Pandas vs PySpark Cheatsheet (Complete with Solutions)

This guide compares common data wrangling operations in **pandas (Python)** and **PySpark (Spark DataFrame API)** with detailed examples and **solved exercises**. Assume a running SparkSession:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("cheatsheet").getOrCreate()
```

1. Reading and Inspecting Data

Task	pandas (Python)	PySpark (Python)
Read CSV	`pd.read_csv("file.csv")`	`spark.read.option("header",True).csv("file.csv", inferSchema=True)`
Read Parquet	`pd.read_parquet("file.parquet")`	`spark.read.parquet("file.parquet")`
Read JSON	`pd.read_json("file.json", lines=True)`	`spark.read.json("file.json")`
Quick look	`df.head(), df.info(), df.describe()`	`df.show(5), df.printSchema(), df.describe().show()`
Set NA on read	`pd.read_csv(..., na_values=["","NA"])`	`spark.read.option("nullValue","").csv(..., header=True)`

Exercise (Solved): Load `sales.csv`, show first 10, dtypes/schema, summary; set missing to 0 on import.

Pandas

```
import pandas as pd
df = pd.read_csv("sales.csv", na_values=["", "NA", "NaN"]).fillna(0)
print(df.head(10))
print(df.dtypes)
print(df.describe(include="all"))
```

PySpark

```
from pyspark.sql import functions as F
df = (spark.read.option("header", True)
              .option("inferSchema", True)
              .option("nullValue", "")
              .csv("sales.csv"))
df = df.fillna(0)
df.show(10, truncate=False)
df.printSchema()
df.describe().show()
```

2. Selecting Columns

Method	pandas	PySpark
Select by name	`df[["col1","col2"]]`	`df.select("col1","col2")`
Range	`df.loc[:, "c1":"c3"]`	(No label range) use explicit list
By position	`df.iloc[:, 0:3]`	`df.select(df.columns[:3])`
Drop	`df.drop(["c1"], axis=1)`	`df.drop("c1")`
By pattern	`df.filter(regex="^col")`	`df.select([c for c in df.columns if c.startswith("col")])`

Exercise (Solved): Keep `address*`, drop names containing `temp`, then 1st 3 + last col.

Pandas

```
addr = df.filter(regex="^address")
df_no_temp = df.drop(columns=df.filter(regex="temp").columns)
subset = pd.concat([df.iloc[:, 0:3], df.iloc[:, -1]], axis=1)
```

PySpark

```
cols = df.columns
addr = df.select([c for c in cols if c.startswith("address")])
df_no_temp = df.select([c for c in cols if "temp" not in c])
subset = df.select(*cols[:3], cols[-1])
```

3. Filtering Rows

Method	pandas	PySpark
Single	`df[df["age"]>30]`	`df.filter(F.col("age")>30)`
Multiple	`(df["a"]>1)&(df["b"]=="X")`	`df.filter((F.col("a")>1)& (F.col("b")=="X"))`
SQL expr	`df.query("a>1 & b=='X'")`	`df.filter("a > 1 AND b = 'X'")`
By position	`df.iloc[0:10]`	`df.limit(10)`

Exercise (Solved): Complex filter + top 20 by salary.
Pandas

```
f = ((df["age"]>35) & (df["salary"]>50000)) | \
    ((df["department"]=="Engineering") & (df["years_experience"]>=3))
top20 = df.loc[f].sort_values("salary", ascending=False).head(20)
```

PySpark

```
f = ((F.col("age")>35) & (F.col("salary")>50000)) | \
    ((F.col("department")=="Engineering") & (F.col("years_experience")>=3))
top20 = df.filter(f).orderBy(F.desc("salary")).limit(20)
```

4. Sorting/Arranging

Operation	pandas	PySpark
Basic sort	`df.sort_values("col")`	`df.orderBy("col")`
Desc	`df.sort_values("col",False)`	`df.orderBy(F.desc("col"))`
Multi	`df.sort_values(["c1","c2"] )`	`df.orderBy("c1","c2")`
Mixed	`ascending=[True,False]`	`df.orderBy(F.asc("c1"),F.desc("c2"))`

Exercise (Solved): Sort by category↑, price↓, rating↓; top 5 per category.
Pandas

```
sorted_df = df.sort_values(["category","price","rating"],
                           ascending=[True, False, False])
top5_each = sorted_df.groupby("category", group_keys=False).head(5)
```

PySpark

