD containing element that satisfy the specified predicate

first() : returns first element

flatMap : flatMap[U] (f: (T) => U)

- Returns an RDD by first applying the function to all elements of the RDD & then flattening the results
- This requires the ~~function~~ predicate returns a collection instead of single element.

ex: Word count - book.flatMap(line => line.split(" "));

fold (zeroValue) (op: (T, T) => T)

→ Aggregates the elements and then result of each partition
→ the zero value is passed as first argument to the predicate during the first call.

foreach (f: T => Unit)

- Applies the predicate to all elements of RDD
- This is an Action & not transformation, & does not return anything
- used for side-effects like printing elements of RDD to screen/file

groupBy (f: (T) => K) returns RDD[(K, Iterable[T])]

- groups a tuple by the specified predicate (the predicate should return a field)
- this function creates a nested structure RDD
- we can use the mapValues (for aggregations) flatmapValues (for sorting/ranking) to process further

~~Reduce By Key~~

intersection (other: RDD)
- returns an RDD which contains common elements to this and other RDD.

keyBy [K] (f: (T) ⇒ K) returns RDD [(K, T)]
- creates a tuple of K/V pair where the key is derives based on applying the predicate
ex: depRDD.keyBy (x ⇒ x.dep.id)
Will return RDD[(dep.id, depRec)]

map - returns a new RDD by applying function to all elements of this RDD.
map[U] (f: T ⇒ U)
- used for doing transformations on whole RDD (not by key)
- This is PUSH model since elements are pushed into our function one by one

mapPartitions (f: (Iterator (T) ⇒ Iterator (U))
mapPartitionsWithIndex [U] (f: (Int, Iterator (T) ⇒ Iterator (U))
- Similar to map but applies the function at partition level ~~instead~~
- So if we need to perform any initialization only once per partition
- mapPartitionsWithIndex() also supplies the partition number, in addition to an Iterable (T) used to iterate over the elements
- These functions returns an Iterable (U) where U is the transformation from V.
- In contrast to map() function this is a pull model since our function needs to pull data from the Iterable

max() (Ordering)
min() (Ordering)

persists (newlevel) - Set RDD's storage level to persists values across operations.

reduce(f: (T, T) ⇒ T)
→ used to perform overall / global aggregations (sum, max, min)
→ can be applied to single elements, tuples etc.
→ cannot be used for per key aggregations.

ex: Find max price of product
product RDD. reduce ((x, y) ⇒ if (x.price > y.price) x else y));

repartition (numPartitions) : return new RDD that has exactly numPartitions

SaveAsObjectFile(path)
saveAsTextFile (path)

SaveAsNewAPIHadoopFile (path, class[key], class[value],
class[OutputFormat[key, value]]);

ex: import org.apache.hadoop.mapreduce.lib.output._

depRDD.saveAsNewAPIHadoopFile (outputpath,
classOf[IntWritable], classOf[Text],
classOf[SequenceFileOutputFormat(IntWritable, Text)])

SortBy (f: (T) ⇒ K, ascending = true, numPartitions =)

- Used for global sorting of RDD.
- Can be used with single fields, pairs, tuples
- the predicate specifies the key to sort on.
- can be used to sort on multiple fields.
  using (k1, k2)

ex: productRDD.sortBy(x ⇒ (x._2, -x._5))
Sorts the product RDD by 2nd field ASC, then by fifth field in DESC

Subtract (other: RDD)
- returns RDD that contains elements in this which are not present in other.

Ranking Global

take (num) - take first n elements.
- this is an action that returns an Array.

takeOrdered (N) (ordering)
- first sorts the elements either by implicit ordering or by the specified ordering. & then takes first n elements.
- by default Ascending order.
- can specify ordering function for DESC order.

productsRDD.take(3); ASC. Default order (first field in tuple.) bottom 3

productsRDD.takeOrdered(3)(Ordering[Int].on(x ⇒ x._1))
(equivalent to previous bottom 3 based on first field

productsRDD.takeOrdered(3)(Ordering[Int].reverse.on(x => x._3))
// Top 3 based on third field

productsRDD.takeOrdered(3)
(Ordering[(Int, Float)].on(x => (x._2, -x._5)))
Top 3 based on ordering with multiple columns.
first column is Int which is 2nd field in tuple
Second column is Float which is 5th field in tuple

top(N)(ordering)
- works in exact opposite way of takeOrdered
by sorting elements in Desc order (based on first field)
- we can specify Ordering function to override this behavior (desc, or other field or multiple fields) by providing ordering function.

