**Crossjoin**

**>>>** df.select("age", "name").collect()

[Row(age=2, name=u'Alice'), Row(age=5, name=u'Bob')]

**>>>** df2.select("name", "height").collect()

[Row(name=u'Tom', height=80), Row(name=u'Bob', height=85)]

**>>>** df.crossJoin(df2.select("height")).select("age", "name", "height").collect()

[Row(age=2, name=u'Alice', height=80), Row(age=2, name=u'Alice', height=85),

Row(age=5, name=u'Bob', height=80), Row(age=5, name=u'Bob', height=85)]

**drop**

**>>>** df.drop('age').collect()

[Row(name=u'Alice'), Row(name=u'Bob')]

**>>>** df.drop(df.age).collect()

[Row(name=u'Alice'), Row(name=u'Bob')]

**>>>** df.join(df2, df.name == df2.name, 'inner').drop(df.name).collect()

[Row(age=5, height=85, name=u'Bob')]

**>>>** df.join(df2, df.name == df2.name, 'inner').drop(df2.name).collect()

[Row(age=5, name=u'Bob', height=85)]

**>>>** df.join(df2, 'name', 'inner').drop('age', 'height').collect()

[Row(name=u'Bob')]

**Dropna()**

**>>>** df4.na.drop().show()

+---+------+-----+

|age|height| name|

+---+------+-----+

| 10| 80|Alice|

+---+------+-----+

Dtypes

**>>>** df4.na.fill(50).show()

+---+------+-----+

|age|height| name|

+---+------+-----+

| 10| 80|Alice|

| 5| 50| Bob|

| 50| 50| Tom|

| 50| 50| null|

+---+------+-----+

**>>>** df4.na.fill({'age': 50, 'name': 'unknown'}).show()

+---+------+-------+

|age|height| name|

+---+------+-------+

| 10| 80| Alice|

| 5| null| Bob|

| 50| null| Tom|

| 50| null|unknown|

+---+------+-------+

**filter**(*condition*)

Filters rows using the given condition.

[**where()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.where) is an alias for [**filter()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.filter).

|  |  |
| --- | --- |
| **Parameters:** | **condition** – a [**Column**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Column) of **[types.BooleanType](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.types.BooleanType" \o "pyspark.sql.types.BooleanType)** or a string of SQL expression. |

**>>>** df.filter(df.age > 3).collect()

[Row(age=5, name=u'Bob')]

**>>>** df.where(df.age == 2).collect()

[Row(age=2, name=u'Alice')]

**>>>** df.filter("age > 3").collect()

[Row(age=5, name=u'Bob')]

**>>>** df.where("age = 2").collect()

[Row(age=2, name=u'Alice')]

**foreach**(*f*)

Applies the f function to all [**Row**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Row) of this **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)**.

This is a shorthand for df.rdd.foreach().

**>>> def** f(person):

**...**  **print**(person.name)

**>>>** df.foreach(f)

**>>> def** f(people):

**...**  **for** person **in** people:

**...**  **print**(person.name)

**>>>** df.foreachPartition(f)

**freqItems**(*cols*, *support=None*)

Finding frequent items for columns, possibly with false positives. Using the frequent element count algorithm described in “<http://dx.doi.org/10.1145/762471.762473>, proposed by Karp, Schenker, and Papadimitriou”. [**DataFrame.freqItems()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.freqItems) and [**DataFrameStatFunctions.freqItems()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrameStatFunctions.freqItems) are aliases.

**Note**

This function is meant for exploratory data analysis, as we make no guarantee about the backward compatibility of the schema of the resulting DataFrame.

|  |  |
| --- | --- |
| **Parameters:** | * **cols** – Names of the columns to calculate frequent items for as a list or tuple of strings. * **support** – The frequency with which to consider an item ‘frequent’. Default is 1%. The support must be greater than 1e-4. |

**groupBy**(*\*cols*)

Groups the **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** using the specified columns, so we can run aggregation on them. See **[GroupedData](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.GroupedData" \o "pyspark.sql.GroupedData)** for all the available aggregate functions.

