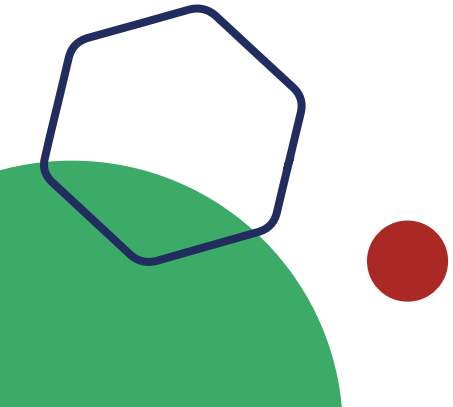
ADVANCING ANALYTICS

Data Transformations

Using PySpark for Data Manipulation

Transforming Data

- Conditional Functions
- String/ Regex Functions
- Complex Data Structures
- Dataframe Joins
- Datetime Functions
- User-Defined Functions



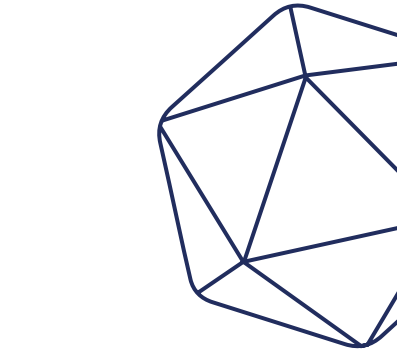
ADVANCING ANALYTICS

Conditional Functions

Conditional Functions

- When and Otherwise Functions

- Handling Nulls



ADVANCING ANALYTICS

When Conditions

```
df.withColumn("testFlag",when(col("BuildingID")== -1, lit(1)))
```

The when() function tests a Boolean condition, and returns the second argument when the condition is met, much like an IF() statement in Excel

The SQL equivalent of this would be a CASE statement

```
SELECT CASE WHEN BuildingID = -1 THEN 1 END AS testFlag
```

ADVANCING ANALYTICS

When Otherwise Conditions

```
df.withColumn("testFlag",
       when(col("BuildingID")== -1, lit(1))
       .otherwise(lit(0))
       )
```

We can add an otherwise() condition that is triggered if our original condition is not met. Our when().otherwise() function then similar to adding an "ELSE" clause to our SQL CASE statement

```
SELECT CASE WHEN BuildingID = -1 THEN 1
                        ELSE 0 END AS testFlag
```

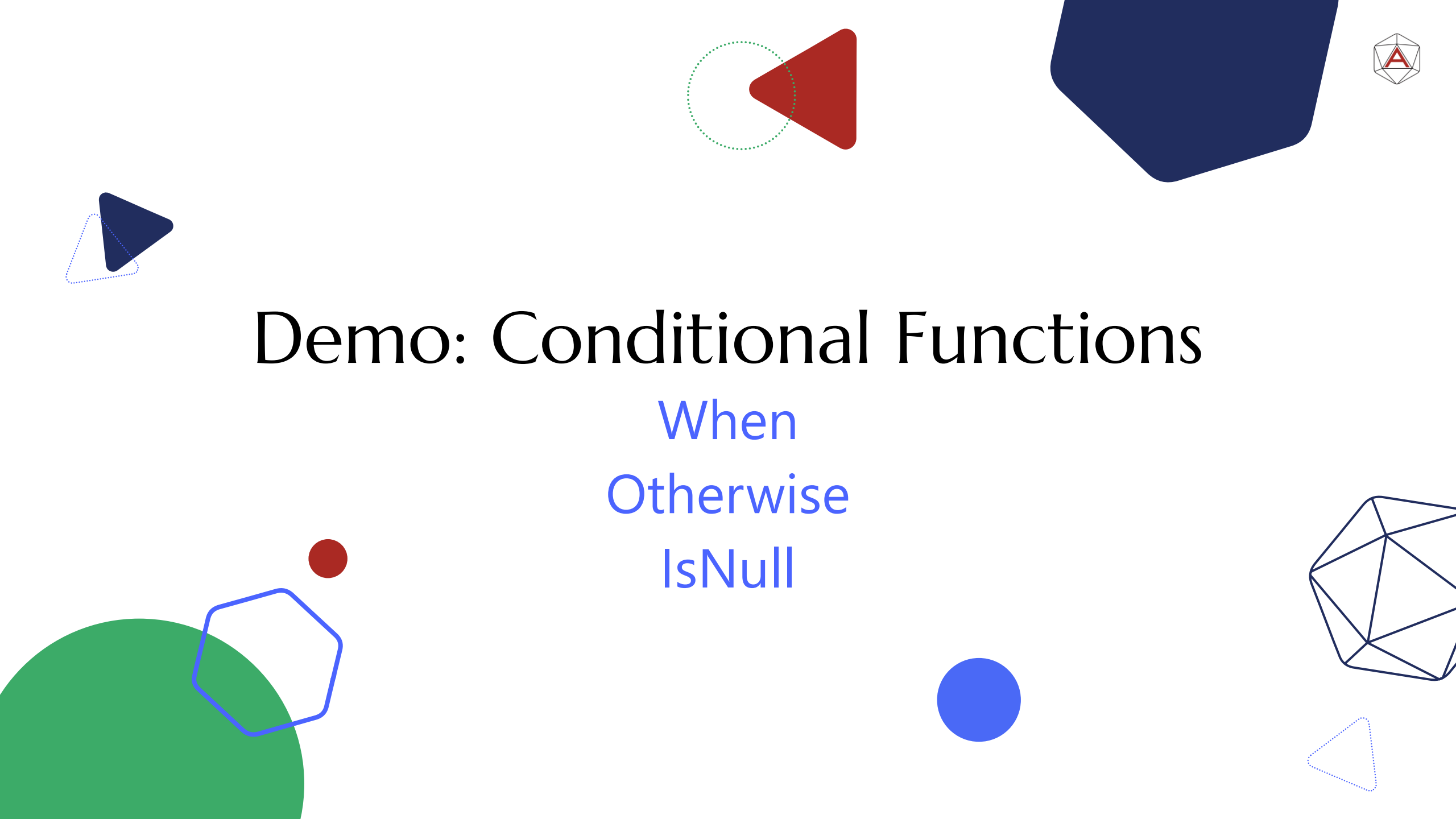
Isnull Tests

```
df.withColumn("noBuildingIDFlag",
       when(isnull(col("BuildingID")), lit(1))
       .otherwise(lit(0))
       )
```

If we need to test whether a record has a null value for a specific column, we can do this with the isnull() function.

This tests the value the specified cell and returns true if it is null, false if it is not.

This is useful for building data validation tests and auditing data quality.



Demo: Conditional Functions

When

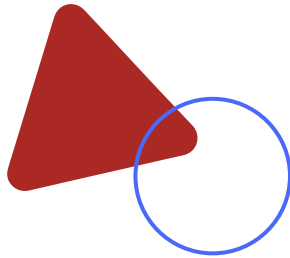
Otherwise

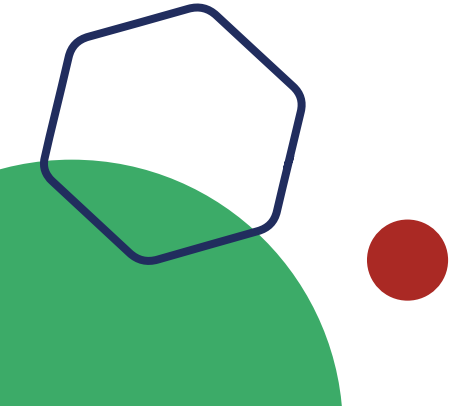
IsNull

LAB01 – Conditional Functions

Conditional Functions Summary

- Case statements can be reproduced with .when().otherwise()
- You can always just use CASE statements inside an expr()
- IsNull is a useful test





ADVANCING ANALYTICS

String/Regex Functions

String/Regex Functions

- Spark has useful string manipulation functions built in
- Regex can be very powerful for data cleansing & validation

ADVANCING ANALYTICS

Data Transformation Functions

```
df.withColumn(“myCol”,

        ltrim() & rtrim()
        trim()
        lpad() & rpad()

        regexp_replace(col(“phone"),“[^0-9]“,””))
        regexp_extract(col(“email"),"(?<=@)[^.]+(?=\.)“,””)
    )
```

Regexp_replace

```
df.withColumn(“myCol”,
       regexp_replace(col(“phone"),“[^0-9]“,””))
)
```

We can use regexp_replace() to search for characters, strings or patterns within an input string, and replace them with another value.

