# An Embarrassment of Pandas

Kade Killary

August 9, 2019

## Contents

# DataFrames

## Options - documentation

```python
# More columns
pd.set_option("display.max_columns", 500)

# More rows
pd.set_option("display.max_rows", 500)

# Floating point precision
pd.set_option("display.precision", 3)

# Increase column width
pd.set_option("max_colwidth", 50)

# Change default plotting backend - Pandas >= 0.25
# https://github.com/PatrikHlobil/Pandas-Bokeh
pd.set_option("plotting.backend", 'pandas_bokeh')
```

## Useful `read_csv()` options - documentation

```python
pd.read_csv(
    "data.csv.gz",
    delimiter = "^",
    # line numbers to skip (i.e. headers in an excel report)
    skiprows = 2,
    # used to denote the start and end of a quoted item
    quotechar = "|",
    # return a subset of columns
    usecols = ["return_date", "company", "sales"],
    # data type for data or columns
    dtype = { "sales": np.float64 },
    # additional strings to recognize as NA/NaN
    na_values = [".", "?"],
    # convert to datetime, instead of object
    parse_dates = ["return_date"],
    # for on-the-fly decompression of on-disk data
    # options - gzip, bz2, zip, xz
    compression = "gzip",
    # encoding to use for reading
    encoding = "latin1",
    # read in a subset of data
    nrows = 100
)
```

## Read csv from URL or S3 - s3fs

```python
pd.read_csv("https://bit.ly/2KyxTFn")

# Requires s3fs library
pd.read_csv("s3://pandas-test/tips.csv")
```

## Read an Excel file - documentation

```
pd.read_excel("numbers.xlsx", sheet_name="Sheet1")

# Multiple sheets with varying parameters
with pd.ExcelFile("numbers.xlsx") as xlsx:
    df1 = pd.read_excel(xlsx, "Sheet1", na_values=["?"])
    df2 = pd.read_excel(xlsx, "Sheet2", na_values=[".", "Missing"])
```

## Read multiple files at once - glob

```
import glob

# ignore_index = True to avoid duplicate index values
df = pd.concat([pd.read_csv(f) for f in glob.glob("*.csv")], ignore_index = True)

# More options
df = pd.concat([pd.read_csv(f, encoding = "latin1") for f in glob.glob("*.csv")])
```

## Recursively grab all files in a directory

```
import os
import glob

files = [os.path.join(root, file)
        for root, dir, files in os.walk("./directory")
        for file in glob.glob("*.csv")]
```

## Read in data from SQLite3

```
import sqlite3

conn = sqlite3.connect("flights.db")
df = pd.read_sql_query("select * from airlines", conn)
conn.close()
```

## Read in data from Postgres - bigquery, snowflake

```
from sqlalchemy import create_engine

# Port 5439 for Redshift
engine = create_engine("postgresql://user@localhost:5432/mydb")
df = pd.read_sql_query("select * from airlines", engine)

# Get results in chunks
for chunk in pd.read_sql_query("select * from airlines", engine, chunksize=5):
    print(chunk)

# Writing back
df.to_sql(
    "table"
    schema="schema"
    # fail, replace or append
    if_exists="append",
    # write back in chunks
    chunksize = 10000
)
```

## Normalizing nested JSON - documentation

```
from pandas.io.json import json_normalize

json_normalize(data, "counties", ["state", "shortname", ["info", "governor"]])

# How deep to normalize - Pandas >= 0.25
json_normalize(data, max_level=1)
```

## Column headers

```
# Lower all values
df.columns = [x.lower() for x in df.columns]

# Strip out punctuation, replace spaces and lower
df.columns = df.columns.str.replace("[^\w\s]", "").str.replace(" ", "_").str.lower()

# Condense multiindex columns
df.columns = ["_".join(col).lower() for col in df.columns]

# Double transpose to remove bottom row for multiindex columns
df.T.reset_index(1, drop=True).T
```

## Filtering DataFrame - using `pd.Series.isin()`

```
df[df["dimension"].isin(["A", "B", "C"])]

# not in
df[~df["dimension"].isin(["A", "B", "C"])]
```

## Filtering DataFrame - using `pd.Series.str.contains()`

```
df[df["dimension"].str.contains("word")]

# not in
df[~df["dimension"].str.contains("word")]
```