[**groupby()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.groupby) is an alias for **[groupBy()](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame.groupBy" \o "pyspark.sql.DataFrame.groupBy)**.

|  |  |
| --- | --- |
| **Parameters:** | **cols** – list of columns to group by. Each element should be a column name (string) or an expression ([**Column**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Column)). |

**>>>** df.groupBy().avg().collect()

[Row(avg(age)=3.5)]

**>>>** sorted(df.groupBy('name').agg({'age': 'mean'}).collect())

[Row(name=u'Alice', avg(age)=2.0), Row(name=u'Bob', avg(age)=5.0)]

**>>>** sorted(df.groupBy(df.name).avg().collect())

[Row(name=u'Alice', avg(age)=2.0), Row(name=u'Bob', avg(age)=5.0)]

**>>>** sorted(df.groupBy(['name', df.age]).count().collect())

[Row(name=u'Alice', age=2, count=1), Row(name=u'Bob', age=5, count=1)]

**join**(*other*, *on=None*, *how=None*)

Joins with another **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)**, using the given join expression.

|  |  |
| --- | --- |
| **Parameters:** | * **other** – Right side of the join * **on** – a string for the join column name, a list of column names, a join expression (Column), or a list of Columns. If *on* is a string or a list of strings indicating the name of the join column(s), the column(s) must exist on both sides, and this performs an equi-join. * **how** – str, default ‘inner’. One of *inner*, *outer*, *left\_outer*, *right\_outer*, *leftsemi*. |

The following performs a full outer join between df1 and df2.

**>>>** df.join(df2, df.name == df2.name, 'outer').select(df.name, df2.height).collect()

[Row(name=None, height=80), Row(name=u'Bob', height=85), Row(name=u'Alice', height=None)]

**>>>** df.join(df2, 'name', 'outer').select('name', 'height').collect()

[Row(name=u'Tom', height=80), Row(name=u'Bob', height=85), Row(name=u'Alice', height=None)]

**>>>** cond = [df.name == df3.name, df.age == df3.age]

**>>>** df.join(df3, cond, 'outer').select(df.name, df3.age).collect()

[Row(name=u'Alice', age=2), Row(name=u'Bob', age=5)]

**>>>** df.join(df2, 'name').select(df.name, df2.height).collect()

[Row(name=u'Bob', height=85)]

**>>>** df.join(df4, ['name', 'age']).select(df.name, df.age).collect()

[Row(name=u'Bob', age=5)]

**limit**(*num*)

Limits the result count to the number specified.

**>>>** df.limit(1).collect()

[Row(age=2, name=u'Alice')]

**>>>** df.limit(0).collect()

[]

**replace**(*to\_replace*, *value*, *subset=None*)

Returns a new **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** replacing a value with another value. [**DataFrame.replace()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.replace) and [**DataFrameNaFunctions.replace()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrameNaFunctions.replace) are aliases of each other.

|  |  |
| --- | --- |
| **Parameters:** | * **to\_replace** – int, long, float, string, or list. Value to be replaced. If the value is a dict, then *value* is ignored and *to\_replace* must be a mapping from column name (string) to replacement value. The value to be replaced must be an int, long, float, or string. * **value** – int, long, float, string, or list. Value to use to replace holes. The replacement value must be an int, long, float, or string. If *value* is a list or tuple, *value* should be of the same length with *to\_replace*. * **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored. |

**>>>** df4.na.replace(10, 20).show()

+----+------+-----+

| age|height| name|

+----+------+-----+

| 20| 80|Alice|

| 5| null| Bob|

|null| null| Tom|

|null| null| null|

+----+------+-----+

**>>>** df4.na.replace(['Alice', 'Bob'], ['A', 'B'], 'name').show()

+----+------+----+

| age|height|name|

+----+------+----+

| 10| 80| A|

| 5| null| B|

|null| null| Tom|

|null| null|null|

+----+------+----+

**selectExpr**(*\*expr*)

Projects a set of SQL expressions and returns a new **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)**.

