**pyspark.pandas.[input/output]**

|  |
| --- |
| pyspark.pandas.range(start: int, end: Optional[int] = None, step: int = 1, num\_partitions: Optional[int] = None) → pyspark.pandas.frame.DataFrame  Create a DataFrame with some range of numbers.  >>>ps.range(start = 100, end = 200, step = 20)  id  0 100  1 120  2 140  3 160  4 180 |

|  |
| --- |
| pyspark.pandas.read\_table(name: str, index\_col: Union[str, List[str], None] = None) → pyspark.pandas.frame.DataFrame  Read a Spark table and return a DataFrame.  ps.range(1).to\_table('%s.my\_table' % db)  ps.read\_table('%s.my\_table' % db) |

|  |
| --- |
| pyspark.pandas.DataFrame.to\_table(name: str, format: Optional[str] = None, mode: str = 'w', partition\_cols: Union[str, List[str], None] = None, index\_col: Union[str, List[str], None] = None, \*\*options: Any) → None  Write the DataFrame into a Spark table. DataFrame.spark.to\_table() is an alias of DataFrame.to\_table().  df = ps.DataFrame(dict(  date=list(pd.date\_range('2012-1-1 12:00:00', periods=3, freq='M')),  country=['KR', 'US', 'JP'],  code=[1, 2 ,3]), columns=['date', 'country', 'code'])  df.to\_table('%s.my\_table' % db, partition\_cols='date') |

|  |
| --- |
| pyspark.pandas.read\_deltapath: str, version: Optional[str] = None, timestamp: Optional[str] = None, index\_col: Union[str, List[str], None] = None, \*\*options: Any) → pyspark.pandas.frame.DataFrame  Read a Delta Lake table on some file system and return a DataFrame.  If the Delta Lake table is already stored in the catalog (aka the metastore), use ‘read\_table’.  ps.range(1).to\_delta('%s/read\_delta/foo' % path)  ps.read\_delta('%s/read\_delta/foo' % path) |

|  |
| --- |
| pyspark.pandas.DataFrame.to\_delta(path: str, mode: str = 'w', partition\_cols: Union[str, List[str], None] = None, index\_col: Union[str, List[str], None] = None, \*\*options: OptionalPrimitiveType) → None  Write the DataFrame out as a Delta Lake table.  df = ps.DataFrame(dict(  date=list(pd.date\_range('2012-1-1 12:00:00', periods=3, freq='M')),  country=['KR', 'US', 'JP'],  code=[1, 2 ,3]), columns=['date', 'country', 'code'])  df.to\_delta('%s/to\_delta/foo' % path, partition\_cols='date') |

|  |
| --- |
| pyspark.pandas.read\_parquet(path: str, columns: Optional[List[str]] = None, index\_col: Optional[List[str]] = None, pandas\_metadata: bool = False, \*\*options: Any) → pyspark.pandas.frame.DataFrame  Load a parquet object from the file path, returning a DataFrame.  ps.range(1).to\_parquet('%s/read\_spark\_io/data.parquet' % path)  ps.read\_parquet('%s/read\_spark\_io/data.parquet' % path, columns=['id']) |

|  |
| --- |
| pyspark.pandas.DataFrame.to\_parquet(path: str, mode: str = 'w', partition\_cols: Union[str, List[str], None] = None, compression: Optional[str] = None, index\_col: Union[str, List[str], None] = None, \*\*options: Any) → None  Write the DataFrame out as a Parquet file or directory.  df = ps.DataFrame(dict(  date=list(pd.date\_range('2012-1-1 12:00:00', periods=3, freq='M')),  country=['KR', 'US', 'JP'],  code=[1, 2 ,3]), columns=['date', 'country', 'code'])  df.to\_parquet('%s/to\_parquet/foo.parquet' % path, partition\_cols='date') |

|  |
| --- |
| pyspark.pandas.read\_orc |

|  |
| --- |
| pyspark.pandas.DataFrame.to\_orc |

|  |
| --- |
| pyspark.pandas.read\_spark\_io(path: Optional[str] = None, format: Optional[str] = None, schema: Union[str, StructType] = None, index\_col: Union[str, List[str], None] = None, \*\*options: Any) → pyspark.pandas.frame.DataFrame  Load a DataFrame from a Spark data source.  ps.range(1).to\_spark\_io('%s/read\_spark\_io/data.parquet' % path)  ps.read\_spark\_io(  '%s/read\_spark\_io/data.parquet' % path, format='parquet', schema='id long') |

