Power BI

DAX: The language of data in power BI

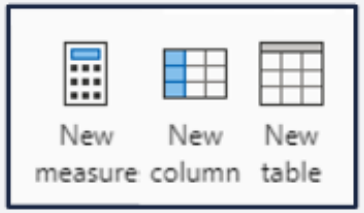
Data Analysis Expressions (**DAX**) is a **programming language** that is used in **Power BI** to **create** calculated **columns**, **measures**, and custom **tables**.

It is a **collection** of **functions**, **operators**, and **constants** that can be used in formulas to perform advanced calculations and data analysis.

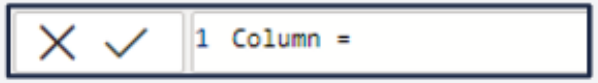
It allows us to write custom calculations using the **formula bar** and **return one or more values**.

How to input DAX in Power BI

We **input** DAX code by selecting the **Modeling** tab in **Report view** or the **Table tools** tab in **Table view** and choosing the appropriate option:



A **formula bar** pops up where we can type our DAX code:



Composing formulas in DAX

> Every **DAX formula** is a combination of **functions**, **operators**, and **constants**. Understanding how these **work together** is key to mastering DAX.

Functions	**Predefined formulas** that **perform specific calculations** using the data in our tables, such as adding up numbers or finding an average.
Operators	**Symbols** or **keywords** that specify the **type of calculation** to perform between elements in a formula, like adding with “+” or multiplying with “*”.
Constants	**Fixed values** that we directly enter into our formulas, such as a number, a text string, or a date.

Using DAX with data

We will use a subset of the **Gender_parity_2022** dataset to illustrate how we use DAX.

Country	Percent_literacy_women	Percent_literacy_men	GDP_per_capita_USD	Population_size
Egypt	67.38999939	78.77999878	4,295.40	110,990,103
Ethiopia	44.42338181	59.24145889	1,027.60	123,379,924
Ghana	76.19000244	84.48999786	2,175.90	33,475,870
Kenya	79.84999847	85.48000336	2,099.30	54,027,487
Morocco	67.37999725	84.80999756	3,527.90	37,457,971
Rwanda	73.30000305	78.69999695	966.3	13,776,698
Nigeria	52.65647888	71.25570679	2,184.40	218,541,212
South Africa	94.53317261	95.54537201	6,776.50	59,893,885

Referencing model objects

> In DAX, we can **reference entire tables**, specific **columns** within tables, or **measures** that we've already created. This allows for a **modular approach** where measures can build on each other and on calculated columns.

Table references	Column references	Measure references
Refers to the **entire table** within the data model. Used in functions that need to consider **the entire dataset**.	Refers to **specific columns** within a table. Used when we need to perform operations on **particular fields**, such as creating calculated columns or defining criteria in filters.	Refers to **measures** that have been defined in the model. Allows the **reuse** of measures within other DAX formulas, maintaining **consistency** and **reducing redundancy** in calculations.
Syntax: `'Table_name'`	**Syntax:** `'Table_name'[Column_name]`	**Syntax:** `[Measure_name]`

Calculated column

> A calculated column is a **new column** that we add to a table that is **computed from other columns**.

Unlike a regular column, a calculated column **uses a DAX formula** that **defines its values**. These are **stored** in the model and can be used in reports just like any other column.

*This isn't just about displaying new information, it's about **creating** it from what we **already have**.*

To understand the balance between female and male literacy rates, we could create a **calculated column** that **stores** the **difference** between these values.

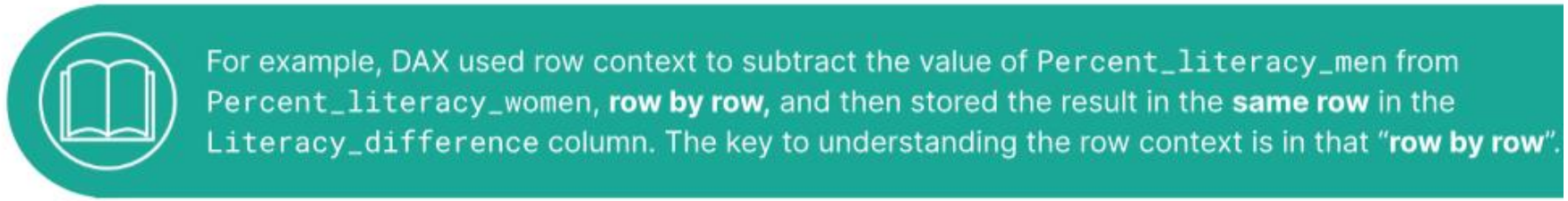
DAX:
```
Literacy_difference = 'Gender_parity_2022'[Percent_literacy_women]-
'Gender_parity_2022'[Percent_literacy_men]
```

Literacy_difference
-11.38999939
-14.81807708
-8.29999542
-5.63000489
-17.43000031
-5.3999939
-18.59922791
-1.0121994

How are values in the calculated columns are determined ?

> **Row context** tells DAX which row to use when **determining the values** for the calculated column.

We can think of the row context as the **current row** under calculation in a table. It's like having a magnifying glass that looks at each row one by one during the calculation.



For example, DAX used row context to subtract the value of `Percent_literacy_men` from `Percent_literacy_women`, **row by row**, and then stored the result in the **same row** in the `Literacy_difference` column. The key to understanding the row context is in that "**row by row**".

E.g., Use the values in this row →

Percent_literacy_women	Percent_literacy_men	Literacy_difference
67.38999939	78.77999878	-11.38999939

To determine the value in the same row in the calculated column

Other places we can use DAX

> In addition to calculated columns, DAX also empowers us to create **measures** and **tables**, to further analyse and summarise our data.