This is a variant of [**select()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.select) that accepts SQL expressions.

**>>>** df.selectExpr("age \* 2", "abs(age)").collect()

[Row((age \* 2)=4, abs(age)=2), Row((age \* 2)=10, abs(age)=5)]

**sort**(*\*cols*, *\*\*kwargs*)

Returns a new **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** sorted by the specified column(s).

|  |  |
| --- | --- |
| **Parameters:** | * **cols** – list of [**Column**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Column) or column names to sort by. * **ascending** – boolean or list of boolean (default True). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of the *cols*. |

**>>>** df.sort(df.age.desc()).collect()

[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]

**>>>** df.sort("age", ascending=False).collect()

[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]

**>>>** df.orderBy(df.age.desc()).collect()

[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]

**>>> from** **pyspark.sql.functions** **import** \*

**>>>** df.sort(asc("age")).collect()

[Row(age=2, name=u'Alice'), Row(age=5, name=u'Bob')]

**>>>** df.orderBy(desc("age"), "name").collect()

[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]

**>>>** df.orderBy(["age", "name"], ascending=[0, 1]).collect()

[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]

**toDF**(*\*cols*)

Returns a new class:*DataFrame* that with new specified column names

|  |  |
| --- | --- |
| **Parameters:** | **cols** – list of new column names (string) |

**>>>** df.toDF('f1', 'f2').collect()

[Row(f1=2, f2=u'Alice'), Row(f1=5, f2=u'Bob')]

**toJSON**(*use\_unicode=True*)

Converts a **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** into a **RDD** of string.

Each row is turned into a JSON document as one element in the returned RDD.

**>>>** df.toJSON().first()

u'{"age":2,"name":"Alice"}'

*New in version 1.3.*

**toLocalIterator**()

Returns an iterator that contains all of the rows in this **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)**. The iterator will consume as much memory as the largest partition in this DataFrame.

**>>>** list(df.toLocalIterator())

[Row(age=2, name=u'Alice'), Row(age=5, name=u'Bob')]

*New in version 2.0.*

**toPandas**()

Returns the contents of this **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** as Pandas pandas.DataFrame.

This is only available if Pandas is installed and available.

**Note**

This method should only be used if the resulting Pandas’s DataFrame is expected to be small, as all the data is loaded into the driver’s memory.

**>>>** df.toPandas()

age name

0 2 Alice

1 5 Bob

**withColumn**(*colName*, *col*)

Returns a new **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** by adding a column or replacing the existing column that has the same name.

|  |  |
| --- | --- |
| **Parameters:** | * **colName** – string, name of the new column. * **col** – a [**Column**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Column) expression for the new column. |

**>>>** df.withColumn('age2', df.age + 2).collect()

[Row(age=2, name=u'Alice', age2=4), Row(age=5, name=u'Bob', age2=7)]

**withColumnRenamed**(*existing*, *new*)

Returns a new **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** by renaming an existing column. This is a no-op if schema doesn’t contain the given column name.

|  |  |
| --- | --- |
| **Parameters:** | * **existing** – string, name of the existing column to rename. * **col** – string, new name of the column. |

**>>>** df.withColumnRenamed('age', 'age2').collect()

[Row(age2=2, name=u'Alice'), Row(age2=5, name=u'Bob')]

**agg**(*\*exprs*)

Compute aggregates and returns the result as a **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)**.

The available aggregate functions are *avg*, *max*, *min*, *sum*, *count*.

If exprs is a single **dict** mapping from string to string, then the key is the column to perform aggregation on, and the value is the aggregate function.