productsRDD.top(3)(Ordering[Float].on(x => x._2))
→ top 3 records based on sorting by field 2

union(other: RDD)
- return the RDD which is union of this with other

unpersist(blocking = true)
- mark an RDD non-persistent & remove all blocks for it from memory & disk

PAIR RDD FUNCTIONS

- only available for tuples with two fields (K, V) pair

reducer function

reduceByKey (func : (V, V) ⇒ V)

- passes in an accumulator & value (one after the other) into the predicate & the predicate should return a single tuple (value) which is associative & commutative.
- Similar to reduce() but the values will be first grouped by the key & then the values for each key will be passed into the predicate.
- is used for doing aggregations (sum, max, min) per key basis.
- is a transformation * Uses a combiner which has same logic as reducer function

Ex: totalOrderAmtPerOrder
= orders.reduceByKey
((acc, value) ⇒ (acc + value));

[Assuming orders is RDD[Int, Float]
↓ orderid ↓ order sub total

CombineByKey [C] (combineFunc : (acc, value) ⇒ Value, func : (acc, value) ⇒ Value)

↑ combiner function ↑ reduce function

- Similar to reduceByKey but used when the combine logic & reduce logic is different
- The input to the combiner & output of combine & reduce should be same datatype

aggregateByKey (zeroValue : U) (combineOp : (U, V) ⇒ U, reduceOp : (U, U) ⇒ U)

- more flexible than reduceByKey & combineByKey.
- Used when we need a different combine & reduce logic & further we need that the input values (elements of values for each key is of different datatype (V) and the output of combine & reduce function need to be of different datatype.
- It also allows us the specify a zero value which gets passed in to combineOp as first U per key.
This zero value also forms the type that we are expecting after the combine & reduce operations

ex: Calculating Avg revenue per day.

```
val revenueAvgPerday = orderRevMap.aggregateByKey(
        ((0.0,0)),   <- we are expecting tuple of (Float, Int)
        ((acc, val) => {(acc._1 + value, acc._2 + 1)}
```

↑ Combine op.

↑ we are accumulating the total in first field of tuple & count in 2nd field.

```
        ((acc, val)) => {(acc._1 + value._1, acc._2 + val._2)}
```

↑ reduce op input parameter match combine op output i.e tuple (Int, Float) so the acc & val both are tuples

↑ we are adding up all totals & all counts

foldByKey (zeroVal : V) (func : (V, V) ⇒ V)

- Similar to ~~foldByKey()~~ aggregateByKey that we can specify a zero value but this does not allow seperate combine & reduce logic and also does not allow changing type of elements & output

groupByKey (numPartitions = default)

- groups a K, V flat structure & returns a key with Iterable (values) nested structure.
- Normally used for sorting & ranking within a key.
- Should not be used for calculating aggregates since this is inefficient (does not use combiner)

ex: val prodsGroupedByCat = products.groupByKey();
Assuming products is K, V pair where K - category id.

mapValues [U] (func: (V) => U)

- used to perform a function over all values per key
- can perform transformations (aggregates) to return single value or sorting, ranking of the values to return a collection - (However this will result in a nested K, V structure.
- For each key the values will be passed in as Iterable to the predicate (PULL model) * In case where V is Iterable
- Normally used after grouping of data

ex:

val grouped = orders.groupByKey() OR orders.groupBy(...)
Assuming K, V where K is order id & V is order subtotals

val revenuePerOrder
= grouped.mapValues(V => V.sum)

↑ V could be Iterable or single value

flatMapValues[U](func: [V] ⇒ U);

- Similar to mapValues but expects U to be a collection instead of single value so that it flattens the result.
- Used mostly for sorting & ranking values within a key.

ex: // Top 3 products per category by price

Val prodGroup = product.groupByKey() or groupBy(---K)

