* + **pyspark.sql.SparkSession**
  + **pyspark.sql.Catalog**
  + **pyspark.sql.DataFrame**
  + **pyspark.sql.Column**
  + **pyspark.sql.Row**
  + **pyspark.sql.GroupedData**
  + **pyspark.sql.PandasCogroupedOps**
  + **pyspark.sql.DataFrameNaFunctions**
  + **pyspark.sql.DataFrameStatFunctions**
  + **pyspark.sql.Window**

**pyspark.sql.SparkSession**

You can create as many SparkSession objects you want using either

* SparkSession.builder or
* SparkSession.newSession.

**SparkSession in PySpark shell**

Be default PySpark shell provides “spark” object; which is an instance of SparkSession class. We can directly use this object where required in spark-shell. Start your “pyspark” shell from $SPARK\_HOME\bin folder and enter the below statement.

sqlcontext = spark.sqlContext

Similar to PySpark shell, In most of the tools, the environment itself creates default SparkSession object for us to use so you don’t have to worry about creating SparkSession object.

**Spark session programmatically**

SparkSession is an entry point to PySpark, and creating a SparkSession instance would be the first statement you would write to program The entry point to programming Spark with the Dataset and DataFrame API.

A SparkSession can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files.

|  |
| --- |
| **spark = SparkSession.builder \**  **.master("local") \**  **.appName("My\_Word\_Count") \**  **.config("spark.some.config.option", "some-value") \**  **.getOrCreate()** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Methods**   |  |  | | --- | --- | | [**createDataFrame**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.createDataFrame.html#pyspark.sql.SparkSession.createDataFrame)**(data[, schema, …])** | Creates a DataFrame from an RDD, a list or a pandas.DataFrame. | | [**getActiveSession**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.getActiveSession.html#pyspark.sql.SparkSession.getActiveSession)**()** | Returns the active [SparkSession](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.html#pyspark.sql.SparkSession) for the current thread, returned by the builder | | [**newSession**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.newSession.html#pyspark.sql.SparkSession.newSession)**()** | Returns a new [SparkSession](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.html#pyspark.sql.SparkSession) as new session, that has separate SQLConf, registered temporary views and UDFs, but shared SparkContext and table cache. | | [**range**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.range.html#pyspark.sql.SparkSession.range)**(start[, end, step, numPartitions])** | Create a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) with single [pyspark.sql.types.LongType](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.types.LongType.html#pyspark.sql.types.LongType) column named id, containing elements in a range from start to end (exclusive) with step value step. | | [**sql**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.sql.html#pyspark.sql.SparkSession.sql)**(sqlQuery)** | Returns a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) representing the result of the given query. | | [**stop**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.stop.html#pyspark.sql.SparkSession.stop)**()** | Stop the underlying SparkContext. | | [**table**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.table.html#pyspark.sql.SparkSession.table)**(tableName)** | Returns the specified table as a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Attributes**   |  |  | | --- | --- | | [**builder**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.html#pyspark.sql.SparkSession.builder) | A class attribute having a **Builder** to construct [**SparkSession**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.html#pyspark.sql.SparkSession) instances. | | [**catalog**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.catalog.html#pyspark.sql.SparkSession.catalog) | Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc. | | [**conf**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.conf.html#pyspark.sql.SparkSession.conf) | Runtime configuration interface for Spark. | | [**read**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.read.html#pyspark.sql.SparkSession.read)**()** | Returns a **DataFrameReader** that can be used to read data in as a [**DataFrame**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). to read records from csv, parquet, avro and more file formats into DataFrame. | | [**readStream**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.readStream.html#pyspark.sql.SparkSession.readStream) | Returns a **DataStreamReader** that can be used to read data streams as a streaming [**DataFrame**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). | | [**sparkContext**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.sparkContext.html#pyspark.sql.SparkSession.sparkContext) | Returns the underlying **SparkContext**. | | [**streams**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.streams.html#pyspark.sql.SparkSession.streams) | Returns a **StreamingQueryManager** that allows managing all the **StreamingQuery** instances active on *this* context. | | [**udf**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.udf.html#pyspark.sql.SparkSession.udf)**()** | Returns a **UDFRegistration** for UDF registration. Creates a PySpark UDF to use it on DataFrame, Dataset, and SQL. | | [**version**](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.version.html#pyspark.sql.SparkSession.version)**()** | The version of Spark on which this application is running. | |

|  |
| --- |
| from pyspark.sql import SparkSession  spark = SparkSession.builder.master("local[1]") \  .appName('SparkByExamples.com') \  .getOrCreate() |

|  |
| --- |
| **builder.appName(name)**  Sets a name for the application, which will be shown in the Spark web UI.  If no application name is set, a randomly generated name will be used. |

|  |
| --- |
| **builder.config(key=None, value=None, conf=None)**  Sets a config option. Options set using this method are automatically propagated to both SparkConf and SparkSession’s own configuration.  Key: str, optional  a key name string for configuration property  value: str, optional  a value for configuration property  conf: SparkConf, optional  an instance of SparkConf |