Alternatively, exprs can also be a list of aggregate [**Column**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Column) expressions.

|  |  |
| --- | --- |
| **Parameters:** | **exprs** – a dict mapping from column name (string) to aggregate functions (string), or a list of [**Column**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Column). |

**>>>** gdf = df.groupBy(df.name)

**>>>** sorted(gdf.agg({"\*": "count"}).collect())

[Row(name=u'Alice', count(1)=1), Row(name=u'Bob', count(1)=1)]

**>>> from** **pyspark.sql** **import** functions **as** F

**>>>** sorted(gdf.agg(F.min(df.age)).collect())

[Row(name=u'Alice', min(age)=2), Row(name=u'Bob', min(age)=5)]

*New in version 1.3.*

**avg**(*\*cols*)

Computes average values for each numeric columns for each group.

[**mean()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.GroupedData.mean) is an alias for **[avg()](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.GroupedData.avg" \o "pyspark.sql.GroupedData.avg)**.

|  |  |
| --- | --- |
| **Parameters:** | **cols** – list of column names (string). Non-numeric columns are ignored. |

**>>>** df.groupBy().avg('age').collect()

[Row(avg(age)=3.5)]

**>>>** df3.groupBy().avg('age', 'height').collect()

[Row(avg(age)=3.5, avg(height)=82.5)]

*New in version 1.3.*

**count**()

Counts the number of records for each group.

**>>>** sorted(df.groupBy(df.age).count().collect())

[Row(age=2, count=1), Row(age=5, count=1)]

*New in version 1.3.*

**max**(*\*cols*)

Computes the max value for each numeric columns for each group.

**>>>** df.groupBy().max('age').collect()

[Row(max(age)=5)]

**>>>** df3.groupBy().max('age', 'height').collect()

[Row(max(age)=5, max(height)=85)]

*New in version 1.3.*

**mean**(*\*cols*)

Computes average values for each numeric columns for each group.

[**mean()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.GroupedData.mean) is an alias for **[avg()](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.GroupedData.avg" \o "pyspark.sql.GroupedData.avg)**.

|  |  |
| --- | --- |
| **Parameters:** | **cols** – list of column names (string). Non-numeric columns are ignored. |

**>>>** df.groupBy().mean('age').collect()

[Row(avg(age)=3.5)]

**>>>** df3.groupBy().mean('age', 'height').collect()

[Row(avg(age)=3.5, avg(height)=82.5)]

*New in version 1.3.*

**min**(*\*cols*)

Computes the min value for each numeric column for each group.

|  |  |
| --- | --- |
| **Parameters:** | **cols** – list of column names (string). Non-numeric columns are ignored. |

**>>>** df.groupBy().min('age').collect()

[Row(min(age)=2)]

**>>>** df3.groupBy().min('age', 'height').collect()

[Row(min(age)=2, min(height)=80)]

*New in version 1.3.*

**pivot**(*pivot\_col*, *values=None*)

Pivots a column of the current [[DataFrame]] and perform the specified aggregation. There are two versions of pivot function: one that requires the caller to specify the list of distinct values to pivot on, and one that does not. The latter is more concise but less efficient, because Spark needs to first compute the list of distinct values internally.

|  |  |
| --- | --- |
| **Parameters:** | * **pivot\_col** – Name of the column to pivot. * **values** – List of values that will be translated to columns in the output DataFrame. |

# Compute the sum of earnings for each year by course with each course as a separate column

**>>>** df4.groupBy("year").pivot("course", ["dotNET", "Java"]).sum("earnings").collect()

[Row(year=2012, dotNET=15000, Java=20000), Row(year=2013, dotNET=48000, Java=30000)]

# Or without specifying column values (less efficient)

**>>>** df4.groupBy("year").pivot("course").sum("earnings").collect()

[Row(year=2012, Java=20000, dotNET=15000), Row(year=2013, Java=30000, dotNET=48000)]

*New in version 1.6.*

**sum**(*\*cols*)

Compute the sum for each numeric columns for each group.

|  |  |
| --- | --- |
| **Parameters:** | **cols** – list of column names (string). Non-numeric columns are ignored. |

**>>>** df.groupBy().sum('age').collect()

[Row(sum(age)=7)]

**>>>** df3.groupBy().sum('age', 'height').collect()

[Row(sum(age)=7, sum(height)=165)]

*New in version 1.3.*

**Column**

*# 1. Select a column out of a DataFrame*

df.colName

df["colName"]

*# 2. Create from an expression*

df.colName + 1

1 / df.colName

**cast**(*dataType*)

Convert the column into type dataType.