This can be basic string matching, or can be complex regular expressions - it's a very useful function for cleaning data!

Regex Crash Course – Pattern Matching

- Firstly, regex is all about pattern matching, we can specify ranges of characters that regex will look for:
  - [A-Z] - searches for individual characters that are uppercase
  - [A-z] - searches for individual characters that are either upper or lowercase
  - [0-9] - searches for numerical digits
  - [A-z]+ - keeps matching characters until it fails to match
  - "^" acts as a "not" – so we can use [^0-9] to say "not a numerical digit"
- Regex has the idea of a "capture group" - groups of characters that match what we are looking for, treated as a single match
- (hello) - searches for the string "hello"

ADVANCING ANALYTICS

Regex Crash Course – Lookaheads/Behinds

This is fairly advanced regex, but is incredibly valuable!

You can wrap your regex command with the following constraints:

- (?<=hello) - positive lookbehind, the match MUST be preceded by this pattern
- (?<!hello) - negative lookbehind, the match MUST NOT be preceded by this pattern

- (?=hello) - positive lookahead, the match MUST be followed by this pattern
- (?!hello) - negative lookahead, the match MUST NOT be followed by this pattern

```
test@advancinganalytics.co.uk
```

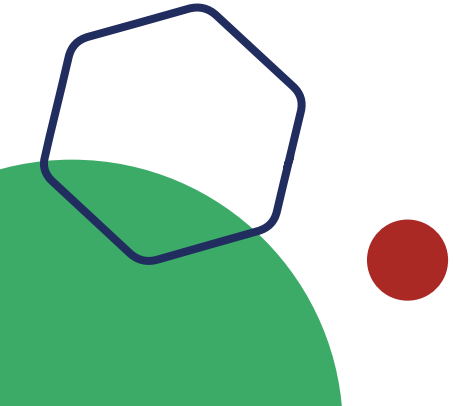
```
(?<=@)([A-z0-9]+)
```

ADVANCING ANALYTICS

Regexp_Extract

```
regexp_extract(col(“email"),"(?<=@)([A-z0-9]+)“,1)
```

The regexp_extract() function uses the same search functionality as the replace function. The patterns that are matched are returned as the results of the function – several different groups can be returned so we can specify which to use.



ADVANCING ANALYTICS

Demo: String/Regex Functions

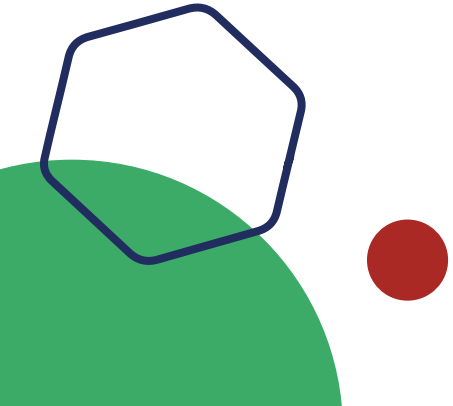
regexp_replace()

regexp_extract()

LABO2 – Regular Expressions

String/Regex Functions Summary

- PySpark has several useful string manipulation functions, but regular expressions are the most powerful
- Regex is tricky at first, but it is worth investing time learning the basics!

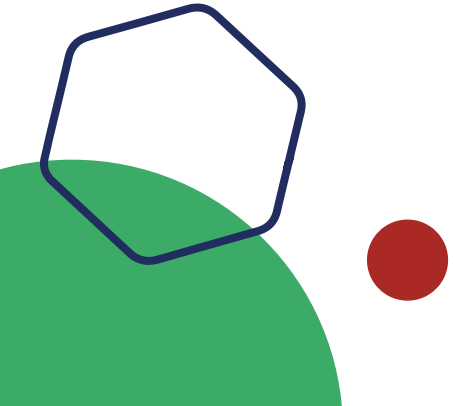


ADVANCING ANALYTICS

Complex Data Structures

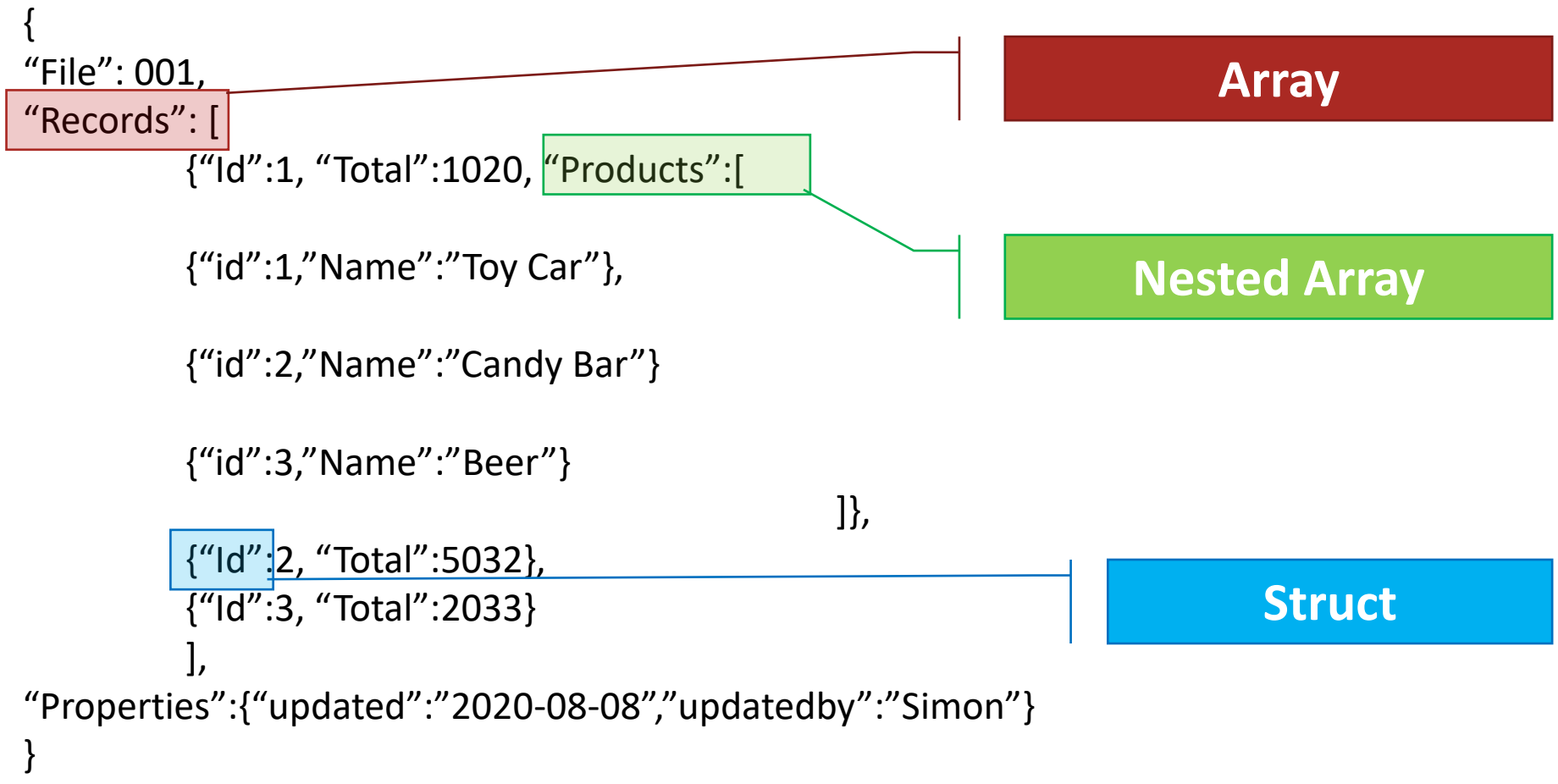
Complex Data Structures

- Working with JSON
- Splits & Explodes



ADVANCING ANALYTICS

What if the data isn't tabular?



ADVANCING ANALYTICS

Working with Nested Structs

```
{
"File": 001,
"Records": [...
            ],
"Properties":{"updated":"2020-08-08","updatedby":"Simon"}
}
```

This is using a syntax known as "Dot Notation" – Bracket Notation is also supported: Properties["updatedby"]

Dot Notation is simpler if you are going down several nested levels!

```
# Query the inner struct attributes
df = df.select(col("Properties.updatedby"))
df.show()
```

Result:

```
Properties.updatedby
--------------------------
Simon
```

ADVANCING ANALYTICS

JSON Inside Columns - from_json() Function

What if you're working with a standard structured data set, but it contains unstructured data inside a column?