## Filtering DataFrame & more - using `df.query()` - documentation

```
df.query("salary > 100000")

df.query("name == 'john'")

df.query("name == 'john' | name == 'jack'")

df.query("name == 'john' and salary > 100000")

df.query("name.str.contains('a')")

# Grab top 1% of earners
df.query("salary > salary.quantile(.99)")

# Make more than the mean
df.query("salary > salary.mean()")
```

```python
# Subset by top 3 most frequent products purchased
df.query("item in item.value_counts().nlargest(3).index")

# Query for null values
df.query("column.isnull()")

# Query for non-nulls
df.query("column.notnull()")

# @ - allows you to refer to variables in the environment
names = ["john", "fred", "jack"]
df.query("name in @names")

# Reference columns with spaces using backticks - Pandas >= 0.25
df.query("`Total Salary` > 100000")
```

### Joining - documentation

```python
# Inner join
pd.merge(df1, df2, on = "key")

# Left join on different key names
pd.merge(
    df1,
    df2,
    right_on = ["right_key"],
    left_on = ["left_key"],
    how = "left"
)
```

### Select columns based on data type

```python
df.select_dtypes(include = "number")
df.select_dtypes(exclude = "object")
```

### Reverse column order

```python
df.loc[:, ::-1]
```

### Correlation matrix

```python
df.corr()

# With another DataFrame
df.corrwith(df_2)
```

### Descriptive statistics

```python
df.describe(include=[np.number]).T

dims = df.describe(include=[pd.Categorical]).T

# Add percent frequency for top dimension
dims["frequency"] = dims["freq"].div(dims["count"])
```

### Styling numeric columns - documentation

```
styling_options = {
    "sales": "${0:,.0f}",
    "percent_of_sales": "{:.2%f}"
}

df.style.format(styling_options)
```

### Add highlighting for max and min values

```
df.style.highlight_max(color = "lightgreen").highlight_min(color = "red")
```

### Conditional formatting for one column

```
df.style.background(subset = ["measure"], cmap = "viridis")
```

## Series

### Value counts as percentages

```
# See NaNs as well
df["meaure"].value_counts(normalize = True, dropna = False)
```

### Replacing errant characters

```
df["sales"].str.replace("$", "")
```

### Replacing false conditions - documentation

```
df["steps_walked"].where(df["steps_walked"] > 0, 0)
```

## Missing Values

### Percent nulls by column

```
(df.isnull().sum() / df.isnull().count()).sort_values(ascending=False)
```

### Dropping columns - documentation

```
df.drop(["column_a", "column_b"], axis = 1)
```

### Dropping duplicate rows - documentation

```
df.drop_duplicates(subset=["order_date", "product"], keep="first")
```

### Dropping columns based on NaN threshold - documentation

```
# Any column with 90% missing values will be dropped
df.dropna(thresh = len(df) * .9, axis = 1)
```

### Replacing using `fillna()` - documentation

```
# Impute DataFrame with all zeroes
df.fillna(0)

# Impute column with all zeroes
df["measure"].fillna(0)

# Impute measure with mean of column
df["measure"].fillna(df["measure"].mean())

# Impute dimension with mode of column
df["dimension"].fillna(df["dimension"].mode())

# Impute by another dimension's mean
df["age"].fillna(df.groupby("sex")["age"].transform("mean"))
```

### Replace values across entire DataFrame

```
df.replace(".", np.nan)

df.replace(0, np.nan)
```

### Replace numeric values containing a letter with NaN

```
df["zipcode"].replace(".*[a-zA-Z].*", np.nan, regex=True)
```

### Drop rows where any value is 0

```
df[(df != 0).all(1)]
```

### Drop rows where all values are 0

```
df = df[(df.T != 0).any()]
```

## Method Chaining

### Chaining multiple operations

```
(pd.read_csv("employee_salaries.csv")
    .query("salary > 0")
    .assign(sex = lambda df: df["sex"].replace({"female": 1, "male: 0}),
            age = lambda df: pd.cut(df["age"].fillna(df["age"].median()),
                                    bins = [df["age"].min(), 18, 40, df["age"].max()],
                                    labels = ["underage", "young", "experienced"]))
    .rename({"name_1": "first_name", "name_2": "last_name"})
)
```

### Pipelines for data processing

```
def fix_headers(df):
    df.columns = df.columns.str.replace("[^\w\s]", "").str.replace(" ", "_").str.lower()
    return df
```

```python
def drop_columns_missing(df, percent):
    df = df.dropna(thresh = len(df) * percent, axis = 1)
    return df

def fill_missing(df, value):
    df = df.fillna(value)
    return df

def replace_and_convert(df, col, orig, new, dtype):
    df[col] = df[col].str.replace(orig, new).astype(dtype)
    return df


(df.pipe(fix_headers)
    .pipe(drop_columns_missing, percent=0.3)
    .pipe(fill_missing, value=0)
    .pipe(replace_and_convert, col="sales", orig="$", new="", dtype=float)
)
```