```
from pyspark.sql.window import Window
w = Window.partitionBy("category").orderBy(F.desc("price"), F.desc("rating"))
ranked = df.orderBy("category").withColumn("rn", F.row_number().over(w))
top5_each = ranked.filter(F.col("rn")<=5).drop("rn")
```

5. Handling Missing Values

Operation	pandas	PySpark
Check	`df.isna().sum()`	`[(c, df.filter(F.col(c).isNull()).count())for c in df.columns]`
Drop	`df.dropna()`	`df.na.drop()`
Fill value	`df["c"].fillna(0)`	`df.fillna({"c":0})`
FFill/BFill	`df["c"].ffill()/bfill()`	`F.last("c", True)over window (use Window)`

Exercise (Solved): Time series: fill sales with prev day; price with category mean; drop rows where both quantity and price missing.

Pandas

```
df = df.sort_values("date")
df["sales"] = df["sales"].ffill()
df["price"] = df.groupby("category")["price"] \
                .transform(lambda s: s.fillna(s.mean()))
df = df.dropna(subset=["quantity","price"], how="all")
```

PySpark

```
from pyspark.sql.window import Window
df = df.orderBy("date")
w = Window.orderBy("date").rowsBetween(Window.unboundedPreceding, 0)
df = df.withColumn("sales", F.last("sales", ignorenulls=True).over(w))

wcat = Window.partitionBy("category")
cat_mean = F.avg("price").over(wcat)
df = df.withColumn("price", F.when(F.col("price").isNull(), cat_mean).otherwise
    (F.col("price")))
df = df.filter(~(F.col("quantity").isNull() & F.col("price").isNull()))
```

6. Creating New Columns

Operation	pandas	PySpark
Calc	`df["bmi"]=df["w"]/df["h"]**2`	`df.withColumn("bmi", F.col("w")/F.col("h")**2)`
If-else	`np.where(cond,a,b)`	`F.when(cond,a).otherwise(b)`
Case	`np.select([...])`	`F.when(...).when(...).otherwise(...)`
Row-wise	`df.apply(f,axis=1)`	UDF / pandas_udf (use sparingly)

Exercise (Solved): Risk score + DTI and flag.

Pandas

```
import numpy as np
high = (df["income"] < 30000) | (df["age"] > 65)
medium = (df["income"] < 50000) & (df["age"].between(25,65, inclusive="both"))
df["risk"] = np.select([high, medium], ["High","Medium"], default="Low")
df["dti"] = df["debt"] / df["income"].replace(0, np.nan)
df["high_dti"] = df["dti"] > 0.4
```

PySpark

```
risk = (F.when((F.col("income") < 30000) | (F.col("age") > 65), "High")
          .when((F.col("income") < 50000) & (F.col("age").between(25,65)), "
              Medium")
          .otherwise("Low"))
df = df.withColumn("risk", risk)
```

```
df = df.withColumn("dti", F.col("debt")/F.when(F.col("income")==0, F.lit(None))
    .otherwise(F.col("income")))
df = df.withColumn("high_dti", F.col("dti") > 0.4)
```

7. Renaming Columns

Operation	pandas	PySpark
Single	`df.rename(columns={"old":"new"})`	`df.withColumnRenamed("old","new")`
Multiple	`df.rename(columns={...})`	reduce with `withColumnRenamed` in a loop
All	`df.columns = [... ]`	`df.toDF(*new_names)`

Exercise (Solved): Standardize names to snake_case; unify variants.

Pandas

```
import re
def to_snake(s):
    s = re.sub('(.)([A-Z][a-z]+)', r'\1_\2', s)
    s = re.sub('([a-z0-9])([A-Z])', r'\1_\2', s)
    s = s.replace(".", "_").replace("-", "_")
    s = re.sub(r'__+', '_', s).lower()
    return s
df = df.rename(columns=lambda c: to_snake(c))
df = df.rename(columns={"fname":"first_name", "firstname":"first_name"})
```

PySpark

```
import re
def to_snake(s):
    s = re.sub('(.)([A-Z][a-z]+)', r'\1_\2', s)
    s = re.sub('([a-z0-9])([A-Z])', r'\1_\2', s)
    s = s.replace(".", "_").replace("-", "_")
    s = re.sub(r'__+', '_', s).lower()
    return s
new_names = [to_snake(c) for c in df.columns]
df = df.toDF(*new_names)
mapping = {"fname":"first_name","firstname":"first_name"}
for old, new in mapping.items():
    if old in df.columns: df = df.withColumnRenamed(old, new)
```

8. Grouping and Aggregation

Task	pandas	PySpark
Mean per group	`groupby("dept")["salary"].mean()`	`df.groupBy("dept").agg(F.mean("salary"))`
Multi stats	`agg(["mean","sum"])`	`df.groupBy("dept").agg(F.mean("s").alias("mean_s"),F.sum("s").alias("sum_s"))`
Rank in group	`groupby("id")["x"].rank()`	`F.dense_rank().over(Window.partitionBy("id").orderBy("x"))`

Exercise (Solved): For region/quarter: total sales, AOV, unique customers, rank region by total within quarter.

Pandas