ID	Name	Audit
1	Toy Car	{"updated":"2020-08-08","updatedby":"Simon"}
2	Candy Bar	{"updated":"2020-08-08","updatedby":"Simon"}
3	Beer	{"updated":"2020-08-08","updatedby":"Simon"}

```
# Parse the embedded JSON inside the "Audit" column
df = df.withColumn("JAudit",from_json(col("Audit"),jschema)
df = df.withColumn("JAudit",expr("Audit:updatedBy")

df = df.select(col("ID"),col("JAudit.updatedby"))
```

Result:

ID	JAudit.updatedby
1	*Simon*
2	*Simon*
3	*Simon*

ADVANCING ANALYTICS

Creating Arrays - Split() Function

Week	Purchases
32	Toy Car, Candy Bar, Beer

Sometimes we get lists of data but held as a single object. Here we have a comma separated string, but we want to treat each purchase as a separate object!

```
# Parse the embedded JSON inside the "Audit" column
df = df.withColumn("Items",split(col("Purchases"),","))

df.show()
```

```
df: pyspark.sql.dataframe.DataFrame
  Week: string
  Purchases: string
  Items: array
      element: string
```

Result:

Week	Purchases	Items
32	*Toy Car, Candy Bar, Beer*	*[Toy Car, Candy Bar, Beer]*

ADVANCING ANALYTICS

Flattening Complex Structures - Explode Function()

explode() Function takes a nested array and creates a new row for each item in that array.

You can only explode one array at a time!

Week	Purchases	Items
32	*Toy Car, Candy Bar, Beer*	*[Toy Car, Candy Bar, Beer]*

```
# Parse the embedded JSON inside the "Audit" column
df = df.withColumn("Items",explode(col("Items")))

df.show()
```

Result:

Week	Purchases	Items
32	*Toy Car, Candy Bar, Beer*	*Toy Car*
32	*Toy Car, Candy Bar, Beer*	*Candy Bar*
32	*Toy Car, Candy Bar, Beer*	*Beer*

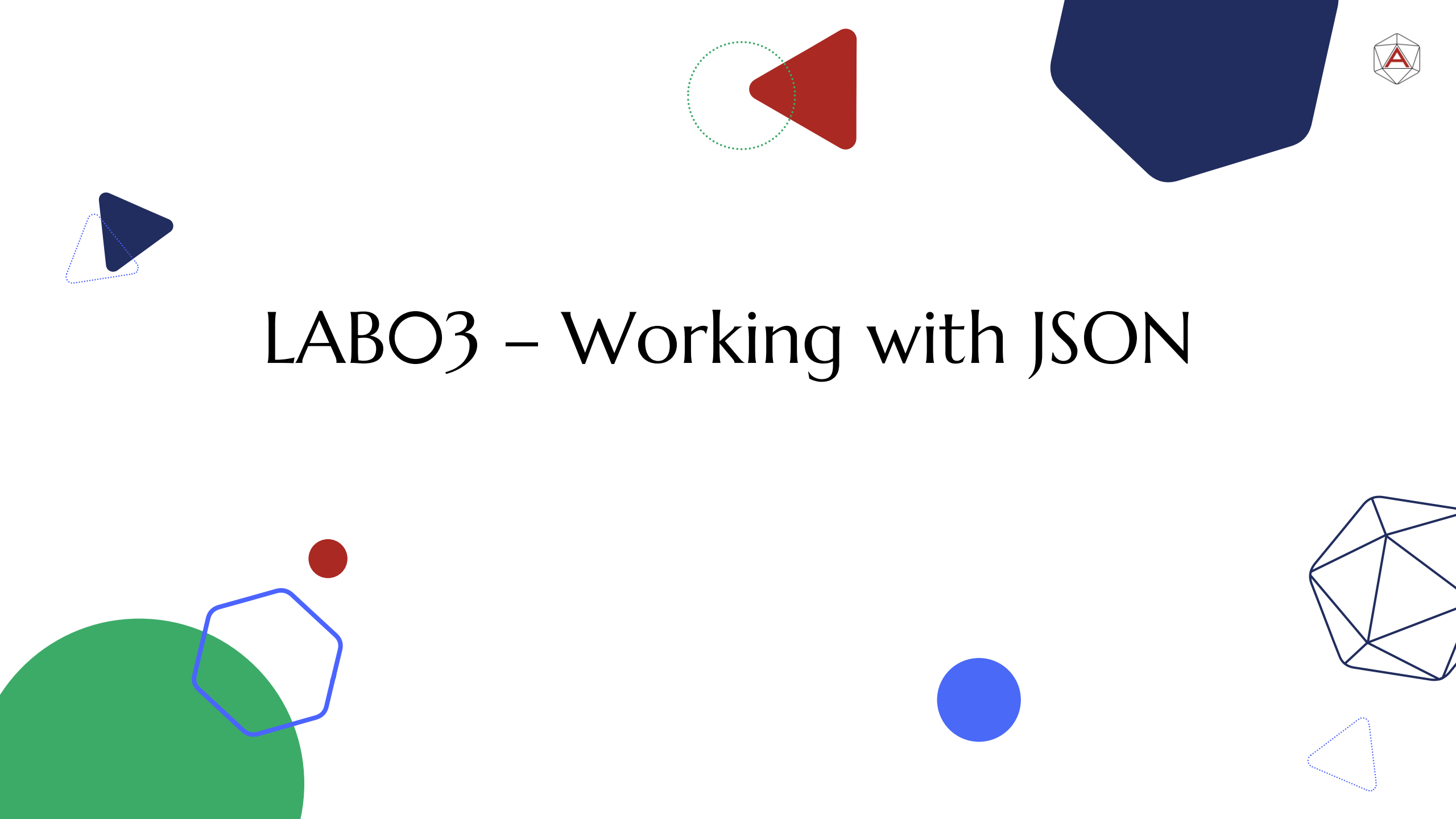
Demo: Working with JSON

Working with JSON

Dot Notation

JSON Parsing

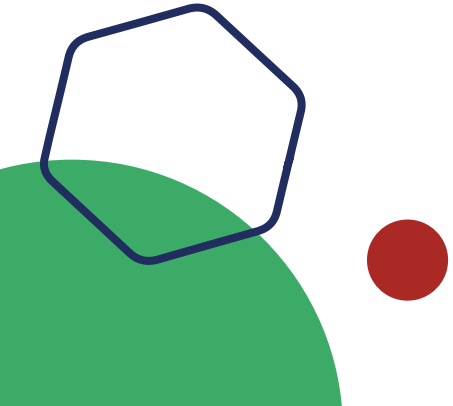
Splits & Explodes



LAB03 – Working with JSON

Complex Data Structures Summary

- Spark is far better at working with complex structures than traditional relational databases
- Users can easily query JSON strings
- Decide when to explode/unpack data based on performance

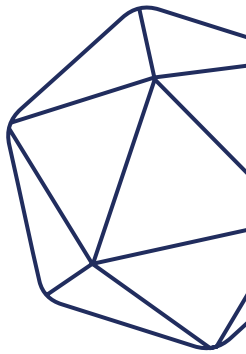


ADVANCING ANALYTICS

Dataframe Joins

Dataframe Joins

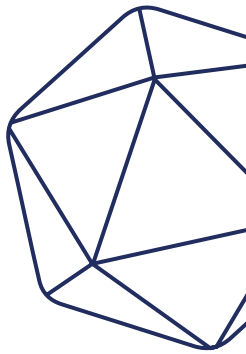
- Basic Joins
- Complex Joins
- Cross Joins

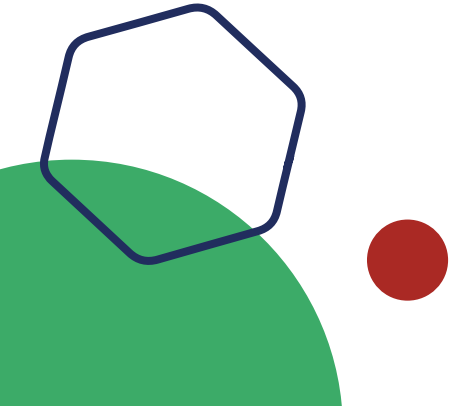


ADVANCING ANALYTICS

Dataframe Joins

```
df1 = somedata
df2 = someotherdata

newdf = df1.join(df2, df1.column == df2.column, 'jointype')
```