Assume products is K,V K is prod cat & V is tuple (product record)

Val top3 = prodGroup.flatMapValues

(x ⇒ x.toList.sortBy(-_4).take(3))

↓

This will result in collection of top 3 records sorted by price DESC

flatMapValues will convert this nested structure K, Iterable(V) into flat structure

K, V1
K, V2
K, V3

countByKey()

- will return the number of values for each key in a local map.
- NOTE: This does not require grouping prior to this call i.e the K,V could be a flat structure.

ex: Val orderItems = ... (Assume orderItems is K,V with order id as key & the order item Record (tuple) as value

Val numOrdersPerOrderId = orderItems.countByKey()

- Nested Tuples (Tuple inside a tuple) is not considered to be nested structure. It still is regarded as flat structure

Keys() - returns an RDD containing keys of each tuple

Values() - returns an RDD containing values of each tuple

lookup(key: K) - returns a List of values for a given RDD key
- This is an action

~~join[W](other: [K,W]) returns RDD[K, (Iterable[V], Iterable[W])]~~

join[W](other: RDD[K,W]) returns RDD[(K, (V,W))]

- joins two RDDs & returns a tuple containing the key & the corresponding records from this & other as a tuple
- generates a flat structure
- performs an inner join (filters out records from this & other that don't match)

ex: val joined = orders.join(orderItems)

Assuming orders is K,V where K is orderid (int)
V is order record string
orderItems is K,V where K is orderid (int)
V is orderItem record string

then joined will be

RDD: [(Int, (String, String))]

Int → key
String → order record
String → orderItem Record

cogrouped.flatMapValues (v => {
for (v1 <- v._1, v2 <- v._2) yield (v1, v2);
})

v1 ← value from first iterable
v2 ← value from 2nd iterable
combine them into a tuple

leftOuterJoin (other : RDD) - same as join but performs left outer join
- which means the 2nd ~~field~~ field in the value tuple of the result can be empty for missing records in other
- generates flat structure

rightOuterJoin (other : RDD)
- same as join but performs right outer join
- the 1st field in the value tuple of the result could be empty for missing records in this
- generates flat structure

fullOuterJoin (other : RDD)
- performs a full outer join
- i.e the 1st or 2nd field in the value tuple of the result will be empty for missing records in this or other respectively
- generates a flat structure.

cogroup (groupWith) [W] (other : RDD[K, V]) : RDD[(K, (Iterable(V), Iterable(W)))]
- performs a full outer join but generates a nested structure because it also groups the result after the join by the key.

ex: Val Stocks - Assume K, V with K as stock symbol (String) & V as stock record (String)
Val companies - Assume K, V with K as stock symbol (String) & V as company details (String)

Val cogrouped = stocks.cogroup(companies)

cogrouped will be RDD[(String, (Iterable(String), Iterable(String))]

String → Key = Stock symbol
Iterable → iterable for iterating through all stock records for this key
Iterable → iterable for iterating through all companies record for this key.

We can remove nesting by using flatMapValues - to combine the two iterable into one Iterable.

cogrouped.flatMapValues (

JAVA Pair RDD

Sort By key (asc : boolean)

- Sort the RDD by key so that each partition contains sorted range of elements

ex : Val products = Assume K, V K is price & V is product record.

val sortedByPrice = products.sortByKey() ASC

products.sortByKey(false) DESC

Maps

Val employees = ~~Map("Manager"~~ 1, "One", 2, "Two", 3, "T

Val emptyMap = Map[Int, String]()

Val numMap = Map(1 -> "One", 2 -> "Two", 3 -> "Three")

* numMap.contains(1)

  numMap.get(1) //Acces

* numMap(2) //Acces

foreach, count(), find(), map(), filter(), flatMap(), exists() ~~same as list~~

min, max, mkString, isEmpty same as List

* filterKeys(f(k) => Boolean)
* keys() list of keys
* values() list of values
* mapValues(f: (V) => C) : Map[K, C]

Tuples - Ordered collection of fields

Val tupleMerge = (103, "Simpson", 10.25)

* tupleMerge._1, tupleMerge._2, tupleMerge._3 // Access

tupleMerge.productIterator // Iterator

tupleMerge.toString()

//Assign multiple variables using tuples

* var(a, b, c, d) = (1, 2, 3, 4)

SETS (Unique elements only

```
val set1 = Set(1, 2, 1, 4, 5, 3, 2)          // Immutable
val set2 = collection.mutable.HashSet(1, 1, 2, 1, 3, 5)  // Mutable
```

* set1.exists(1)
* set1 -= 2 // remove element
* set1 --= Array(1, 2) // remove elements.