```
agg = (df.groupby(["region","quarter"])
         .agg(total_sales=("sales","sum"),
              orders=("order_id","nunique"),
```

```
            unique_customers=("customer_id","nunique"),
            avg_order_value=("sales","mean"))
        .reset_index())
agg["rank_in_qtr"] = agg.groupby("quarter")["total_sales"] \
                        .rank(method="dense", ascending=False).astype(int)
```

PySpark

```
from pyspark.sql.window import Window
agg = (df.groupBy("region","quarter")
         .agg(F.sum("sales").alias("total_sales"),
              F.countDistinct("order_id").alias("orders"),
              F.countDistinct("customer_id").alias("unique_customers"),
              F.avg("sales").alias("avg_order_value")))
wq = Window.partitionBy("quarter").orderBy(F.desc("total_sales"))
agg = agg.withColumn("rank_in_qtr", F.dense_rank().over(wq))
```

9. Joins

Type	pandas	PySpark
Inner	`pd.merge(df1,df2,on="id")`	`df1.join(df2,"id","inner")`
Left	`how="left"`	`df1.join(df2,"id","left")`
Right	`how="right"`	`df1.join(df2,"id","right")`
Full	`how="outer"`	`df1.join(df2,"id","outer")`
Semi/Anti	—	`left_semi / left_anti joins`

Exercise (Solved): Join customers→orders→products; customers never ordered; products never sold; revenue per customer category.

Pandas

```
cust_orders = pd.merge(customers, orders, on="customer_id", how="left")
full = pd.merge(cust_orders, products, on="product_id", how="left")
full["revenue"] = full["quantity"] * full["price"]

never_ordered = customers[~customers["customer_id"].isin(orders["customer_id"])
    ]

sold_products = orders["product_id"].dropna().unique()
never_sold = products[~products["product_id"].isin(sold_products)]

rev_by_cat = (full.groupby("customer_category")["revenue"]
                .sum().reset_index())
```

PySpark

```
full = (customers.join(orders, "customer_id", "left")
                 .join(products, "product_id", "left")
                 .withColumn("revenue", F.col("quantity")*F.col("price")))

never_ordered = customers.join(orders.select("customer_id").distinct(),
                               on="customer_id", how="left_anti")

never_sold = products.join(orders.select("product_id").distinct(),
                           on="product_id", how="left_anti")

rev_by_cat = full.groupBy("customer_category").agg(F.sum("revenue").alias("
    revenue"))
```

10. Reshaping Data

Operation	pandas	PySpark
Wide→Long	`df.melt(...)`	`F.expr("stack(...)")+ select`
Long→Wide	`df.pivot(...)`	`df.groupBy(idx).pivot(col).agg(F.first("value"))`
Explode	`df.explode("col")`	`df.withColumn("col", F.explode("col"))`

Exercise (Solved): Wide (Q1..Q4) → long; compute YoY growth by region/quarter; pivot back to show growth by quarter and region.

Pandas

```
# columns: region, year, Q1, Q2, Q3, Q4
long = df.melt(id_vars=["region","year"],
               value_vars=["Q1","Q2","Q3","Q4"],
               var_name="quarter", value_name="sales")
long = long.sort_values(["region","quarter","year"])
long["yoy_growth"] = long.groupby(["region","quarter"])["sales"].pct_change()

growth = (long.pivot_table(index="region", columns="quarter",
                           values="yoy_growth", aggfunc="last")
               .reset_index())
```

PySpark