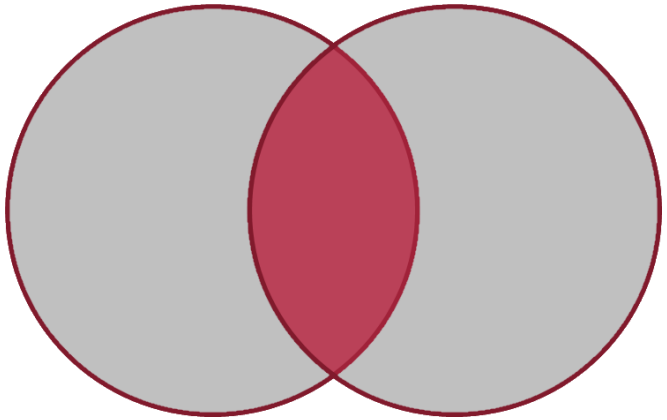




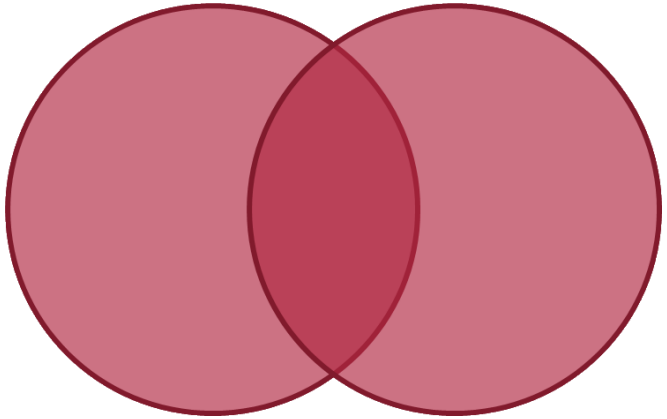
ADVANCING ANALYTICS

Inner & Outer Joins

```
'inner' (default)
```

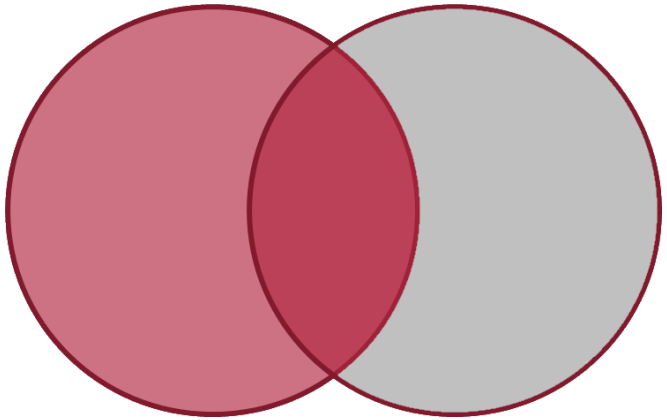


```
'outer',

'full',

'fullouter',
'full_outer'
```

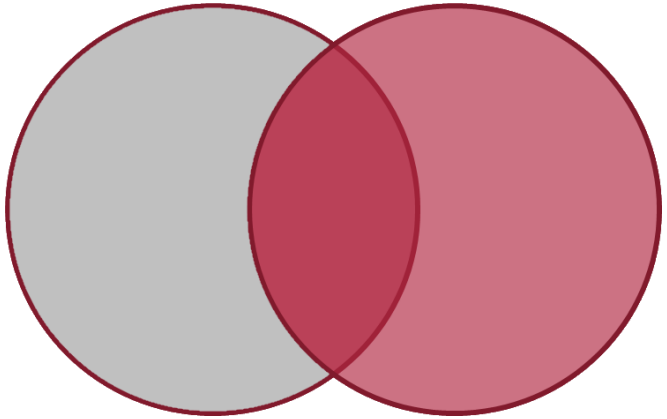


Left & Right Joins

```
'leftouter',
'left',
'left_outer'
```



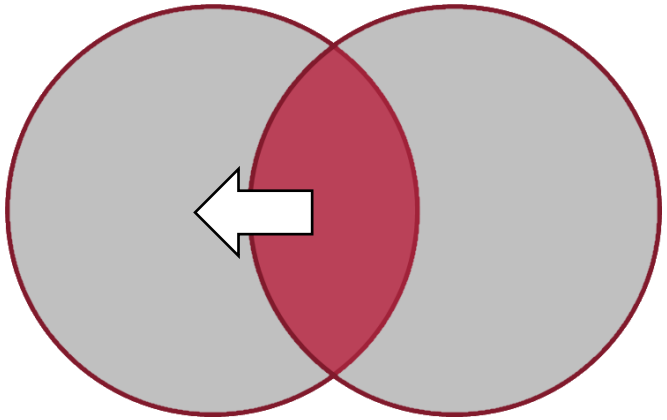
```
'rightouter',
'right',
'right_outer'
```



ADVANCING ANALYTICS

Semi Join

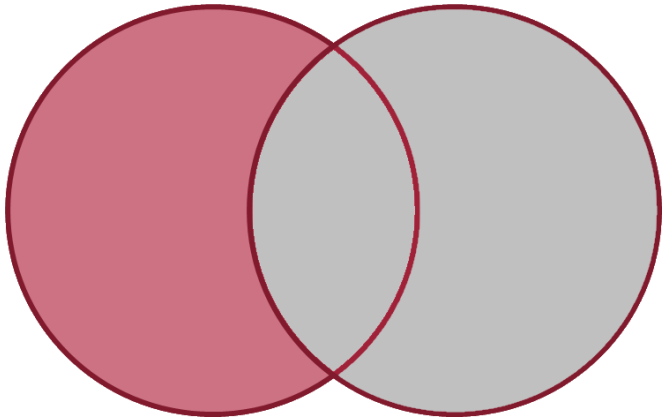
```
'leftsemi',

'left_semi'
```



```
Bit like:
"EXISTS"

"INTERSECT"
```

Anti Join

```
'leftanti',

'left_anti'
```

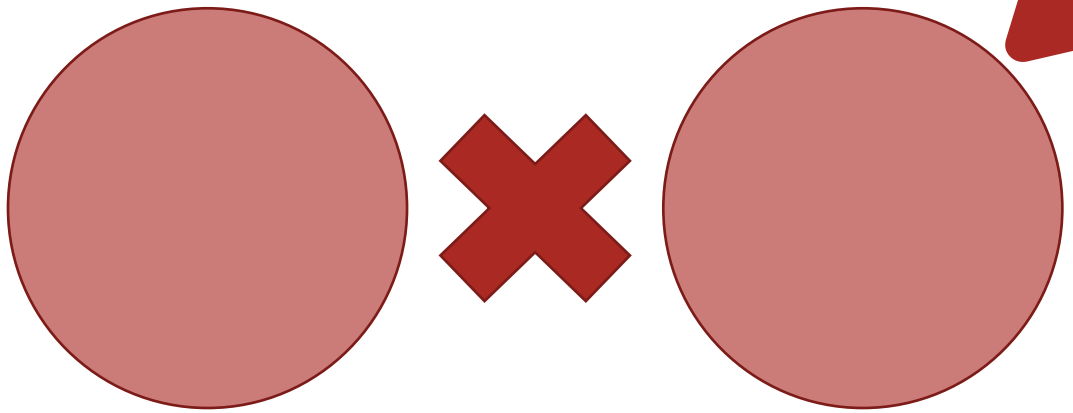


```
Bit like
"NOT EXISTS"

"EXCEPT"
```