**>>>** df.select(df.age.cast("string").alias('ages')).collect()

[Row(ages=u'2'), Row(ages=u'5')]

**>>>** df.select(df.age.cast(StringType()).alias('ages')).collect()

[Row(ages=u'2'), Row(ages=u'5')]

**getItem**(*key*)

An expression that gets an item at position ordinal out of a list, or gets an item by key out of a dict.

**>>>** df = sc.parallelize([([1, 2], {"key": "value"})]).toDF(["l", "d"])

**>>>** df.select(df.l.getItem(0), df.d.getItem("key")).show()

+----+------+

|l[0]|d[key]|

+----+------+

| 1| value|

+----+------+

**>>>** df.select(df.l[0], df.d["key"]).show()

+----+------+

|l[0]|d[key]|

+----+------+

| 1| value|

+----+------+

**isin**(*\*cols*)

A boolean expression that is evaluated to true if the value of this expression is contained by the evaluated values of the arguments.

**>>>** df[df.name.isin("Bob", "Mike")].collect()

[Row(age=5, name=u'Bob')]

**>>>** df[df.age.isin([1, 2, 3])].collect()

[Row(age=2, name=u'Alice')]

**otherwise**(*value*)

Evaluates a list of conditions and returns one of multiple possible result expressions. If **[Column.otherwise()](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.Column.otherwise" \o "pyspark.sql.Column.otherwise)** is not invoked, None is returned for unmatched conditions.

See **[pyspark.sql.functions.when()](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.functions.when" \o "pyspark.sql.functions.when)** for example usage.

|  |  |
| --- | --- |
| **Parameters:** | **value** – a literal value, or a [**Column**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Column) expression. |

**>>> from** **pyspark.sql** **import** functions **as** F

**>>>** df.select(df.name, F.when(df.age > 3, 1).otherwise(0)).show()

+-----+-------------------------------------+

| name|CASE WHEN (age > 3) THEN 1 ELSE 0 END|

+-----+-------------------------------------+

|Alice| 0|

| Bob| 1|

+-----+-------------------------------------+

**substr**(*startPos*, *length*)

Return a [**Column**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Column) which is a substring of the column.

|  |  |
| --- | --- |
| **Parameters:** | * **startPos** – start position (int or Column) * **length** – length of the substring (int or Column) |

**>>>** df.select(df.name.substr(1, 3).alias("col")).collect()

[Row(col=u'Ali'), Row(col=u'Bob')]

*class*pyspark.sql.**Row**

A row in **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)**. The fields in it can be accessed:

* like attributes (row.key)
* like dictionary values (row[key])

key in row will search through row keys.

Row can be used to create a row object by using named arguments, the fields will be sorted by names.

**>>>** row = Row(name="Alice", age=11)

**>>>** row

Row(age=11, name='Alice')

**>>>** row['name'], row['age']

('Alice', 11)

**>>>** row.name, row.age

('Alice', 11)

**>>>** 'name' **in** row

True

**>>>** 'wrong\_key' **in** row

False

Row also can be used to create another Row like class, then it could be used to create Row objects, such as

**>>>** Person = Row("name", "age")

**>>>** Person

<Row(name, age)>

**>>>** 'name' **in** Person

True

**>>>** 'wrong\_key' **in** Person

False

**>>>** Person("Alice", 11)

Row(name='Alice', age=11)