```
# columns: region, year, Q1, Q2, Q3, Q4
expr = "stack(4,␣'Q1',␣Q1,␣'Q2',␣Q2,␣'Q3',␣Q3,␣'Q4',␣Q4)␣as␣(quarter,␣sales)"
long = (df.select("region","year", F.expr(expr))
          .orderBy("region","quarter","year"))

from pyspark.sql.window import Window
w = Window.partitionBy("region","quarter").orderBy("year")
prev = F.lag("sales").over(w)
long = long.withColumn("yoy_growth", (F.col("sales")-prev)/prev)

growth = (long.groupBy("region")
               .pivot("quarter", ["Q1","Q2","Q3","Q4"])
               .agg(F.last("yoy_growth")))
```

11. String Operations

Operation	pandas	PySpark
Contains	`str.contains("a")`	`F.col("c").like("%a%")/ F.contains`
Regex extract	`str.extract("re")`	`F.regexp_extract("c","re",1)`
Replace	`str.replace("a","b")`	`F.regexp_replace("c","a","b")`
Split	`str.split("-",expand=True)`	`F.split("c","-")+ withColumn from array`
Lower/Upper	`str.lower()/upper()`	`F.lower("c")/F.upper("c")`

Exercise (Solved): Clean addresses: extract ZIP, standardize state abbrev, split names, flag apartments.

Pandas

```
import re
df["zip"] = df["address"].str.extract(r'(\b\d{5}\b)')
state_map = {"Calif.":"CA", "California":"CA", "Tx":"TX"}
df["state"] = df["state"].replace(state_map).str.upper()
names = df["full_name"].str.split(r"\s+", n=1, expand=True)
```

```
df["first_name"] = names[0]; df["last_name"] = names[1]
df["has_apartment"] = df["address"].str.contains(
    r'\b(apt|apartment|unit|#)\b', case=False, na=False)
```

PySpark

```
state_map = {"Calif.":"CA", "California":"CA", "Tx":"TX"}
mapping_expr = F.create_map([F.lit(x) for kv in state_map.items() for x in kv])

df = (df.withColumn("zip", F.regexp_extract("address", r'\\b\\d{5}\\b', 0))
        .withColumn("state", F.upper(F.coalesce(mapping_expr[F.col("state")], F
            .col("state"))))
        .withColumn("split_name", F.split(F.col("full_name"), r"\\s+", 2))
        .withColumn("first_name", F.col("split_name")[0])
        .withColumn("last_name", F.col("split_name")[1])
        .withColumn("has_apartment", F.lower("address").rlike(r"\\b(apt|
            apartment|unit|#)\\b"))
        .drop("split_name"))
```

12. Duplicates and Sampling

Operation	pandas	PySpark
Drop dups	`drop_duplicates()`	`dropDuplicates()`
Sample rows	`sample(n=100)`	`sample(withReplacement=False, fraction=0.1, seed=42)`
Top-n	`nlargest(10,"col")`	`orderBy(desc("col")).limit(10)`

Exercise (Solved): Remove duplicates by email+phone, sample 20%, top 10% by purchase value.

Pandas

```
dedup = df.drop_duplicates(subset=["email","phone"])
sample20 = dedup.sample(frac=0.2, random_state=42)
total = (orders.groupby("customer_id")["purchase_value"].sum()
           .reset_index(name="total_purchase"))
q90 = total["total_purchase"].quantile(0.90)
top10pct = total[total["total_purchase"] >= q90]
```

PySpark

```
dedup = df.dropDuplicates(["email","phone"])
sample20 = dedup.sample(withReplacement=False, fraction=0.2, seed=42)

total = orders.groupBy("customer_id").agg(F.sum("purchase_value").alias("
    total_purchase"))
w = Window.orderBy("total_purchase")
# Approximate: use percentile_approx for threshold
q90 = total.agg(F.expr("percentile_approx(total_purchase, 0.90) as q")).collect
    ()[0]["q"]
top10pct = total.filter(F.col("total_purchase") >= F.lit(q90))
```

13. Concatenating/Binding

Operation	pandas	PySpark
Rows (append)	`pd.concat([df1,df2])`	`df1.unionByName(df2)`
Cols (side)	`pd.concat([df1,df2],axis=1)`	`df1.join(df2, on=key, how="inner")(` no direct col-bind)

Exercise (Solved): Combine monthly files by region, add source id, compute YTD by region and category.

Pandas

```
import glob, os, pandas as pd
files = glob.glob("data/region_*_2025-*.csv")
dfs = []
for f in files:
    d = pd.read_csv(f)
    d["source"] = os.path.basename(f)
    dfs.append(d)
master = pd.concat(dfs, ignore_index=True)
ytd = (master.groupby(["region","product_category"])["sales"]
             .sum().reset_index(name="ytd_sales"))
```

PySpark