Recommended Read - Effective Pandas

## Aggregation

### Use as_index = False to avoid setting index

```python
# this
df.groupby("dimension", as_index = False)["measure"].sum()

# versus this
df.groupby("dimension")["measure"].sum().reset_index()
```

### By date offset - documentation

```python
# H for hours
# D for days
# W for weeks
# WOM for week of month
# Q for quarter end
# A for year end
df.groupby(pd.Grouper(key = "date", freq = "M"))["measure"].agg(["sum", "mean"])
```

### Measure by dimension - documentation

```python
# count - number of non-null observations
# sum - sum of values
# mean - mean of values
# mad - mean absolute deviation
# median - arithmetic median of values
# min - minimum
# max - maxmimum
# mode - mode
# std - unbiased standard deviation
# first - first value
# last - last value
# nunique - unique values
df.groupby("dimension")["measure"].sum()
```

```
# Specific aggregations for columns
df.groupby("dimension").agg(
    {
        "sales": ["mean", "sum"],
        "sale_date": "first",
        "customer": "nunique"
    }
)
```

### Pivot table - documentation

```
pd.pivot_table(
    df,
    values=["sales", "orders"],
    index=["customer_id"],
    aggfunc={
        "sales": ["sum", "mean"],
        "orders": "nunique"
    }
)
```

### Named aggregations - `Pandas >= 0.25` - documentation

```
# DataFrame - Version 1
df.groupby("country").agg(
    min_height = pd.NamedAgg(column = "height", aggfunc = "min"),
    max_height = pd.NamedAgg(column = "height", aggfunc = "max"),
    average_weight = pd.NamedAgg(column = "weight", aggfunc = np.mean)
)

# DataFrame - Version 2
df.groupby("country").agg(
    min_height=("height", "min"),
    max_heights=("height", "max"),
    average_weight=("weight", np.mean)
)

# Series
df.groupby("gender").height.agg(
    min_height="min",
    max_height="max"
)
```

## New Columns

### Using `df.eval()`

```
df["sales"] = df.eval("price * quantity")

# Assign to different DataFrame
pd.eval("sales = df.price * df.quantity", target=df_2)

# Multiline assignment
df.eval("""
aov = price / quantity
```

```
aov_gt_50 = (price / quantity) > 50
top_3_customers = customer_id in customer_id.value_counts().nlargest(3).index
bottom_3_customers = customer_id in customer_id.value_counts().nsmallest(3).index
""")
```

### Based on one condition - using `np.where()`

```
np.where(df["gender"] == "Male", 1, 0)
```

### Based on multiple conditions - using `np.where()`

```
np.where(df["measure"] < 5, "Low", np.where(df["measure"] < 10, "Medium", "High"))
```

### Based on multiple conditions - using `np.select()`

```
conditions = [
    df["country"].str.contains("spain"),
    df["country"].str.contains("italy"),
    df["country"].str.contains("chile"),
    df["country"].str.contains("brazil")
]

choices = ["europe", "europe", "south america", "south america"]

data["continent"] = np.select(conditions, choices, default = "other")
```

### Based on manual mapping - using `pd.Series.map()`

```
values = {"Low": 1, "Medium": 2, "High": 3}

df["dimension"].map(values)
```

### Automatically generate mappings from dimension

```
dimension_mappings = {v: k for k, v in enumerate(df["dimension"].unique())}

df["dimension"].map(dimension_mappings)
```

### Splitting a string column

```
df["email"].str.split("@", expand = True)[0]
```

### Using list comprehensions

```
df["domain"] = [x.split("@")[1] for x in df["email"]]
```

### Using regular expressions

```
import re

pattern = "([A-Z0-9._%+-]+)@([A-Z0-9.-]+)"
```

```
# Inserting colum headers, applied after extract
pattern = "(?P<email>[A-Z0-9._%+-]+)@(?P<domain>[A-Z0-9.-]+)"

# Generates two columns
email_components = df["email"].str.extract(pattern, flags=re.IGNORECASE)
```