**asDict**(*recursive=False*)

Return as an dict

|  |  |
| --- | --- |
| **Parameters:** | **recursive** – turns the nested Row as dict (default: False). |

**>>>** Row(name="Alice", age=11).asDict() == {'name': 'Alice', 'age': 11}

True

**>>>** row = Row(key=1, value=Row(name='a', age=2))

**>>>** row.asDict() == {'key': 1, 'value': Row(age=2, name='a')}

True

**>>>** row.asDict(True) == {'key': 1, 'value': {'name': 'a', 'age': 2}}

True

*class*pyspark.sql.**DataFrameNaFunctions**(*df*)

Functionality for working with missing data in **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)**.

*New in version 1.4.*

**drop**(*how='any'*, *thresh=None*, *subset=None*)

Returns a new **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** omitting rows with null values. **[DataFrame.dropna()](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame.dropna" \o "pyspark.sql.DataFrame.dropna)** and **[DataFrameNaFunctions.drop()](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrameNaFunctions.drop" \o "pyspark.sql.DataFrameNaFunctions.drop)** are aliases of each other.

|  |  |
| --- | --- |
| **Parameters:** | * **how** – ‘any’ or ‘all’. If ‘any’, drop a row if it contains any nulls. If ‘all’, drop a row only if all its values are null. * **thresh** – int, default None If specified, drop rows that have less than *thresh* non-null values. This overwrites the *how* parameter. * **subset** – optional list of column names to consider. |

**>>>** df4.na.drop().show()

+---+------+-----+

|age|height| name|

+---+------+-----+

| 10| 80|Alice|

+---+------+-----+

*New in version 1.3.1.*

**fill**(*value*, *subset=None*)

Replace null values, alias for na.fill(). [**DataFrame.fillna()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.fillna) and [**DataFrameNaFunctions.fill()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrameNaFunctions.fill) are aliases of each other.

|  |  |
| --- | --- |
| **Parameters:** | * **value** – int, long, float, string, or dict. Value to replace null values with. If the value is a dict, then *subset* is ignored and *value* must be a mapping from column name (string) to replacement value. The replacement value must be an int, long, float, or string. * **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored. |

**>>>** df4.na.fill(50).show()

+---+------+-----+

|age|height| name|

+---+------+-----+

| 10| 80|Alice|

| 5| 50| Bob|

| 50| 50| Tom|

| 50| 50| null|

+---+------+-----+

**>>>** df4.na.fill({'age': 50, 'name': 'unknown'}).show()

+---+------+-------+

|age|height| name|

+---+------+-------+

| 10| 80| Alice|

| 5| null| Bob|

| 50| null| Tom|

| 50| null|unknown|

+---+------+-------+

*New in version 1.3.1.*

**replace**(*to\_replace*, *value*, *subset=None*)

Returns a new **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** replacing a value with another value. [**DataFrame.replace()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.replace) and [**DataFrameNaFunctions.replace()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrameNaFunctions.replace) are aliases of each other.

|  |  |
| --- | --- |
| **Parameters:** | * **to\_replace** – int, long, float, string, or list. Value to be replaced. If the value is a dict, then *value* is ignored and *to\_replace* must be a mapping from column name (string) to replacement value. The value to be replaced must be an int, long, float, or string. * **value** – int, long, float, string, or list. Value to use to replace holes. The replacement value must be an int, long, float, or string. If *value* is a list or tuple, *value* should be of the same length with *to\_replace*. * **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored. |

**>>>** df4.na.replace(10, 20).show()

+----+------+-----+

| age|height| name|

+----+------+-----+

| 20| 80|Alice|

| 5| null| Bob|

|null| null| Tom|

|null| null| null|

+----+------+-----+

**>>>** df4.na.replace(['Alice', 'Bob'], ['A', 'B'], 'name').show()

+----+------+----+

| age|height|name|

+----+------+----+

| 10| 80| A|

| 5| null| B|

|null| null| Tom|

|null| null|null|

+----+------+----+

**replace**(*to\_replace*, *value*, *subset=None*)

Returns a new **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)** replacing a value with another value. [**DataFrame.replace()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.replace) and [**DataFrameNaFunctions.replace()**](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrameNaFunctions.replace) are aliases of each other.