```
import glob, os
files = glob.glob("data/region_*_2025-*.csv")
from functools import reduce
dfs = []
for f in files:
    d = (spark.read.option("header",True).option("inferSchema",True).csv(f)
                  .withColumn("source", F.lit(os.path.basename(f))))
    dfs.append(d)
master = reduce(lambda a,b: a.unionByName(b), dfs)
ytd = master.groupBy("region","product_category").agg(F.sum("sales").alias("
    ytd_sales"))
```

14. Date/Time Operations

Operation	pandas	PySpark
Parse	`pd.to_datetime(...)`	`F.to_date("col","yyyy-MM-dd")/ F.to_timestamp`
Parts	`dt.year / .month`	`F.year("d"), F.month("d")`
Diff	`(d2-d1).dt.days`	`F.datediff("d2","d1")`
Quarter	`dt.to_period("Q")`	`F.quarter("d")`

Exercise (Solved): Lifetime (first→last), quarters, inactive ¿90 days.
Pandas

```
orders["date"] = pd.to_datetime(orders["date"])
cust = (orders.groupby("customer_id")["date"]
              .agg(first="min", last="max", recent="max").reset_index())
cust["lifetime_days"] = (cust["last"] - cust["first"]).dt.days
cust["quarter"] = cust["last"].dt.to_period("Q").astype(str)

ref = pd.Timestamp.today().normalize()
inactive = cust[(ref - cust["recent"]).dt.days > 90]
```

PySpark

```
o = orders.withColumn("date", F.to_date("date"))
cust = (o.groupBy("customer_id")
         .agg(F.min("date").alias("first"),
              F.max("date").alias("last"),
              F.max("date").alias("recent")))
cust = cust.withColumn("lifetime_days", F.datediff("last","first")) \
           .withColumn("quarter", F.concat_ws("␣Q", F.year("last"), F.quarter("
               last")))
ref = F.current_date()
inactive = cust.filter(F.datediff(ref, F.col("recent")) > 90)
```

15. Applying Custom Functions

Method	pandas	PySpark
Column-wise	`apply/transform`	`use built-ins; F.expr; when needed: UDF / pandas\_udf`
Row-wise	`df.apply(f,axis=1)`	`UDF / pandas\_udf (vectorized)(avoid if possible)`
Group apply	`groupby().apply(f)`	`window functions; groupBy+agg; mapInPandas`

Exercise (Solved): RFM scoring + segment ranking.
Pandas

```
import numpy as np, pandas as pd
orders["date"] = pd.to_datetime(orders["date"])
snapshot = orders["date"].max()
rfm = (orders.groupby("customer_id")
         .agg(Recency=("date", lambda s: (snapshot - s.max()).days),
              Frequency=("order_id","nunique"),
              Monetary=("amount","sum"))
         .reset_index())
rfm["R_Score"] = pd.qcut(rfm["Recency"], 5, labels=[5,4,3,2,1]).astype(int)
rfm["F_Score"] = pd.qcut(rfm["Frequency"].rank(method="first"), 5, labels
    =[1,2,3,4,5]).astype(int)
rfm["M_Score"] = pd.qcut(rfm["Monetary"].rank(method="first"), 5, labels
    =[1,2,3,4,5]).astype(int)
rfm["RFM_Score"] = rfm[["R_Score","F_Score","M_Score"]].sum(axis=1)
rfm["segment"] = np.where(rfm["RFM_Score"]>=12, "Gold",
                   np.where(rfm["RFM_Score"]>=9, "Silver","Bronze"))
rfm = rfm.sort_values(["segment","RFM_Score"], ascending=[True,False])
```

PySpark

```
o = orders.withColumn("date", F.to_date("date"))
snapshot = o.agg(F.max("date").alias("snap")).collect()[0]["snap"]
rfm = (o.groupBy("customer_id")
         .agg(F.datediff(F.lit(snapshot), F.max("date")).alias("Recency"),
              F.countDistinct("order_id").alias("Frequency"),
              F.sum("amount").alias("Monetary")))

# Quintiles via ntile over sorted windows
from pyspark.sql.window import Window
wR = Window.orderBy(F.desc("Recency"))  # smaller Recency => higher score
wF = Window.orderBy("Frequency")
wM = Window.orderBy("Monetary")

rfm = (rfm
   .withColumn("R_Score", F.ntile(5).over(wR))
   .withColumn("F_Score", F.ntile(5).over(wF))
   .withColumn("M_Score", F.ntile(5).over(wM))
   .withColumn("RFM_Score", F.col("R_Score")+F.col("F_Score")+F.col("M_Score"))
   .withColumn("segment",
       F.when(F.col("RFM_Score")>=12,"Gold")
        .when(F.col("RFM_Score")>=9,"Silver")
        .otherwise("Bronze")))
```

16. Data Type Conversions

Operation	pandas	PySpark

To numeric	`pd.to_numeric(...,errors="coerce")`	`df.withColumn("c", F.col("c").cast("double"))`
To datetime	`pd.to_datetime(...,errors="coerce")`	`F.to_date("c","yyyy-MM-dd")/ F.to_timestamp`
Categorical	`astype("category")`	`no categorical type; use string + dictionary`

Exercise (Solved): Fix numeric-as-strings; parse mixed date formats; optimize with categories (Spark: keep as string).

Pandas