### Widening a column - documentation

```
df.pivot(index = "date", columns = "companies", values = "sales")
```

## Feature Engineering

### Instead of split-apply-combine, transform()

```
# this
df["mean_company_salary"] = df.groupby("company")["salary"].transform("mean")

# versus this
mean_salary = df.groupby("company")["salary"]\
    .agg("mean")\
    .rename("mean_salary")\
    .reset_index()

df_new = df.merge(mean_salary)
```

### Extracting various date components - documentation

```
df["date"].dt.year
df["date"].dt.quarter
df["date"].dt.month
df["date"].dt.week
df["date"].dt.day
df["date"].dt.weekday
df["date"].dt.weekday_name
df["date"].dt.hour
```

### Time between two dates

```
# Days between
df["first_date"].sub(df["second_date"]).div(np.timedelta64(1, "D"))

# Months between
df["first_date"].sub(df["second_date"]).div(np.timedelta64(1, "M"))

# Equivalent to above
(df["first_date] - df["second_date"]) / np.timedelta64(1, "M")
```

### Weekend column

```
df["is_weekend"] = np.where(df["date"].dt.dayofweek.isin([5, 6]), 1, 0)
```

### Get prior date

```
df.sort_values(by=["customer_id, "order_date"])\
    .groupby("customer_id")["order_date"].shift(periods=1)
```

### Days since prior date

```
df.sort_values(by = ["customer_id", "order_date"])\
    .groupby("customer_id")["order_date"]\
    .diff()\
    .div(np.timedelta64(1, "D"))
```

### Percent change since prior date

```
df.sort_values(by = ["customer_id", "order_date"])\
    .groupby("customer_id")["order_date"]\
    .pct_change()
```

### Percentile rank for measure

```
df["salary"].rank(pct=True)
```

### Occurrences of word in row

```
import re

df["review"].str.count("great", flags=re.IGNORECASE)
```

### Distinct list aggregation

```
df["unique_products"] = df.groupby("customer_id").agg({"products": "unique"})

# Transform each element -> row - Pandas >= 0.25
df["unique_products"].explode()
```

### User-item matrix

```
df.groupby("customer_id")["products"].value_counts().unstack().fillna(0)
```

### Binning

```
pd.qcut(data["measure"], q = 4, labels = False)

# Numeric
pd.cut(df["measure"], bins = 4, labels = False)

# Dimension
pd.cut(df["age"], bins = [0, 18, 25, 99], labels = ["child", "young adult", "adult"])
```

### Dummy variables

```
# Use drop_first = True to avoid collinearity
pd.get_dummies(df, drop_first = True)
```

### Sort and take first value by dimension

```
df.sort_values(by = "variable").groupby("dimension").first()
```

### MinMax normalization

```
df["salary_minmax"] = (
    df["salary"] - df["salary"].min()) / (df["salary"].max() - df["salary"].min()
)
```

### Z-score normalization

```
df["salary_zscore"] = (df["salary"] - df["salary"].mean()) / df["salary"].std()
```

### Log transformation

```
# For positive data with no zeroes
np.log(df["sales"])

# For positive data with zeroes
np.log1p(df["sales"])

# Convert back - get predictions if target is log transformed
np.expm1(df["sales"])
```

### Boxcox transformation

```
from scipy import stats

# Must be positive
stats.boxcox(df["sales"])[0]
```

### Reciprocal transformation

```
df["age_reciprocal"] = 1.0 / df["age"]
```

### Square root transformation

```
df["age_sqrt"] = np.sqrt(df["age"])
```

### Winsorization

```
upper_limit = np.percentile(df["salary"].values, 99)
lower_limit = np.percentile(df["salary"].values, 1)

df["salary"].clip(lower = lower_limit, upper = upper_limit)
```

### Mean encoding

```
df.groupby("dimension")["target"].transform("mean")
```

### Z-scores for outliers

```
from scipy import stats
import numpy as np

z = np.abs(stats.zscores(df))
df = df[(z < 3).all(axis = 1)]
```

### Interquartile range (IQR)

```
q1 = df["salary"].quantile(0.25)
q3 = df["salary"].quantile(0.75)
iqr = q3 - q1

df.query("(@q1 - 1.5 * @iqr) <= salary <= (@q3 + 1.5 * @iqr)")
```