ADVANCING ANALYTICS

Cross Join

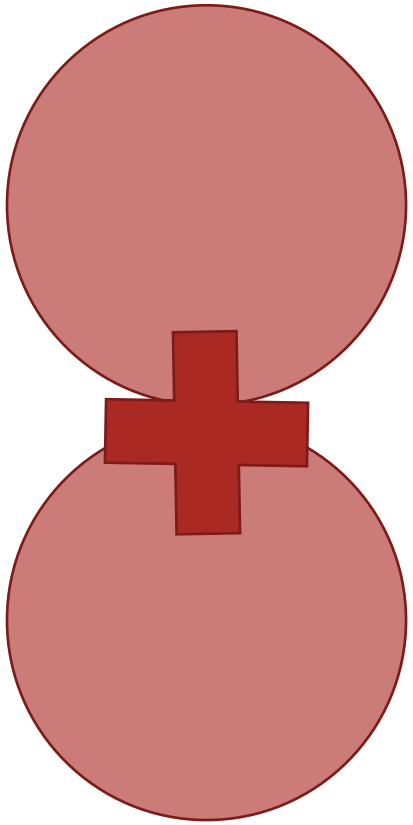
```
df.crossJoin()
```



ADVANCING ANALYTICS

Union

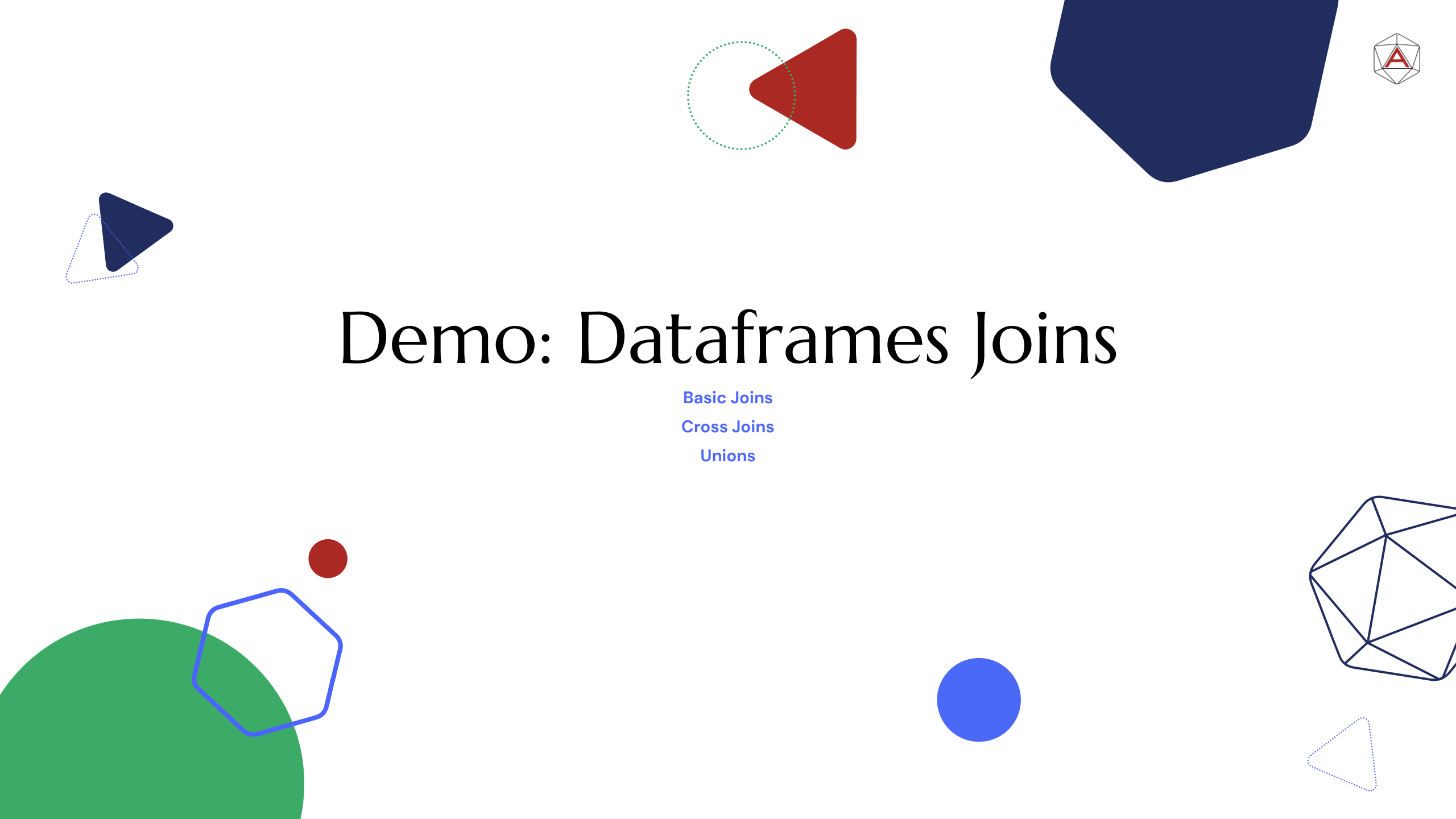
```
df.union()
```



There is a deprecated function called *unionAll()* – this is not the same as a SQL *UNION ALL* which removes duplicates!

To remove duplicate records, simply follow the union with a *.distinct()*

ADVANCING ANALYTICS



Demo: Dataframes Joins

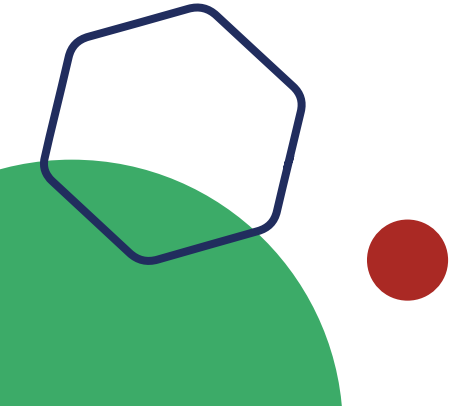
Basic Joins

Cross Joins

Unions

Dataframe Joins Summary

- Joins in Spark are very similar to SQL, except they always take the full set of columns
- Remember semi and anti-joins – they are useful
- CrossJoins are discouraged, but can be done if needed

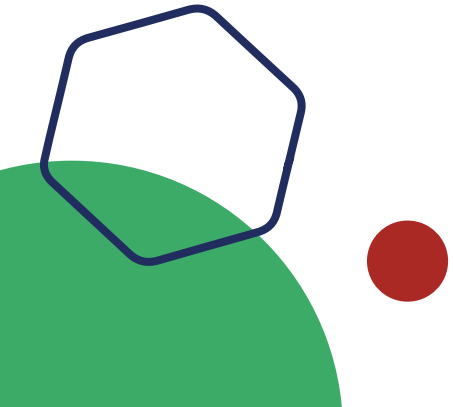


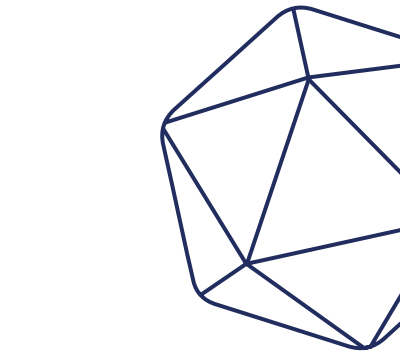
ADVANCING ANALYTICS

DateTime Functions

Datetime Functions

- Date Conversion
- Patterns
- Functions
- Date Manipulation





ADVANCING ANALYTICS

Parsing Dates

Spark has **two primary** date types:

- **TimeStampType:** Similar to SQL's datetime, this is the full date and time. It can be formatted and presented in different manners but always contains the full context. Spark's Timestamp type also includes a timezone reference, based on the session's time zone if not explicitly set.

  "2019-10-18T00:05:56.000+0000"

- **DateType:** Similar to SQL's date type, this is just the date reference

Watch out when using inferSchema - unless it is in the exact default format, it may not recognize these date/Timestamp objects and treat them as strings

ADVANCING ANALYTICS

Parsing Dates - Conversion Functions

There are several built-in functions for converting an existing column to a well-forced date/timestamp attribute:

```
# Parse an existing string in format “2020-01-28” to Date into a new column
df = df.withColumn(“myDate”,to_date(col(“DateString”), ”yyyy-MM-dd”))

# Parse an existing string in format “01/28/2020 10:14” to TimeStamp
df = df.withColumn(“myTS”, to_timestamp(col(“TSString”), ”MM/dd/yyyy HH:mm:ss”))
```

Spark 3.0 has functions to build a date/timestamp out of multiple objects:

```
# Parse an existing string in format “2020-01-28” to Date into a new column
df = df.withColumn(“myDate”, expr(“make_date(YearString,MonthString,DayString”))
```

Parsing Dates - Datetime Patterns

Symbol	Meaning	Presentation	Examples
y	year	year	2020; 20
D	day-of-year	number(3)	189
M/L	month-of-year	month	7; 07; Jul; July
d	day-of-month	number(3)	28
F	week-of-month	number(1)	3
a	am-pm-of-day	am-pm	PM
h	clock-hour-of-am-pm (1-12)	number(2)	12
K	hour-of-am-pm (0-11)	number(2)	0
k	clock-hour-of-day (1-24)	number(2)	0
H	hour-of-day (0-23)	number(2)	0
m	minute-of-hour	number(2)	30
s	second-of-minute	number(2)	55
S	fraction-of-second	fraction	978

Current Date Functions

It is very common that we would want to tag records or compare them to the current date/time. Spark has a couple of functions built in to help:

- **current_date() –** Returns the current date
- **current_timestamp()** – Returns the current timestamp, using the executor's timezone unless overridden in parameters!