```
num_cols = ["qty","price","amount"]
for c in num_cols:
    df[c] = pd.to_numeric(df[c], errors="coerce")
df["date"] = pd.to_datetime(df["date"], errors="coerce", format=None)
mask = df["date"].isna()
df.loc[mask, "date"] = pd.to_datetime(df.loc[mask, "date_str_alt"], errors="
    coerce")
low_card = [c for c in df.columns if df[c].dtype=="object" and df[c].nunique()
    < 0.2*len(df)]
for c in low_card:
    df[c] = df[c].astype("category")
```

PySpark

```
for c in ["qty","price","amount"]:
    df = df.withColumn(c, F.col(c).cast("double"))
# try multiple date formats
df = df.withColumn("date_parsed",
        F.coalesce(F.to_date("date","yyyy-MM-dd"),
                   F.to_date("date","dd-MM-yyyy"),
                   F.to_date("date_str_alt","MM/dd/yyyy")))
# low-cardinality optimization not applicable as categorical; keep as string
```

17. Index Operations

Operation	pandas	PySpark
Reset	`reset_index()`	(no row index; always explicit columns)
Set	`set_index("col")`	(use `orderBy`/Windows with explicit cols)

Exercise (Solved): Create MultiIndex (date, product_id), reset for analysis, then restore and resample monthly per product.

Pandas

```
sales["date"] = pd.to_datetime(sales["date"])
mi = sales.set_index(["date","product_id"]).sort_index()
flat = mi.reset_index()
monthly = (flat.set_index(["date","product_id"])
               .groupby(level="product_id")
               .apply(lambda g: g.reset_index(level=1, drop=True)
                                 .resample("M")["units"].sum()))
```

PySpark

```
from pyspark.sql.functions import date_trunc
sales = sales.withColumn("date", F.to_date("date"))
monthly = (sales
  .withColumn("month", date_trunc("month", F.col("date")))
  .groupBy("product_id","month")
  .agg(F.sum("units").alias("units")))
```

18. Categorical Data

Operation	pandas	PySpark
Create	`pd.Categorical(...)`	`use strings; for ML: StringIndexer`
Order	`ordered=True`	`add order column / Window`
Recode	`replace()/cat.rename_categories`	`F.when()/mapping via create_map`

Exercise (Solved): Ordered factors, reorder categories by avg rating, recode inconsistent names.
Pandas

```
order = ["Poor","Fair","Good","Excellent"]
df["satisfaction"] = pd.Categorical(df["satisfaction"], categories=order,
    ordered=True)
avg = df.groupby("product_category")["rating"].mean()
ordered_cats = avg.sort_values().index
df["product_category"] = pd.Categorical(df["product_category"], categories=
    ordered_cats, ordered=True)
df["product_category"] = df["product_category"].replace({"elec":"Electronics",
    "Elec":"Electronics"})
```

PySpark

```
order = ["Poor","Fair","Good","Excellent"]
mapping_expr = F.create_map([F.lit(x) for kv in {"elec":"Electronics","Elec":"
    Electronics"}.items() for x in kv])
df = df.withColumn("product_category", F.coalesce(mapping_expr[F.col("
    product_category")], F.col("product_category")))
avg = df.groupBy("product_category").agg(F.avg("rating").alias("avg"))
# to "order" categories, keep avg and join later or use Window when needed
```

19. Value Counts and Frequency Tables

Operation	pandas	PySpark
Counts	`value_counts()`	`df.groupBy("col").count().orderBy(F.desc("count"))`
Proportions	`value_counts(normalize=True)`	`with total: count/total via F.sum("count")`
Crosstab	`pd.crosstab(c1,c2)`	`df.stat.crosstab("c1","c2")`

Exercise (Solved): Purchase freq by segment; payment vs customer type; popular product pairs.
Pandas