// Most methods same as list

Mutable collections

```
var arrayBuffer = ArrayBuffer(1, 2, 3)
var listBuffer = collection.mutable.ListBuffer(1, 2, 3)
var hashSet = collection.mutable.Set(0, 1, 1, 2, 2)
var hashMap = collection.mutable.Map("one" -> 1, "two" -> 2)

arrayBuffer = new ArrayBuffer[Int]
listBuffer = new collection.mutable.ListBuffer()
hashSet = new collection.mutable.HashSet()
hashMap = new collection.mutable.HashMap()
```

For loops

Range (1 to 10) // gives list of to 10 inclusive
Range (1 until 10) // give list of to 10 exclusive

Range (1, 10) // 1 to 10
Range (1, 10, 2) // steps of 2

1 to 10 // 1 to 10 inclusive
1 until 10 // 1 to 10 exclusive
1 to 10 by 2 // Steps of 2
10 to 1 by -2

Transformations using yield

for (i <- 1 to 10) yield i * i) // transform
for (i <- 1 to 10 if i % 2 == 0) yield // conditional transform

for (i <- 1 to 5; j <- 6 to 10) yield (i, j) ; multiple generators

// Pattern matching
Var list = List ((1, 2), (3, 4), (4, 5))
for ((n1, n2) <- list) println (n1 + n2)

for ((1, n2) <- list) yield n2 ; select and yield where first field is 1

Function

a is automatically forced to val

a is pass by reference

```
def doSomething(a : Int) : Unit = {
}
```

no return type

```
def getSum (num1 : Int, num2 : Int) : Int = {
}
getSum ( 5, 2 )          // Normal call
get(num2 = 2, num1 = 1)  // Named argument
```

```
def getSum2 (args : Int * ) : Int = {// Variable args
    for (item ← args)
        sum += item
    return sum
}

getSum2 ( 1, 2, 3, 4 )

var mList = (1, 2, 3, 4)
    getSum2 (mList)      // This will not work since we need val
    getSum2 (mList : _*) // This works. ✓
```

Higher Order Func is a function which takes a func as parameter

```
def higherfunc (func : (Int) ⇒ Double, num : Int) = {
    func (num)
}
```

This parameter is a function which takes in Int & returns Double

~~Val divisor5 = (num : Double) ⇒ num/5;~~

```
higherfunc ( num : Int ⇒ num/5 , 15)
```

Anonymous Functions

```
def add1(x: Int, y: Int) = x + y; //method (named)

val add1 = (x: Int, y: Int) => x + y; //anonymous methods

def doSomeOp(f: (Int, Int) => Int) = {  // This is higher order function.
    f(6, 2)
}

//Addition    doSomeOp((x, y) => x + y)    // Anonymous
// Substraction  doSomeOp((x, y) => x - y)    // functions
//Multiplication  doSomeOp((x, y) => x * y)
                doSomeOp(_ * _)    // concise syntax
```

DateTime (joda)

import org.

DateTime (Joda)

```
import org.joda.time._
import org.joda.time.format._

val currentDT = new DateTime()
val specificDT = new DateTime(2015, 5, 24, 18, 1, 34,
                              DateTimeZone.getDefault())

val unixTimeMillis = DateTimeUtils.currentTimeMillis()
val dt = new DateTime(unixTimeMillis, DateTimeZone.getDefault())

val unixTimeMillis = specificDT.getMillis()

val formatter1 = DateTimeFormat.forPattern("dd MMM yyyy hh:mm:ss.
val formatter2 = DateTimeFormat.forPattern("yyyy-mm-dd hh:mm:ss")

val dt = formatter1.parseDateTime("26 OCT 2015 03:15:05.245 AM")
val str = formatter2.print(dt)
    //OR
val str = dt.toString("yyyy-MM-dd hh:mm:ss")
```