|  |  |
| --- | --- |
| **Parameters:** | * **to\_replace** – int, long, float, string, or list. Value to be replaced. If the value is a dict, then *value* is ignored and *to\_replace* must be a mapping from column name (string) to replacement value. The value to be replaced must be an int, long, float, or string. * **value** – int, long, float, string, or list. Value to use to replace holes. The replacement value must be an int, long, float, or string. If *value* is a list or tuple, *value* should be of the same length with *to\_replace*. * **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored. |

**>>>** df4.na.replace(10, 20).show()

+----+------+-----+

| age|height| name|

+----+------+-----+

| 20| 80|Alice|

| 5| null| Bob|

|null| null| Tom|

|null| null| null|

+----+------+-----+

**>>>** df4.na.replace(['Alice', 'Bob'], ['A', 'B'], 'name').show()

+----+------+----+

| age|height|name|

+----+------+----+

| 10| 80| A|

| 5| null| B|

|null| null| Tom|

|null| null|null|

+----+------+----+

**READ**

**>>>** df1 = spark.read.json('python/test\_support/sql/people.json')

**>>>** df1.dtypes

[('age', 'bigint'), ('name', 'string')]

**>>>** rdd = sc.textFile('python/test\_support/sql/people.json')

**>>>** df2 = spark.read.json(rdd)

**>>>** df2.dtypes

[('age', 'bigint'), ('name', 'string')]

**table**(*tableName*)

Returns the specified table as a **[DataFrame](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrame" \o "pyspark.sql.DataFrame)**.

|  |  |
| --- | --- |
| **Parameters:** | **tableName** – string, name of the table. |

**>>>** df = spark.read.parquet('python/test\_support/sql/parquet\_partitioned')

**>>>** df.createOrReplaceTempView('tmpTable')

**>>>** spark.read.table('tmpTable').dtypes

[('name', 'string'), ('year', 'int'), ('month', 'int'), ('day', 'int')]

*class*pyspark.sql.types.**ArrayType**(*elementType*, *containsNull=True*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#ArrayType)

Array data type.

|  |  |
| --- | --- |
| **Parameters:** | * **elementType** – **[DataType](http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html" \l "pyspark.sql.types.DataType" \o "pyspark.sql.types.DataType)** of each element in the array. * **containsNull** – boolean, whether the array can contain null (None) values. |

**fromInternal**(*obj*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.fromInternal)

Converts an internal SQL object into a native Python object.

**json**()[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.json)

**jsonValue**()[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.jsonValue)

**needConversion**()[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.needConversion)

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for ArrayType/MapType/StructType.

**simpleString**()[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.simpleString)

**toInternal**(*obj*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.toInternal)

Converts a Python object into an internal SQL object.

**fromInternal**(*obj*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.fromInternal)

Converts an internal SQL object into a native Python object.

**json**()[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.json)

**jsonValue**()[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.jsonValue)

**needConversion**()[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.needConversion)

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for ArrayType/MapType/StructType.

**simpleString**()[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.simpleString)

**toInternal**(*obj*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/types.html#DataType.toInternal)

Converts a Python object into an internal SQL object.

pyspark.sql.functions.**array**(*\*cols*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/functions.html#array)

Creates a new array column.

|  |  |
| --- | --- |
| **Parameters:** | **cols** – list of column names (string) or list of **Column**expressions that have the same data type. |

**>>>** df.select(array('age', 'age').alias("arr")).collect()

[Row(arr=[2, 2]), Row(arr=[5, 5])]

**>>>** df.select(array([df.age, df.age]).alias("arr")).collect()

[Row(arr=[2, 2]), Row(arr=[5, 5])]