```
freq = customers["segment"].value_counts(normalize=False).reset_index(name="n")
cros = pd.crosstab(orders["payment_method"], orders["customer_type"])
from itertools import combinations
pairs = (orders.groupby("order_id")["product_id"].apply(lambda s: list(set(s)))
    )
rows = []
for prods in pairs:
    for a,b in combinations(sorted(prods), 2):
        rows.append((a,b))
pairs_df = pd.DataFrame(rows, columns=["prod_a","prod_b"])
popular_pairs = pairs_df.value_counts().reset_index(name="count").sort_values("
    count", ascending=False)
```

PySpark

```
freq = customers.groupBy("segment").count().orderBy(F.desc("count"))
cros = orders.stat.crosstab("payment_method","customer_type")
```

```
# product pairs
items = (orders.groupBy("order_id")
               .agg(F.collect_set("product_id").alias("items")))
pairs = (items.withColumn("pairs",
            F.expr("transform(sequence(0,␣size(items)-1),␣x␣->␣named_struct('a
                ',␣items[x],␣'b',␣null))")))

# Easier: explode combinations via SQL expression:
pairs = items.select(
    F.expr("explode_outer(transform(aggregate(sequence(0,␣size(items)-1),␣array
        (),"
           "(acc,␣i)␣->␣concat(acc,␣transform(sequence(i+1,␣size(items)-1),␣j␣
               ->␣named_struct('a',␣items[i],␣'b',␣items[j]))))),␣x␣->␣x))").
               alias("pair")
).select("pair.a","pair.b").groupBy("a","b").count().orderBy(F.desc("count"))
```

20. Row/Column Summary Operations

Operation	pandas	PySpark
Row sums	`df.sum(axis=1)`	`F.expr("col1+col2+col3")or reduce(add)`
Row means	`df.mean(axis=1)`	`(sum of cols)/n`

Exercise (Solved): Portfolio value, diversification count, incomplete accounts.

Pandas

```
hold_cols = ["asset_a","asset_b","asset_c","asset_d"]
df["portfolio_value"] = df[hold_cols].sum(axis=1)
df["diversification"] = (df[hold_cols] != 0).sum(axis=1)
incomplete = df[df[hold_cols].isna().any(axis=1)]
```

PySpark

```
hold_cols = ["asset_a","asset_b","asset_c","asset_d"]
df = df.withColumn("portfolio_value", sum([F.coalesce(F.col(c),F.lit(0.0)) for
    c in hold_cols]))
df = df.withColumn("diversification", sum([(F.col(c)!=0).cast("int") for c in
    hold_cols]))
incomplete = df.filter(sum([F.col(c).isNull().cast("int") for c in hold_cols])
    > 0)
```

21. Advanced Text Operations

Operation	pandas	PySpark
Count patt.	`str.count("pat")`	`F.size(F.regex_extract_all not native; use regexp_replace trick)`
Find all	`str.findall("pat")`	`no direct; use regexp\_extract in loops or spark regex + split`
Normalize	`str.replace(..., regex=True)`	`F.regexp_replace`

Exercise (Solved): Feedback: count sentiment words, extract emails/phones, normalize product mentions, length categories.

Pandas

```
pos_words = ["good","great","excellent","love"]
neg_words = ["bad","poor","terrible","hate"]
df["pos_count"] = df["feedback"].str.lower().apply(lambda s: sum(s.count(w) for
    w in pos_words))
```

```
df["neg_count"] = df["feedback"].str.lower().apply(lambda s: sum(s.count(w) for
     w in neg_words))
df["emails"] = df["feedback"].str.findall(r'[A-Za-z0-9._%+-]+@[A-Za-z0
    -9.-]+\\.[A-Za-z]{2,}')
df["phones"] = df["feedback"].str.findall(r'\\+?\\d[\\d\\s\\-]{7,}\\d')
df["feedback_std"] = df["feedback"].str.replace(r'iPhone\\s*12','IPHONE12',
    regex=True)
df["text_len"] = df["feedback"].str.len()
df["len_bucket"] = pd.cut(df["text_len"], bins=[0,50,150,1e9],
                          labels=["short","medium","long"])
```

PySpark

