

# DataFrame column operations

CLEANING DATA WITH PYSPARK



**Mike Metzger**

Data Engineering Consultant

# DataFrame refresher

DataFrames:

- Made up of rows & columns
- Immutable
- Use various transformation operations to modify data



```
# Return rows where name starts with "M"
voter_df.filter(voter_df.name.like('M%'))
# Return name and position only
voters = voter_df.select('name', 'position')
```

# Common DataFrame transformations

- Filter / Where 

```
voter_df.filter(voter_df.date > '1/1/2019') # or voter_df.where(...)
```

- Select

```
voter_df.select(voter_df.name)
```

- withColumn

```
voter_df.withColumn('year', voter_df.date.year)
```

- drop

```
voter_df.drop('unused_column')
```

# Filtering data

- Remove nulls
- Remove odd entries
- Split data from combined sources
- Negate with ~

```
voter_df.filter(voter_df['name'].isNotNull())  
voter_df.filter(voter_df.date.year > 1800)  
voter_df.where(voter_df['_c0'].contains('VOTE'))  
voter_df.where(~ voter_df._c1.isNull())
```

# Column string transformations

- Contained in `pyspark.sql.functions`

```
import pyspark.sql.functions as F
```

- Applied per column as transformation

```
voter_df.withColumn('upper', F.upper('name'))
```

- Can create intermediary columns

```
voter_df.withColumn('splits', F.split('name', ' '))
```

- Can cast to other types

```
voter_df.withColumn('year', voter_df['_c4'].cast(IntegerType()))
```

# ArrayType() column functions

Various utility functions / transformations to interact with ArrayType()

`.size(<column>)` - returns length of arrayType() column

`.getItem(<index>)` - used to retrieve a specific item at index of list column.

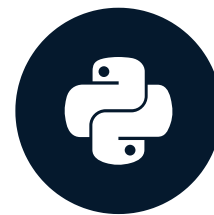


# Let's practice!

CLEANING DATA WITH PYSPARK

# Conditional DataFrame column operations

CLEANING DATA WITH PYSPARK



**Mike Metzger**

Data Engineering Consultant



# Conditional clauses

*Conditional Clauses* are:

- Inline version of if / then / else 🗨
- `.when()`
- `.otherwise()`

# Conditional example

```
.when(<if condition>, <then x>)
```

```
df.select(df.Name, df.Age, F.when(df.Age >= 18, "Adult"))
```

Name	Age	
Alice	14	
Bob	18	Adult
Candice	38	Adult

# Another example

Multiple `.when()`

```
df.select(df.Name, df.Age,  
         .when(df.Age >= 18, "Adult")  
         .when(df.Age < 18, "Minor"))
```

Name	Age	
Alice	14	Minor
Bob	18	Adult
Candice	38	Adult

# Otherwise

`.otherwise()` is like `else`

```
df.select(df.Name, df.Age,  
          .when(df.Age >= 18, "Adult")  
          .otherwise("Minor"))
```



Name	Age	
Alice	14	Minor
Bob	18	Adult
Candice	38	Adult

**Let's practice!**  
CLEANING DATA WITH PYSPARK

# User defined functions

CLEANING DATA WITH PYSPARK



**Mike Metzger**

Data Engineering Consultant

# Defined...

*User defined functions* or UDFs



- Python method
- Wrapped via the `pyspark.sql.functions.udf` method
- Stored as a variable
- Called like a normal Spark function

# Reverse string UDF

Define a Python method

```
def reverseString(mystr):  
    return mystr[::-1]
```

Wrap the function and store as a variable

```
udfReverseString = udf(reverseString, StringType())
```



Use with Spark

```
user_df = user_df.withColumn('ReverseName',  
                             udfReverseString(user_df.Name))
```



# Argument-less example

```
def sortingCap():  
    return random.choice(['G', 'H', 'R', 'S'])  
udfSortingCap = udf(sortingCap, StringType())  
user_df = user_df.withColumn('Class', udfSortingCap())
```

Name	Age	Class
Alice	14	H
Bob	18	S
Candice	63	G

**Let's practice!**  
CLEANING DATA WITH PYSPARK

# Partitioning and lazy processing


CLEANING DATA WITH PYSPARK





**Mike Metzger**

Data Engineering Consultant

# Partitioning


- DataFrames are broken up into partitions
- Partition size can vary
- Each partition is handled independently 

# Lazy processing

- Transformations are **lazy** 
  - `.withColumn(...)`
  - `.select(...)`
- Nothing is actually done until an action is performed
  - `.count()`
  - `.write(...)`
- Transformations can be re-ordered for best performance 
- Sometimes causes unexpected behavior

# Adding IDs

Normal ID fields:

- Common in relational databases
- Most usually an integer increasing, sequential and unique
- Not very parallel 

id	last name	first name	state
0	Smith	John	TX
1	Wilson	A.	IL
2	Adams	Wendy	OR

# Monotonically increasing IDs

`pyspark.sql.functions.monotonically_increasing_id()`

- Integer (64-bit), increases in value, unique
- Not necessarily sequential (gaps exist)
- Completely parallel

id	last name	first name	state
0	Smith	John	TX
134520871	Wilson	A.	IL
675824594	Adams	Wendy	OR



# Notes

Remember, Spark is *lazy*!

- Occasionally out of order
- If performing a join, ID may be assigned after the join
- Test your transformations







# Let's practice!

CLEANING DATA WITH PYSPARK