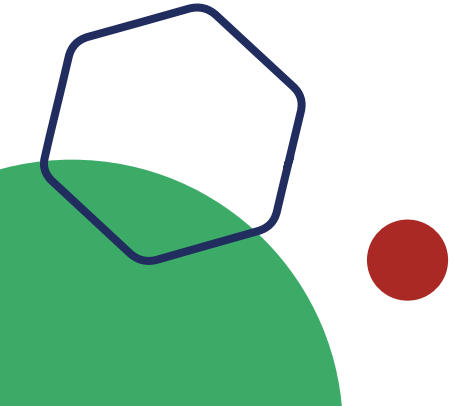
```
# Mark every record with the current date & time for auditing
df = df.withColumn("audit_time", current_timestamp())
```

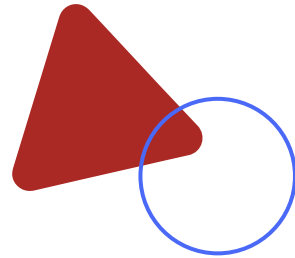
ADVANCING ANALYTICS

Date Manipulation Functions

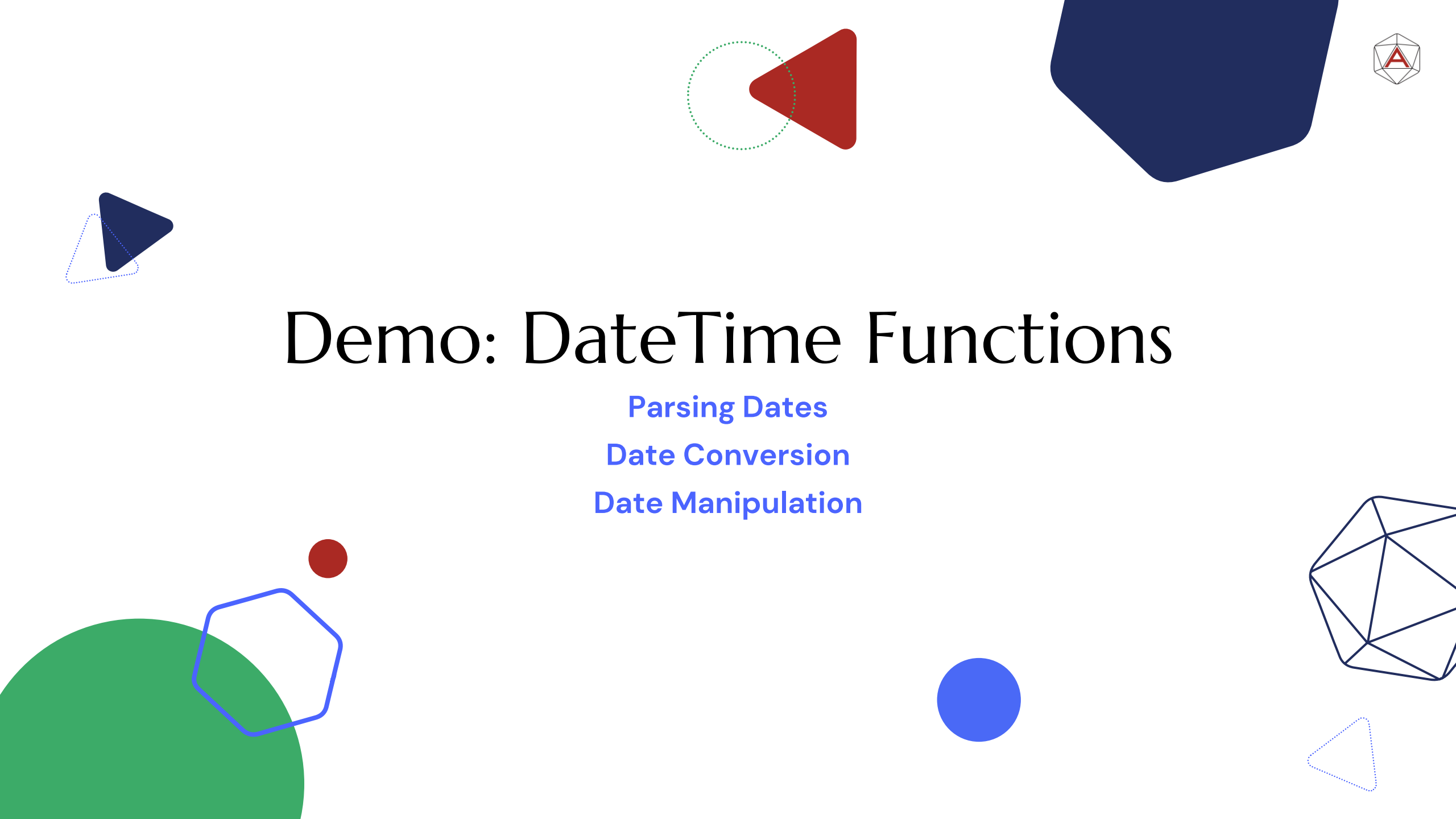
If we need to add/subtract from a date, or compare two dates, we have functions available very similar to those available in the SQL world:

- **date_add() -** Adds a number of days from a provided date/timestamp
- **date_sub()** - Subtracts a number of days *(although you can also use date_add with negative)*
- **add_months()** - Adds/Subtracts a number of months to the provided date
- **months_between()** - Returns the number of months between two dates
- **datediff()** - Returns the number of days between two dates





ADVANCING ANALYTICS



Demo: DateTime Functions

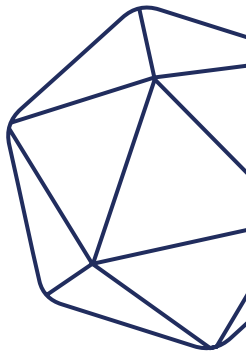
Parsing Dates

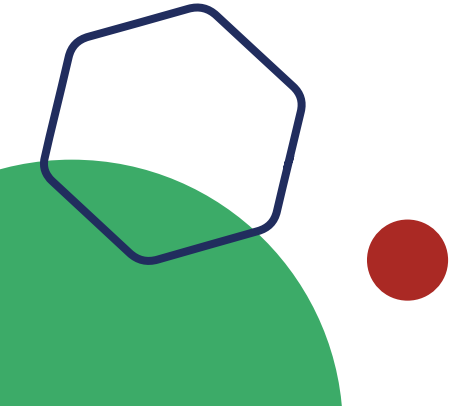
Date Conversion

Date Manipulation

DateTime Functions Summary

- Spark has many different functions built in to work with complex date manipulation
- The functions aren't quite the same as T-SQL – so be careful
- You might need to convert incoming date to the right date format before getting started!



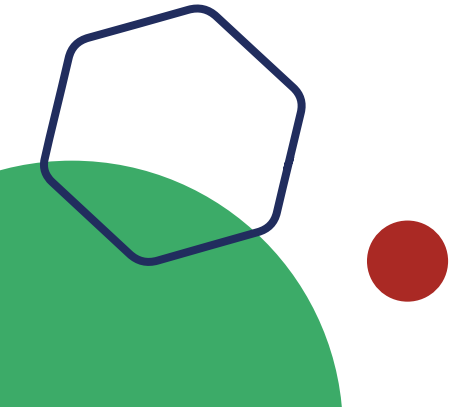


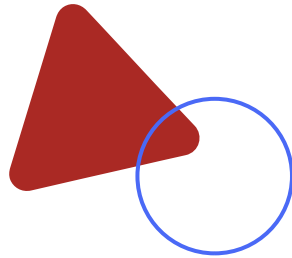
ADVANCING ANALYTICS

User-Defined Functions

User-Defined Functions

- What is a UDF
- Scala UDFs
- Python UDFs
- Vectorised Python UDFs
- Also known as a Pandas UDF.