|  |
| --- |
| **builder.getOrCreate()**  Gets an existing SparkSession or, if there is no existing one, creates a new one based on the options set in this builder.  This method first checks whether there is **a valid global default SparkSession**, and if yes, return that one. If no valid global default SparkSession exists, the method creates a new SparkSession and assigns the newly created SparkSession as the global default.  >>>s1 = SparkSession.builder.config("k1", "v1").getOrCreate()  >>>s1.conf.get("k1") == "v1"  True  In case an existing SparkSession is returned, the config options specified in this builder will be applied to the existing SparkSession.  >>>s2 = SparkSession.builder.config("k2", "v2").getOrCreate()  >>>s1.conf.get("k1") == s2.conf.get("k1")  True  >>>s1.conf.get("k2") == s2.conf.get("k2")  True |

|  |
| --- |
| **builder.master(master: str\_url\_for\_master)**  Sets the Spark master URL to connect to, such as “local” to run locally, **“local[4]”** to run locally with 4 cores, or “spark://master:7077” to run on a Spark standalone cluster.  If you are running it on the cluster you need to use your master name as an argument to master(). usually, it would be either yarn or mesos depends on your cluster setup.  Use local[x] when running in Standalone mode. x should be an integer value and should be greater than 0; this represents how many partitions it should create when using RDD, DataFrame, and Dataset. Ideally, x value should be the number of CPU cores you have. |

|  |
| --- |
| **pyspark.sql.SparkSession.catalog**  property SparkSession.catalog  Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc.  Returns: catalog |

|  |
| --- |
| **pyspark.sql.SparkSession.conf**  This is the interface through which the user **can get and set all Spark and Hadoop configurations** that are relevant to Spark SQL. When getting the value of a config, this defaults to the value set in the underlying SparkContext, if any.  Returns: pyspark.sql.conf.RuntimeConfig |

|  |
| --- |
| **pyspark.sql.SparkSession.createDataFrame(**  **data,**  **schema=None,**  **samplingRatio=None,**  **verifySchema=True**  **)**  -from an **RDD, a list or a pandas.DataFrame**  -**schema is a list of column names**, the type of each column will be inferred from data  -When schema is None, it will try to infer the schema  -If schema inference is needed, samplingRatio is used **to determined the ratio of rows used for schema inference.** The first row will be used if samplingRatio is None  -VerifySchema will verify data types of every row against schema. Enabled by default.  k = [('Alice', 1)]  spark.createDataFrame(k).collect()  spark.createDataFrame(k, ['name', 'age']).collect()  d = [{'name': 'Alice', 'age': 1}]  spark.createDataFrame(d).collect()  rdd = sc.parallelize(k)  spark.createDataFrame(rdd).collect()  df = spark.createDataFrame(rdd, ['name', 'age'])  df.collect()  from pyspark.sql import Row  Person = Row('name', 'age')  person = rdd.map(lambda r: Person(\*r))  df2 = spark.createDataFrame(person)  df2.collect()  from pyspark.sql.types import \*  schema = StructType([  StructField("name", StringType(), True),  StructField("age", IntegerType(), True)  ])  df3 = spark.createDataFrame(rdd, schema)  df3.collect() |

|  |
| --- |
| **pyspark.sql.SparkSession.getActiveSession()**  Returns the active SparkSession for the current thread, returned by the builder.  Returns: SparkSession  s = SparkSession.getActiveSession()  k = [('Alice', 1)]  rdd = s.sparkContext.parallelize(k)  df = s.createDataFrame(rdd, ['name', 'age'])  df.select("age").collect() |

|  |
| --- |
| **pyspark.sql.SparkSession.newSession()**  Returns a new SparkSession as new session, that has **separate SQLConf, registered temporary views and UDFs, but shared SparkContext and table cache.** |

|  |
| --- |
| **pyspark.sql.(start, end=None, step=1, numPartitions=None)**  Create a DataFrame with single pyspark.sql.types.LongType column named id, containing elements in a range from start to end (exclusive) with step value step.  >>>spark.range(1, 7, 2).collect()  [Row(id=1), Row(id=3), Row(id=5)] |