|  |
| --- |
| pyspark.pandas.DataFrame.to\_spark\_io  df = ps.DataFrame(dict(  date=list(pd.date\_range('2012-1-1 12:00:00', periods=3, freq='M')),  country=['KR', 'US', 'JP'],  code=[1, 2 ,3]), columns=['date', 'country', 'code'])  df.to\_spark\_io(path='%s/to\_spark\_io/foo.json' % path, format='json') |

|  |
| --- |
| pyspark.pandas.read\_csv(path: str, sep: str = ',', header: Union[str, int, None] = 'infer', names: Union[str, List[str], None] = None, index\_col: Union[str, List[str], None] = None, usecols: Union[List[int], List[str], Callable[[str], bool], None] = None, squeeze: bool = False, mangle\_dupe\_cols: bool = True, dtype: Union[str, numpy.dtype, pandas.core.dtypes.base.ExtensionDtype, Dict[str, Union[str, numpy.dtype, pandas.core.dtypes.base.ExtensionDtype]], None] = None, nrows: Optional[int] = None, parse\_dates: bool = False, quotechar: Optional[str] = None, escapechar: Optional[str] = None, comment: Optional[str] = None, \*\*options: Any) → Union[pyspark.pandas.frame.DataFrame, pyspark.pandas.series.Series]  **path*str***  The path string storing the CSV file to be read.  **sep*str, default ‘,’***  Delimiter to use. Must be a single character.  **header*int, default ‘infer’***  Whether to to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to *header=0* and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to *header=None*. Explicitly pass *header=0* to be able to replace existing names  **names*str or array-like, optional***  List of column names to use. If file contains no header row, then you should explicitly pass *header=None*. Duplicates in this list will cause an error to be issued. If a string is given, it should be a DDL-formatted string in Spark SQL, which is preferred to avoid schema inference for better performance.  **index\_col: str or list of str, optional, default: None**  Index column of table in Spark.  **usecols*list-like or callable, optional***  Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in names or inferred from the document header row(s). If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to *True*.  **squeeze*bool, default False***  If the parsed data only contains one column then return a Series.  **mangle\_dupe\_cols*bool, default True***  Duplicate columns will be specified as ‘X0’, ‘X1’, … ‘XN’, rather than ‘X’ … ‘X’. Passing in False will cause data to be overwritten if there are duplicate names in the columns. Currently only *True* is allowed.  **dtype*Type name or dict of column -> type, default None***  Data type for data or columns. E.g. {‘a’: np.float64, ‘b’: np.int32} Use str or object together with suitable na\_values settings to preserve and not interpret dtype.  **nrows*int, default None***  Number of rows to read from the CSV file.  **parse\_dates*boolean or list of ints or names or list of lists or dict, default False.***  Currently only *False* is allowed.  **quotechar*str (length 1), optional***  The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.  **escapechar*str (length 1), default None***  One-character string used to escape delimiter  **comment: str, optional**  Indicates the line should not be parsed.  **options*dict***  All other options passed directly into Spark’s data source. |

|  |
| --- |
| pyspark.pandas.DataFrame.to\_csv  df = ps.DataFrame(dict(  date=list(pd.date\_range('2012-1-1 12:00:00', periods=3, freq='M')),  country=['KR', 'US', 'JP'],  code=[1, 2 ,3]), columns=['date', 'country', 'code'])  df.sort\_values(by="date")  print(df.to\_csv())  df.cummax().to\_csv(path=r'%s/to\_csv/foo.csv' % path, num\_files=1)  ps.read\_csv(  path=r'%s/to\_csv/foo.csv' % path  ).sort\_values(by="date") |

|  |
| --- |
| pyspark.pandas.read\_clipboard  pyspark.pandas.DataFrame.to\_clipboard |