UDF

Functions are great for encoding repeated tasks but come with some performance considerations!

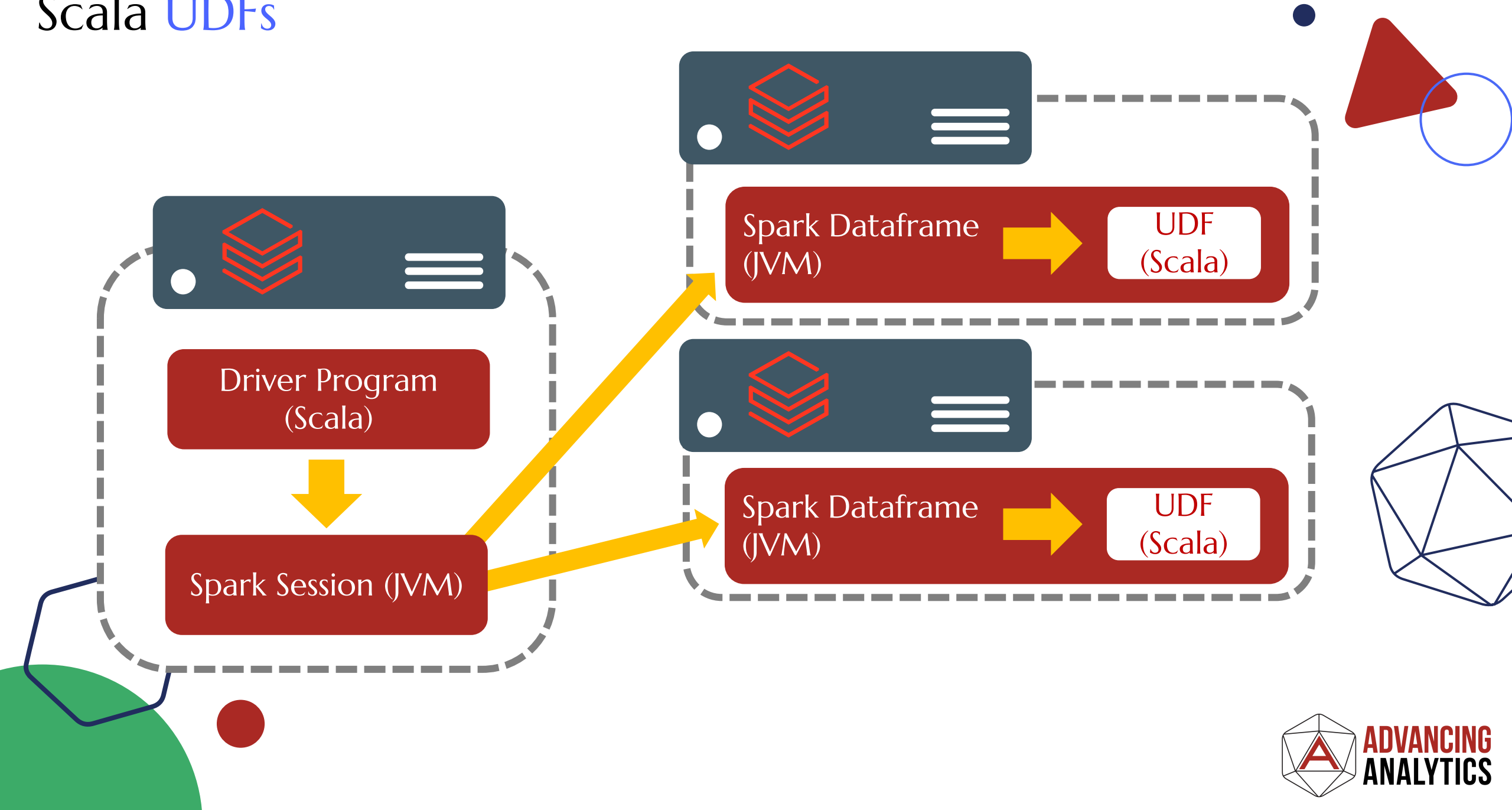
```
from pyspark.sql.types import LongType

def doubleme(x):
  return x + x

spark.udf.register("DoubleMe", doubleme, LongType())
```

Once registered, UDFs can be called on DataFrames directly, or via SQL

```
%sql
select x, DoubleMe(x) as DoubleX from MyNumbers
```