*New in version 1.4.*

pyspark.sql.functions.**array\_contains**(*col*, *value*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/functions.html#array_contains)

Collection function: returns True if the array contains the given value. The collection elements and value must be of the same type.

|  |  |
| --- | --- |
| **Parameters:** | * **col** – name of column containing array * **value** – value to check for in array |

**>>>** df = spark.createDataFrame([(["a", "b", "c"],), ([],)], ['data'])

**>>>** df.select(array\_contains(df.data, "a")).collect()

[Row(array\_contains(data, a)=True), Row(array\_contains(data, a)=False)]

*New in version 1.5.*

pyspark.sql.functions.**explode**(*col*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/functions.html#explode)

Returns a new row for each element in the given array or map.

**>>> from** **pyspark.sql** **import** Row

**>>>** eDF = spark.createDataFrame([Row(a=1, intlist=[1,2,3], mapfield={"a": "b"})])

**>>>** eDF.select(explode(eDF.intlist).alias("anInt")).collect()

[Row(anInt=1), Row(anInt=2), Row(anInt=3)]

**>>>** eDF.select(explode(eDF.mapfield).alias("key", "value")).show()

+---+-----+

|key|value|

+---+-----+

| a| b|

+---+-----+

pyspark.sql.functions.**get\_json\_object**(*col*, *path*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/functions.html#get_json_object)

Extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It will return null if the input json string is invalid.

|  |  |
| --- | --- |
| **Parameters:** | * **col** – string column in json format * **path** – path to the json object to extract |

**>>>** data = [("1", '''{"f1": "value1", "f2": "value2"}'''), ("2", '''{"f1": "value12"}''')]

**>>>** df = spark.createDataFrame(data, ("key", "jstring"))

**>>>** df.select(df.key, get\_json\_object(df.jstring, '$.f1').alias("c0"), \

**...**  get\_json\_object(df.jstring, '$.f2').alias("c1") ).collect()

[Row(key=u'1', c0=u'value1', c1=u'value2'), Row(key=u'2', c0=u'value12', c1=None)]

pyspark.sql.functions.**instr**(*str*, *substr*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/functions.html#instr)

Locate the position of the first occurrence of substr column in the given string. Returns null if either of the arguments are null.

**Note**

The position is not zero based, but 1 based index. Returns 0 if substr could not be found in str.

**>>>** df = spark.createDataFrame([('abcd',)], ['s',])

**>>>** df.select(instr(df.s, 'b').alias('s')).collect()

[Row(s=2)]

*New in version 1.6.*

pyspark.sql.functions.**json\_tuple**(*col*, *\*fields*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/functions.html#json_tuple)

Creates a new row for a json column according to the given field names.

|  |  |
| --- | --- |
| **Parameters:** | * **col** – string column in json format * **fields** – list of fields to extract |

**>>>** data = [("1", '''{"f1": "value1", "f2": "value2"}'''), ("2", '''{"f1": "value12"}''')]

**>>>** df = spark.createDataFrame(data, ("key", "jstring"))

**>>>** df.select(df.key, json\_tuple(df.jstring, 'f1', 'f2')).collect()

[Row(key=u'1', c0=u'value1', c1=u'value2'), Row(key=u'2', c0=u'value12', c1=None)]

*New in version 1.6.*

pyspark.sql.functions.**posexplode**(*col*)[[source]](http://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/functions.html#posexplode)

Returns a new row for each element with position in the given array or map.

**>>> from** **pyspark.sql** **import** Row

**>>>** eDF = spark.createDataFrame([Row(a=1, intlist=[1,2,3], mapfield={"a": "b"})])

**>>>** eDF.select(posexplode(eDF.intlist)).collect()

[Row(pos=0, col=1), Row(pos=1, col=2), Row(pos=2, col=3)]

**>>>** eDF.select(posexplode(eDF.mapfield)).show()

+---+---+-----+

|pos|key|value|

+---+---+-----+

| 0| a| b|

+---+---+-----+