### Geocoder - github

### Geopy - github

```
import geocoder

df["lat_long"] = df["ip"].apply(lambda x: geocoder.ip(x).latlng)
```

### RFM - Recency, Frequency and Monetary

```
rfm = (
    df.groupby("customer_id")
    .agg(
        {
            "order_date": lambda x: (x.max() - x.min()).days,
            "order_id": "nunique",
            "price": "mean",
        }
    )
    .rename(
        columns={"order_date": "recency", "order_id": "frequency", "price": "monetary"}
    )
)

rfm_quantiles = rfm.quantile(q=[0.2, 0.4, 0.6, 0.8])

recency_conditions = [
    rfm.recency >= rfm_quantiles.recency.iloc[3],
    rfm.recency >= rfm_quantiles.recency.iloc[2],
    rfm.recency >= rfm_quantiles.recency.iloc[1],
    rfm.recency >= rfm_quantiles.recency.iloc[0],
    rfm.recency <= rfm_quantiles.recency.iloc[0],
]

frequency_conditions = [
    rfm.frequency <= rfm_quantiles.frequency.iloc[0],
    rfm.frequency <= rfm_quantiles.frequency.iloc[1],
    rfm.frequency <= rfm_quantiles.frequency.iloc[2],
    rfm.frequency <= rfm_quantiles.frequency.iloc[3],
    rfm.frequency >= rfm_quantiles.frequency.iloc[3],
```

```
]

monetary_conditions = [
    rfm.monetary <= rfm_quantiles.monetary.iloc[0],
    rfm.monetary <= rfm_quantiles.monetary.iloc[1],
    rfm.monetary <= rfm_quantiles.monetary.iloc[2],
    rfm.monetary <= rfm_quantiles.monetary.iloc[3],
    rfm.monetary >= rfm_quantiles.monetary.iloc[3],
]

ranks = [1, 2, 3, 4, 5]

rfm["r"] = np.select(recency_conditions, ranks, "other")
rfm["f"] = np.select(frequency_conditions, ranks, "other")
rfm["m"] = np.select(monetary_conditions, ranks, "other")

rfm["segment"] = rfm["r"].astype(str).add(rfm["f"].astype(str))

segment_map = {
    r"[1-2][1-2]": "hibernating",
    r"[1-2][3-4]": "at risk",
    r"[1-2]5": "cannot lose",
    r"3[1-2]": "about to sleep",
    r"33": "need attention",
    r"[3-4][4-5]": "loyal customers",
    r"41": "promising",
    r"51": "new customers",
    r"[4-5][2-3]": "potential loyalists",
    r"5[4-5]": "champions",
}

rfm["segment"] = rfm.segment.replace(segment_map, regex=True)
```

### Haversine

```
import numpy as np
from numpy import pi, deg2rad, cos, sin, arcsin, sqrt

def haversine(s_lat, s_lng, e_lat, e_lng):
    """
    determines the great-circle distance between two point
    on a sphere given their longitudes and latitudes
    """

    # approximate radius of earth in miles
    R = 3959.87433

    s_lat = s_lat * np.pi / 180.0
    s_lng = np.deg2rad(s_lng)
    e_lat = np.deg2rad(e_lat)
    e_lng = np.deg2rad(e_lng)

    d = (
        np.sin((e_lat - s_lat) / 2) ** 2
        + np.cos(s_lat) * np.cos(e_lat) * np.sin((e_lng - s_lng) / 2) ** 2
    )
```

```
    return 2 * R * np.arcsin(np.sqrt(d))


df['distance'] = haversine(
    df["start_lat"].values,
    df["start_long"].values,
    df["end_lat"].values,
    df["end_long"].values
)
```

### Manhattan

```
def manhattan(s_lat, s_lng, e_lat, e_lng):
    """
    sum of horizontal and vertical distance between
    two points
    """
    a = haversine(s_lat, s_lng, s_lat, e_lng)
    b = haversine(s_lat, s_lng, e_lat, s_lng)
    return a + b
```

## Random

### Union two categorical columns - documentation

```
from pandas.api.types import union_categoricals

food = pd.Categorical(["burger king", "wendys"])
food_2 = pd.Categorical(["burger king", "chipotle"])

union_categoricals([food, food_2])
```

### Testing - documentation

```
from pandas.util.testing import assert_frame_equal

# Methods for Series and Index as well
assert_frame_equal(df_1, df_2)
```

### Checking data types - documentation

```
from pandas.api.types import is_numeric_dtype

is_numeric_dtype("hello world")
# False
```

### Infer column dtype, useful to remap column dtypes documentation

```
from pandas.api.types import infer_dtype

infer_dtype(["john", np.nan, "jack"], skipna=True)
# string

infer_dtype(["john", np.nan, "jack"], skipna=False)
# mixed
```