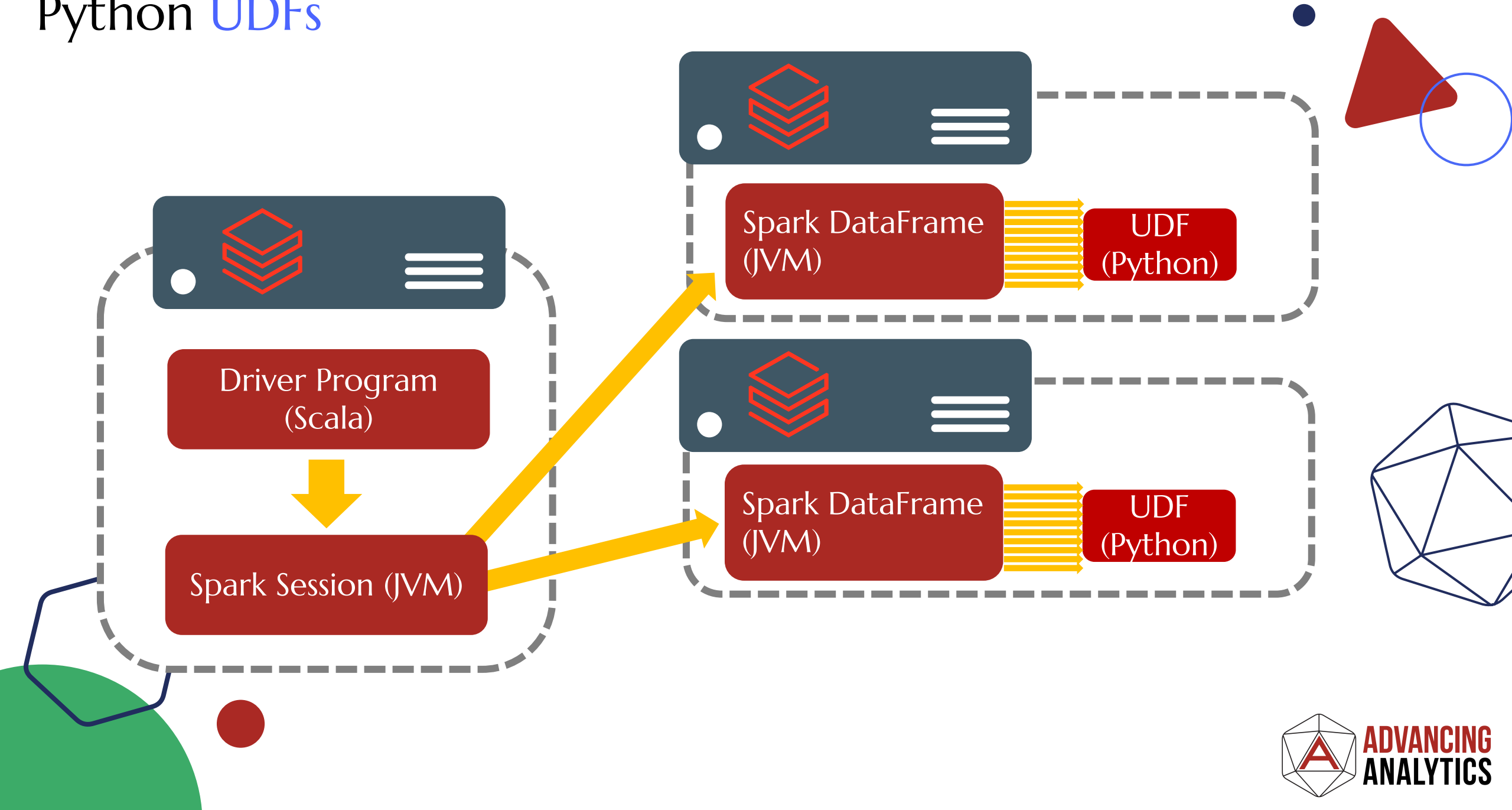
ADVANCING ANALYTICS

Scala UDFs



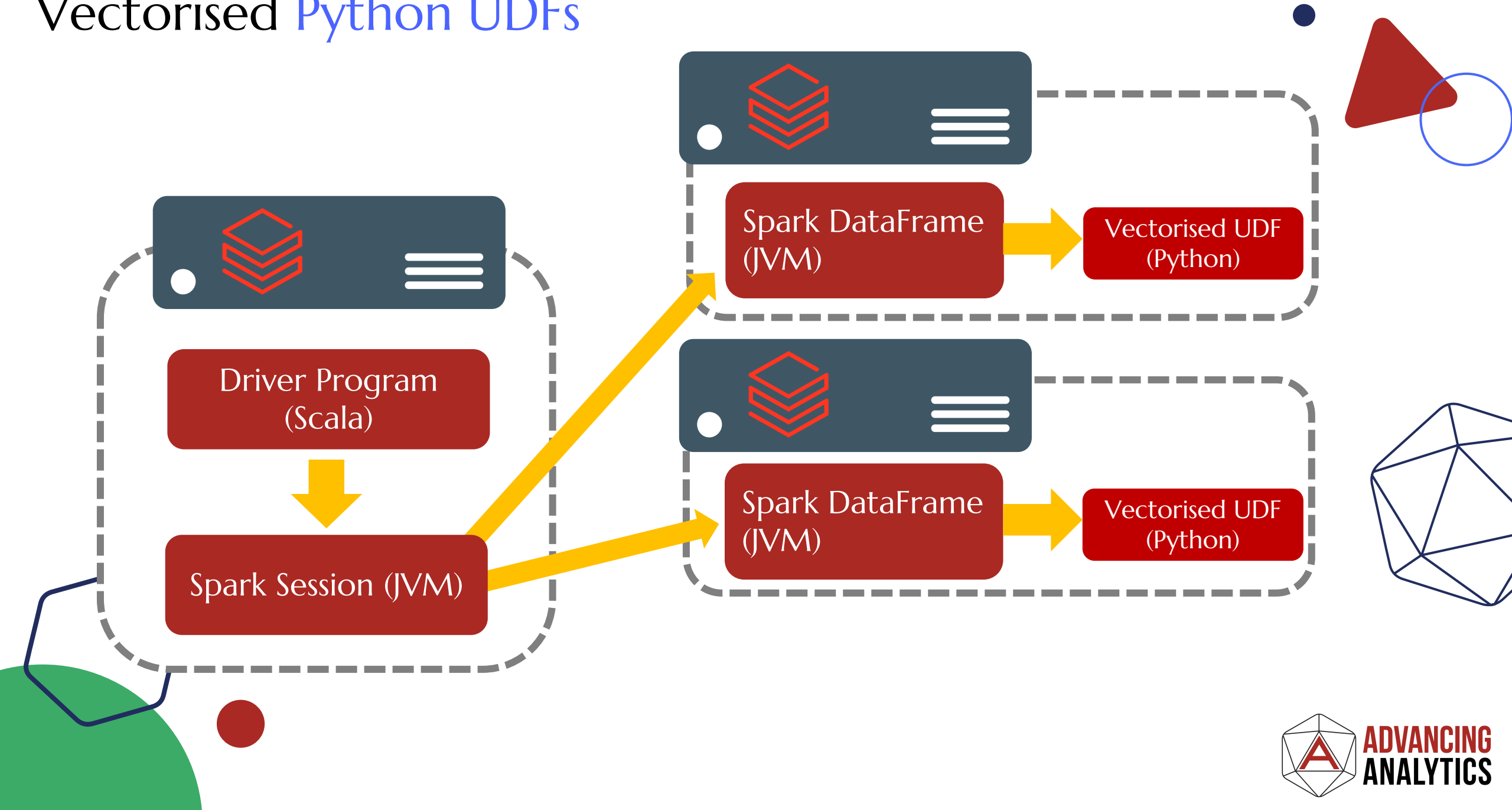
ADVANCING ANALYTICS

Python UDFs



ADVANCING ANALYTICS

Vectorised Python UDFs



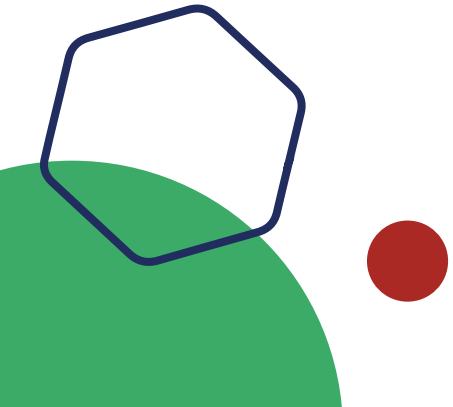
ADVANCING ANALYTICS

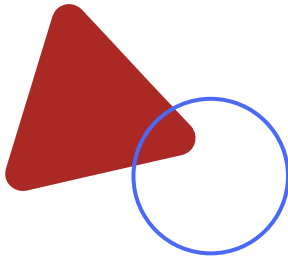
Non-UDF Functions

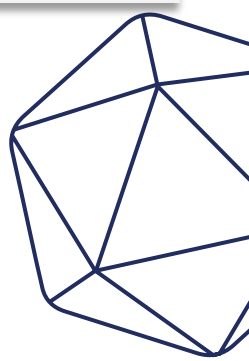
Not all functions are poor performing. We can automate tasks at the dataframe level, and this is incredibly powerful!

```
def addAuditDate(df):

        df.withColumn("_auditDate",current_timestamp())
        return df
```

Once registered, UDFs can be called on DataFrames directly, or via SQL







ADVANCING ANALYTICS

SQL UDFs

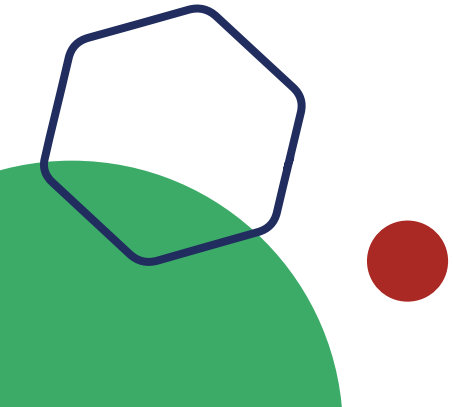
A relatively new addition to Databricks is the ability to write SQL user defined functions. Unlike pyspark / R, these functions are passed directly to the Catalyst engine and compiled natively

```
CREATE FUNCTION doubleme(x INT COMMENT ‘Doubles any number')
  RETURNS INT

  RETURN x + x
```

If you absolutely have to write a user defined function – this will likely be the most performant!

UDF Summary

- User Defined Functions often have a performance hit
- It is very rare that we cannot use the inbuilt pyspark functions instead of a UDF
- If we absolutely have to use functions:
  - Scala/SQL perform the best
  - Then Vectorised UDFs
  - Python UDFs will be slow



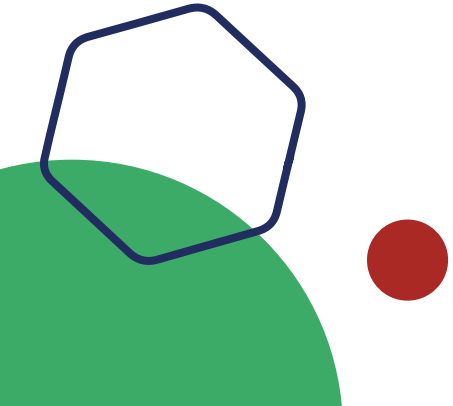
ADVANCING ANALYTICS

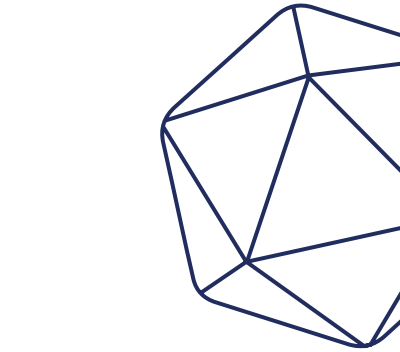
Demo: User-defined Functions

Recap

Recap

- Spark is a very powerful data transformation engine
- JSON & Complex Structures are handled easily
- Joins, Dates & Conditional Logic are familiar, but have their quirks
- Be careful with UDFs





ADVANCING ANALYTICS

ADVANCING ANALYTICS