DateTime Java import java.util.—;

val currentDT = new Date()

val calendar
// SpecificDT = Calendar.getInstance(TimeZone.getDefault)
calendar.set(2015, 9, 17, 15, 7, 3)
val specificDT = calendar.getTime()

val unixTimeMillis = System.currentTimeMillis()
val dtFromUnix = new Date(unixTimeMillis)

val unixTimeMillis = specificDT.getTime()

// Formatting
import java.text.SimpleDateFormat("dd MMM yyyy hh:mm:ss SSS a")

val fmt1 = new SimpleDateFormat("dd MMM yyyy hh:mm:ss a")
val dt = fmt1.parse("15 OCT 2015 03:15:25 PM")
val str
val fmt2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
val str = fmt2.format(dt);

// Pattern Matching / switch case

n match {
// Simple match
case 0 => 1;
// Conditional match
case x if x > 0 => n * fact(n-1)

// everything else
case _ => -1;

Python

- interpreted as well can be compiled
- object oriented + scripting + functional
- interactive , batch
  - interactive → > python
    > directly write python statements at the prompt
  - batch → > python myprog.py

Built functions

abs(x) - x (int, float)

* bool(x) - string to boolean

chr(i) - i is int.

* filter (func, iterable) - returns iterator for which the func returns True for elements in iterable.
* float(x) - string to float

format (same as str( ))

hex(x) - x is int to hex format string "0x"

input(prompt) -

```
str= input("Enter something")
--> This is some text
print(str)
```

* int(x) - convert string to int

* len(s) - s is string.

* list(iterable) - convert iterable to list
* set(iterable) - convert iterable to set (This is used to remove duplicates)
* map(func, iterable) - transform iterable to new sequence by applying the function.

* max(iterable, key, default)
* min
key → opt, default → opt

open(file, mode='r')

mode → 'r', 'w', 'a', 'b', 't'

* print(*objects, sep='', end='\n')
print(x, y, z ...)

* range(start, stop) - returns immutable sequence type

* reversed(seq) - returns a reversed iterator

* slice(start, stop, step)
step → opt

* sorted (iterable, key, reverse)
  - reverse → True, or false
  - key → funct to specify key to sort on

sorted (values, key = lambda x : x[6], reverse = True)

* sum (iterable) - returns sum of all elements

* tuple (iterable) - convert to tuple

```
print()
    print(x)
    print(x, y, z)

print("%s %d %.5f" % (str, x, y))

print("The value is", x)
```

* String is kind of sequence

Sequences - List, Tuple, Range, String

* Common operations

Immutable Sequence Common operations (~~tuples~~ tuples, Keys of Dictionary)

x in seq — True if x present in seq.

x not in seq

seq1 + seq2 - Concatenate of seq1 & seq2

* For string concatenation the other types must be converted to string explicitly
  eg: "The num is " + str(index)

seq[i] - access ith element of the sequence

seq[i:j] - take slice from i to j (exclusive)

seq[i:j:k] - take slice from i to j in steps of k

len(seq) - length of sequence.

min(seq) - smallest element of seq.

max(seq) - largest element of seq

seq.count(x) - total number of occurences of x in seq.

Mutable Sequence Common Operations (List, Dictionary Values, Strings)

seq[i] = x → assign x to ith element of seq.

seq[i:j] = t → slice of seq from i to j

seq.append(x) → appends x to end of sequence

seq.clear() → removes all elements from seq.

seq.insert(i, x) → inserts x into ith position

seq.pop([i]) → retrieves & removes ith element

seq.remove(x) → removes first occurence of x

seq.reverse() → reverses seq. in place

Text Sequence - str

- Strings can be enclosed in single, double or triple quotes
- Strings in triple quotes can span multiple lines.

str (obj) -> returns string version of object

str.startswith (str, start, end)

str.endswith (str, start, end)
opt

str.find (substr, start, end)

str == "Some string" → returns true

str.format (*args, **kwargs)