|  |
| --- |
| **pyspark.sql.SparkSession.read**  Using csv("path") or format("csv").load("path") of DataFrameReader, you can read a CSV file into a PySpark DataFrame, These methods take a file path to read from as an argument. When you use format("csv") method, you can also specify the Data sources by their fully qualified name, but for built-in sources, you can simply use their short names (csv,json, parquet, jdbc, text e.t.c). by default data type for all these columns is treated as String.  spark = SparkSession.builder().master("local[1]")  .appName("SparkByExamples.com")  .getOrCreate()  df = spark.read.csv("/tmp/resources/zipcodes.csv")  df.printSchema()  df = spark.read.format("csv")  .load("/tmp/resources/zipcodes.csv")  df.printSchema()  df2 = spark.read.option("header",True) \  .csv("/tmp/resources/zipcodes.csv")  df3 = spark.read.options(header='True',  inferSchema='True',  delimiter=',') \  .csv("/tmp/resources/zipcodes.csv")  #Read Multiple CSV Files  df1 = spark.read.csv("path1, path2, path3")  df2 = spark.read.csv("Folder path")  # delimiter option is used to specify the column delimiter of the CSV file. By default, it is comma (,) character, but can be set to any character like pipe(|), tab (\t), space using this option.  df3 = spark.read.options(delimiter=',') \  .csv("C:/apps/zipcodes.csv")  df4 = spark.read.options(inferSchema='True',delimiter=',') \  .csv("src/main/resources/zipcodes.csv")  df4 = spark.read.option("inferSchema",True) \  .option("delimiter",",") \  .csv("src/main/resources/zipcodes.csv")  If you know the schema of the file ahead and do not want to use the inferSchema option for column names and types, use user-defined custom column names and type using schema option.  schema = StructType() \  .add("RecordNumber",IntegerType(),True) \  .add("Zipcode",IntegerType(),True) \  .add("ZipCodeType",StringType(),True) \  .add("City",StringType(),True) \  df\_with\_schema = spark.read.format("csv") \  .option("header", True) \  .schema(schema) \  .load("/tmp/resources/zipcodes.csv")  Use the write() method of the PySpark DataFrameWriter object to write PySpark DataFrame to a CSV file. **Other options available quote, escape, nullValue, dateFormat, quoteMode .**  df.write.option("header",True) \  .csv("/tmp/spark\_output/zipcodes")  df2.write.options(header='True', delimiter=',') \  .csv("/tmp/spark\_output/zipcodes")  PySpark DataFrameWriter also has a method mode() to specify saving mode.  overwrite – mode is used to overwrite the existing file.  append – To add the data to the existing file.  ignore – Ignores write operation when the file already exists.  error – a default option when the file already exists, it returns an error.  df2.write.mode('overwrite')  .csv("/tmp/spark\_output/zipcodes")  # OR  df2.write.format("csv")  .mode('overwrite')  .save("/tmp/spark\_output/zipcodes")  import pyspark  from pyspark.sql import SparkSession  from pyspark.sql.types import StructType,StructField, StringType, IntegerType  from pyspark.sql.types import ArrayType, DoubleType, BooleanType  from pyspark.sql.functions import col,array\_contains  spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()  df = spark.read.csv("/tmp/resources/zipcodes.csv")  df.printSchema()  df2 = spark.read.option("header",True) \  .csv("/tmp/resources/zipcodes.csv")  df2.printSchema()    df3 = spark.read.options(header='True', delimiter=',') \  .csv("/tmp/resources/zipcodes.csv")  df3.printSchema()  schema = StructType() \  .add("RecordNumber",IntegerType(),True) \  .add("Zipcode",IntegerType(),True) \  .add("ZipCodeType",StringType(),True) \  .add("City",StringType(),True) \  .add("State",StringType(),True)  df\_with\_schema = spark.read.format("csv") \  .option("header", True) \  .schema(schema) \  .load(/tmp/resources/zipcodes.csv")  df\_with\_schema.printSchema()  df2.write.option("header",True) \  .csv("/tmp/spark\_output/zipcodes123") |

|  |
| --- |
| **pyspark.sql.SparkSession.readStream**  Returns a DataStreamReader that can be used to read data streams as a streaming DataFrame.  Returns: DataStreamReader |

|  |
| --- |
| **pyspark.sql.SparkSession.sparkContext**  Returns the underlying SparkContext. |

|  |
| --- |
| **pyspark.sql.SparkSession.sql(sqlQuery)**  Returns a DataFrame representing the result of the given query.  df.createOrReplaceTempView("table1")  df2 = spark.sql("SELECT field1 AS f1, field2 as f2 from table1")  df2.collect() |

|  |
| --- |
| **pyspark.sql.SparkSession.stop()**  Stop the underlying SparkContext. |

|  |
| --- |
| **pyspark.sql.SparkSession.streams**  Returns a StreamingQueryManager that allows managing all the StreamingQuery instances active on this context. |

|  |
| --- |
| **pyspark.sql.SparkSession.table(tablename)**  Returns the specified table as a DataFrame.  df.createOrReplaceTempView("table1")  df2 = spark.table("table1")  sorted(df.collect()) == sorted(df2.collect())  🡪True |

|  |
| --- |
| **pyspark.sql.SparkSession.udf**  Returns a UDFRegistration for UDF registration. |

|  |
| --- |
| **pyspark.sql.SparkSession.version**  The version of Spark on which this application is running. |

**pyspark.sql.conf.RuntimeConfig**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **pyspark.sql.conf.RuntimeConfig(jconf)**  User-facing configuration API, accessible through SparkSession.conf.  Options set here are automatically propagated to the Hadoop configuration during I/O.  **Methods**   |  |  | | --- | --- | | **get**(key[, default]) | Returns the value of Spark runtime configuration property for the given key, assuming it is set. | | **isModifiable**(key) | Indicates whether the configuration property with the given key is modifiable in the current session. | | **set**(key, value) | Sets the given Spark runtime configuration property. | | **unset**(key) | Resets the configuration property for the given key. | |