|  |
| --- |
| pyspark.pandas.read\_excel  pyspark.pandas.DataFrame.to\_excel  df1.to\_excel("output.xlsx",  sheet\_name='Sheet\_name\_1')  with pd.ExcelWriter('output.xlsx') as writer:  df1.to\_excel(writer, sheet\_name='Sheet\_name\_1')  df2.to\_excel(writer, sheet\_name='Sheet\_name\_2') |

|  |
| --- |
| pyspark.pandas.read\_json  pyspark.pandas.DataFrame.to\_json  df['col 1'].to\_json(path=r'%s/to\_json/foo.json' % path, num\_files=1, index\_col="index")  ps.read\_json(  path=r'%s/to\_json/foo.json' % path, index\_col="index"  ).sort\_values(by="col 1") |

|  |
| --- |
| pyspark.pandas.read\_html  pyspark.pandas.DataFrame.to\_html |

|  |
| --- |
| pyspark.pandas.read\_sql\_table  ps.read\_sql\_table('table\_name', 'jdbc:postgresql:db\_name')  pyspark.pandas.read\_sql\_query  ps.read\_sql\_query('SELECT \* FROM table\_name', 'jdbc:postgresql:db\_name')  pyspark.pandas.read\_sql  ps.read\_sql('table\_name', 'jdbc:postgresql:db\_name')  ps.read\_sql('SELECT \* FROM table\_name', 'jdbc:postgresql:db\_name') |

**pyspark.pandas.[general\_functions]**

|  |
| --- |
| pyspark.pandas.reset\_option(key: str) → None  Reset one option to their default value.  Pass “all” as argument to reset all options. |

|  |
| --- |
| pyspark.pandas.get\_option(key: str, default: Union[Any, pyspark.\_globals.\_NoValueType] = <no value>) → Any  Retrieves the value of the specified option. |

|  |
| --- |
| pyspark.pandas.set\_option(key: str, value: Any) → None  Sets the value of the specified option. |

|  |
| --- |
| pyspark.pandas.option\_context(\*args: Any) → Iterator[None]  Context manager to temporarily set options in the with statement context.  You need to invoke as option\_context(pat, val, [(pat, val), ...]).  >>>with option\_context('display.max\_rows', 10, 'compute.max\_rows', 5):  print(get\_option('display.max\_rows'), get\_option('compute.max\_rows'))  10 5  >>>print(get\_option('display.max\_rows'), get\_option('compute.max\_rows'))  1000 1000 |

|  |
| --- |
| pyspark.pandas.melt  Unpivot a DataFrame from wide format to long format, optionally leaving identifier variables set.  This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (id\_vars), while all other columns, considered measured variables (value\_vars), are “unpivoted” to the row axis, leaving just two non-identifier columns, ‘variable’ and ‘value’.  ps.melt(df)  df.melt(id\_vars='A')  df.melt(value\_vars='A')  ps.melt(df, id\_vars=['A', 'B'])  df.melt(id\_vars=['A'], value\_vars=['C']) |

|  |
| --- |
| pyspark.pandas.merge  Merge DataFrame objects with a database-style join.  Returns: A DataFrame of the two merged objects.  merged = ps.merge(df1, df2, left\_on='lkey', right\_on='rkey')  merged.sort\_values(by=['lkey', 'value\_x', 'rkey', 'value\_y'])  left\_psdf = ps.DataFrame({'A': [1, 2]})  right\_psdf = ps.DataFrame({'B': ['x', 'y']}, index=[1, 2])  ps.merge(left\_psdf, right\_psdf, left\_index=True, right\_index=True).sort\_index() |

|  |
| --- |
| pyspark.pandas.get\_dummies  Convert categorical variable into dummy/indicator variables, also known as one hot encoding.  s = ps.Series(list('abca'))  ps.get\_dummies(s)  ps.get\_dummies(ps.Series(list('abcaa'))) |

|  |
| --- |
| pyspark.pandas.concat  Concatenate pandas-on-Spark objects along a particular axis with optional set logic along the other axes.  from pyspark.pandas.config import set\_option, reset\_option  set\_option("compute.ops\_on\_diff\_frames", True)  s1 = ps.Series(['a', 'b'])  s2 = ps.Series(['c', 'd'])  ps.concat([s1, s2])  ps.concat([s1, s2], ignore\_index=True)  ps.concat([df2, s1]) |