- perform string formatting operation

"The sum of {0} & {1} is {2}".format(1, 2, 1+2)

str.lower()

str.upper()

str.split (sep, maxsplit)

str.strip() - removes leading & trailing whitespace

Set Types

set(iterable) - converts iterable to set
→ removes duplicates

~~len(set)~~ len(s)

x in set

set.add(elem)

set.remove(elem)

set.pop()

set.clear()

Map Type - dict

dict1 = dict("one" = 1, "two" = 2, "three" = 3)

len(dict)

dict[key) → returns value corresponding to key else keyError

dict[key] = value → sets value for key.

key in dict - whether k is present in dictionary

iter(dict) - returns an iterator over the keys

dict.clear()

dict.copy()

dict.keys()

dict.values()

Formatting

```
print ("%s %4d %.5f" % (a, b, c))

str1 = "{} {:4d} {:.5f}".format(a, b, c)
```

Conversion

int(x) , float(x) , str(x)
list(x) , tuple(x) , set(x)

Flow Control

```
if (condn):
    statements
    :
elif (condn):
    statements
else:
    statements
```

```
for (x in list1):
    print(x)

for (x in range(0,3)):
    for (y in range(0:3):
        print (list1[x][y])
```

Comments # Single line
''' Multiline
"""
'''

While (condn) :

    Statements :

Functions (Pass by reference by default)

```
def funcName (arg1 , arg2 -- ) :
    print - -
    statements : . . .
```

Variable arguments

```
def funcName (*args) :
    for (x in args)
        - -
        :
        :
```

default args :

```
def funcName (arg1 , arg2 = 50) :
    Statements
    :
```

Named args

```
funcName ( arg2 = 30 , arg1 = "str1")
```

Anonymous

```
func1 = lambda x, y : x + y
```

* lambda can have only one statement.
Hence if requires more complex logic define a function & call within the lambda
OR
use function chaining.

func1(3, 2) gives 6

List Demo

```
grocery_list = ['Juice', 'Tomatoes']
empty_list = []
```

Access: grocery_list[0]

modify: grocery_list[0] = "Apple"

slice: grocery_list[0:3] = (→ exclusive)

```
other_list = ['Potatoes']
```

Append (combine/concatenate): grocery_list + other_list

Append: grocery_list.append('Mango')

Extend (Append multiple): grocery_list.extend(['Onions', 'Grapes'])

Insert: grocery_list.insert(3, 'oranges')

Remove: grocery_list.remove('Apples');

Reverse: grocery_list.reverse()

Distinct: set(grocery_list) // convert to set

Count: len(grocery_list)

Min: min(grocery_list) // works with numbers

Max: max(num_list)

Sum: sum(num_list)

print: print(grocery_list);

Membership : if ('Apples' in grocery_list)
if ("Bananas" not in grocery_list)

TupleDemo : enclosed in ()
pi_tuple = (3, 1, 4, 5, 9)

modification : list(pituple) // convert to list & modify
// Tuples are immutable

All other operations same as list

Sorted (iterable, funckey, reverse = true/false)
↳ lambda expr which provides sort key.

reversed (iterable, func: key)
↳ lambda expr which provides reverse key.

Dictionary : (Keys immutable (tuples, string, numbers)
Values mutable
: keys are automatically sorted
: represented by { key: value, key: value }

Access : dict[1]
modify : dict[1] = "One"
keys : dict1.keys()
values : dict1.values()
Insert : dict[8] = "Eight"
Add multiple : dict1.~~extend~~ update ({9: "Nine", 10: "Ten"})

Clear : ~~dict[1] = 0~~ dict.clear()
~~dict~~ Count : len(dict1)

String demo

Format	"{} ---- {:04d} ---- {:.5f}".format(a, b, c)
Upper	str1.upper()
lower	str1.lower()
Find	str1.find("Sometext")
length	len(str1)
replace	str1.replace("oldtext", "newtext")
Trim	str1.strip()
Split	str1.split(",")
Equals	str1 == "Sometext"
Startswith	str1.startswith("Sometext")
endswith	str1.endswith("Sometext")
Substring	str1[0:4] → exclusive
Concat	str1 + str2