```
pos_words = ["good","great","excellent","love"]
neg_words = ["bad","poor","terrible","hate"]

def count_words(col, words):
    expr = "␣+␣".join([f"length(lower({col}))␣-␣length(replace(lower({col}),␣'{
        w}',␣''))" for w in words])
    # each count divides by len(w)
    add = "␣+␣".join([f"(length(lower({col}))␣-␣length(replace(lower({col}),␣'{
        w}',␣'')))␣/␣{len(w)}" for w in words])
    return F.expr(f"({add})")

df = (df.withColumn("pos_count", count_words("feedback", pos_words))
        .withColumn("neg_count", count_words("feedback", neg_words))
        .withColumn("emails", F.expr("regexp_extract_all(feedback,␣'[A-Za-z0-9.
            _%+-]+@[A-Za-z0-9.-]+\\\\.[A-Za-z]{2,}',␣0)"))
        .withColumn("phones", F.expr("regexp_extract_all(feedback,␣'\\\\+?\\\\d
            [\\\\d\\\\s\\\\-]{7,}\\\\d',␣0)"))
        .withColumn("feedback_std", F.regexp_replace("feedback", r"iPhone\\s*12
            ", "IPHONE12"))
        .withColumn("text_len", F.length("feedback"))
        .withColumn("len_bucket",
            F.when(F.col("text_len")<=50,"short")
             .when(F.col("text_len")<=150,"medium")
             .otherwise("long")))
# Note: regexp_extract_all is available in newer Spark; otherwise split-based
    workaround may be needed.
```

22. Conditional Operations

Operation	pandas	PySpark
Mask/Where	`mask()/where()`	`F.when()/otherwise`
Clip	`clip(lower,upper)`	`F.least/F.greatest or exprmin/max` logic—

Exercise (Solved): Cap at 99th pct; negative revenues to 0; mask junior salaries.
Pandas

```
p99 = df["value"].quantile(0.99)
df["value_capped"] = df["value"].clip(upper=p99)
df["revenue"] = df["revenue"].clip(lower=0)
df["salary_masked"] = df["salary"].where(df["level"]!="Junior", other=pd.NA)
```

PySpark

```
p99 = df.agg(F.expr("percentile_approx(value,␣0.99)␣as␣p")).collect()[0]["p"]
df = df.withColumn("value_capped", F.least(F.col("value"), F.lit(p99)))
df = df.withColumn("revenue", F.greatest(F.col("revenue"), F.lit(0)))
df = df.withColumn("salary_masked", F.when(F.col("level")=="Junior", F.lit(None
    )).otherwise(F.col("salary")))
```

23. Exporting Data

Format	pandas	PySpark
CSV	`to_csv("out.csv", index=False)`	`df.write.mode("overwrite").option("header",True).csv("out_dir")`
Parquet	`to_parquet("out.parquet")`	`df.write.mode("overwrite").parquet("out.parquet")`
Partitioned	—	`df.write.partitionBy("year","month").parquet("out")`

Exercise (Solved): Export CSV; partitioned Parquet by department and quarter.

Pandas

```
df.to_csv("processed.csv", index=False)
df.to_parquet("processed.parquet")
```

PySpark

```
df.write.mode("overwrite").option("header", True).csv("processed_csv_dir")
df.write.mode("overwrite").partitionBy("department","quarter").parquet("
    processed_parquet")
```

24. Window Functions (extra)

Task	pandas	PySpark
Rolling sum	`df["r"] = df["x"].rolling(3).sum()`	`w=Window.orderBy("t").rowsBetween(-2,0); F.sum("x").over(w)`
Rank + dense rank	`groupby(...).rank()`	`F.rank().over(w), F.dense_rank().over(w)`
Lag/Lead	`groupby(...).shift()`	`F.lag("x",1).over(w), F.lead("x",1).over(w)`

Exercise (Solved): For each product: 7-day rolling sum of units; day-over-day growth; rank days by sales.

Pandas

```
sales["date"] = pd.to_datetime(sales["date"])
sales = sales.sort_values(["product_id","date"])
sales["roll7"] = sales.groupby("product_id")["units"].transform(lambda s: s.
    rolling(7, min_periods=1).sum())
sales["prev"] = sales.groupby("product_id")["units"].shift(1)
sales["dod_growth"] = (sales["units"] - sales["prev"]) / sales["prev"]
sales["rank_day"] = sales.groupby("product_id")["units"].rank(method="dense",
    ascending=False)
```

PySpark

```
from pyspark.sql.window import Window
sales = sales.withColumn("date", F.to_date("date")).orderBy("product_id","date"
    )
wprod = Window.partitionBy("product_id").orderBy("date").rowsBetween(-6,0)
sales = sales.withColumn("roll7", F.sum("units").over(wprod))
sales = sales.withColumn("prev", F.lag("units",1).over(Window.partitionBy("
    product_id").orderBy("date")))
sales = sales.withColumn("dod_growth", (F.col("units")-F.col("prev"))/F.col("
    prev"))
sales = sales.withColumn("rank_day", F.dense_rank().over(Window.partitionBy("
    product_id").orderBy(F.desc("units"))))
```