|  |
| --- |
| pyspark.pandas.sql (query: str, index\_col: Union[str, List[str], None] = None, globals: Optional[Dict[str, Any]] = None, locals: Optional[Dict[str, Any]] = None, \*\*kwargs: Any) → pyspark.pandas.frame.DataFrame  psdf = ps.DataFrame({"A": [1, 2, 3], "B":[4, 5, 6]}, index=['a', 'b', 'c'])  psdf\_reset\_index = psdf.reset\_index()  ps.sql("SELECT \* FROM {psdf\_reset\_index}", index\_col="index")  psdf = ps.DataFrame(  {"A": [1, 2, 3], "B": [4, 5, 6]},  index=pd.MultiIndex.from\_tuples(  [("a", "b"), ("c", "d"), ("e", "f")], names=["index1", "index2"]  ),  )  psdf\_reset\_index = psdf.reset\_index()  ps.sql("SELECT \* FROM {psdf\_reset\_index}", index\_col=["index1", "index2"]) |

|  |
| --- |
| pyspark.pandas.broadcast (obj: pyspark.pandas.frame.DataFrame) → pyspark.pandas.frame.DataFrame  Marks a DataFrame as small enough for use in broadcast joins.  df1 = ps.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],  'value': [1, 2, 3, 5]},  columns=['lkey', 'value']).set\_index('lkey')  df2 = ps.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],  'value': [5, 6, 7, 8]},  columns=['rkey', 'value']).set\_index('rkey')  merged = df1.merge(ps.broadcast(df2), left\_index=True, right\_index=True)  merged.spark.explain()  == Physical Plan ==  ...BroadcastHashJoin... |

|  |
| --- |
| pyspark.pandas.to\_numeric(arg)  Convert argument to a numeric type.  psser = ps.Series(['1.0', '2', '-3'])  ps.to\_numeric(psser) |

|  |
| --- |
| pyspark.pandas.isna(obj)  Detect missing values for an array-like object.  This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays).  ps.isna('dog')  ps.isna(np.nan)  ps.isna(array)  ps.isna(df) |

|  |
| --- |
| pyspark.pandas.isnull(obj)  Detect missing values for an array-like object.  This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays).  ps.isnull(df.b) |

|  |
| --- |
| pyspark.pandas.notna(obj)  Detect existing (non-missing) values.  Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. NA values, such as None or numpy.NaN, get mapped to False values.  ps.notna(ser) |

|  |
| --- |
| pyspark.pandas.notnull(obj) |

|  |
| --- |
| pyspark.pandas.to\_datetime(arg, errors: str = 'raise', format: Optional[str] = None, unit: Optional[str] = None, infer\_datetime\_format: bool = False, origin: str = 'unix')  df = ps.DataFrame({'year': [2015, 2016],  'month': [2, 3],  'day': [4, 5]})  ps.to\_datetime(df)  >>>ps.to\_datetime('13000101', format='%Y%m%d', errors='ignore')  datetime.datetime(1300, 1, 1, 0, 0)  >>>ps.to\_datetime('13000101', format='%Y%m%d', errors='coerce')  NaT  s = ps.Series(['3/11/2000', '3/12/2000', '3/13/2000'] \* 1000)  s.head()  import timeit  timeit.timeit(  lambda: repr(ps.to\_datetime(s, infer\_datetime\_format=True)),  number = 1) |

|  |
| --- |
| pyspark.pandas.date\_range(start: Union[str, Any] = None, end: Union[str, Any] = None, periods: Optional[int] = None, freq: Union[str, pandas.\_libs.tslibs.offsets.DateOffset, None] = None, tz: Union[str, datetime.tzinfo, None] = None, normalize: bool = False, name: Optional[str] = None, closed: Optional[str] = None, \*\*kwargs: Any) → pyspark.pandas.indexes.datetimes.DatetimeIndex  ps.date\_range(start='1/1/2018', end='1/08/2018')  ps.date\_range(start='1/1/2018', periods=8)  ps.date\_range(end='1/1/2018', periods=8)  ps.date\_range(  start='2018-04-24', end='2018-04-27', periods=3  ) |