Measure

- Measures are **calculations** created using DAX that are computed **dynamically** based on the **current report context**.
- They are often used to perform **summarisation operations** like sum, average, or count, that work across the **entire table** or within the filters set by users, not just row by row.
- Unlike calculated columns, measures **do not store their results** in the database – they're re-calculated whenever needed.

Table

- Custom tables can be created using DAX to **organise data** in **new ways** or **summarise** it for **specific views**.
- They allow us to **combine data** from **multiple sources** and **store** it within the Power BI model.
- These tables can include **calculated columns** for specific analysis needs, and **measures** can be used **alongside** these tables to provide dynamic summaries.

Choosing between calculated column and measure

	Calculated column	Measure
Purpose	Used to **add new data** to an existing table, **expanding** its **detail**.	Used to **aggregate data**, providing **high-level summaries** and insights.
Evaluation	Created using **row context** and is computed during data refresh, becoming a **permanent** part of the table.	Evaluated when a query is made based on the current **filters** we have set, allowing **dynamic** calculation based on the **current view**.
Storage	Becomes part of the table's **storage** (in import mode) and can influence **report size** and **performance**.	**Doesn't store data** – it calculates **on demand** and is typically more efficient for **summarisation**.
Visual use	Can be used **like any other column** to sort, filter, and group data in reports.	Typically used in visuals to **present aggregated data**, such as totals or averages.

DAX functions

DAX functions

Functions are **prewritten formulas** that we can use to **perform operations on data**. They can be as simple as adding two numbers or as complex as filtering a table based on multiple conditions.

In DAX, these functions are enhanced by the ability to **reference various model objects** like tables, columns, and measures, enabling refined and powerful data calculations within our Power BI reports.

For example, to analyse literacy rate trends by gender, we could use the AVERAGE function on `Percent_literacy_women` and `Percent_literacy_men`.

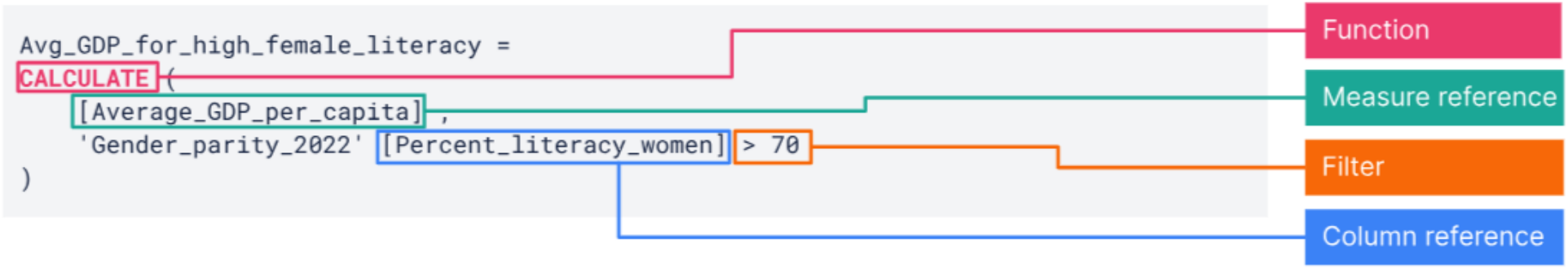
```
Average_women_literacy_rate = AVERAGE('Gender_parity_2022'[Percent_literacy_women])

Average_men_literacy_rate = AVERAGE('Gender_parity_2022'[Percent_literacy_men])
```

Bringing all together

Let's create a measure that calculates the average GDP for countries with a female literacy rate above 70%.

1. First, we define a measure called `Average_GDP_per_capita` to calculate the average GDP.

2. Next, we create a measure that calculates the average GDP for countries with a female literacy rate above 70%.



DAX data types, operators and variables

In DAX, we work with various **data types**, each tailored to specific **kinds of data**, such as numbers, text, and dates. It determines **how data can be manipulated** in calculations.

Operators in DAX, like arithmetic, comparison, string, and logical operators, allow us to **perform calculations** and **evaluate expressions**, shaping how we analyse and derive insights.

DAX **variables** enable us to **store intermediate results**, simplify complex expressions, and enhance formula readability and performance.

DAX data types

> Data types in DAX define the **kind of data** that can be stored in a column or used in expressions and calculations.

There are a few data types that we can use in DAX, but we will focus on the following:

Whole number	Decimal number	Boolean	Text
Integer values without fractions.	Real numbers with fractions.	True or false values.	Character strings.

Date	Currency	Blank
Dates and times.	Monetary values.	Represents an absence of data.

Using DAX with data

We will use the following `Country_stats` dataset to illustrate how we use DAX.

Country_ID	Region	Is_landlocked	GDP_per_capita_USD	Population_size	Area_km2	Survey_date
EGY	Northern Africa	FALSE	4,295.40	110,990,103	1,010,408.45	2019-07-16
ETH	Eastern Africa	TRUE	1,027.60	123,379,924	1,112,000.71	2022-07-28
GHA	Western Africa	FALSE	2,175.90	33,475,870	239,535.52	BLANK
KEN	Eastern Africa	FALSE	2,099.30	54,027,487	580,367.59	2022-02-01
MAR	Northern Africa	FALSE	3,527.90	37,457,971	446,300.34	2021-06-21
RWA	Eastern Africa	TRUE	966.3	13,776,698	26,338.50	2022-04-12
NGA	Western Africa	FALSE	2,184.40	218,541,212	923,769.19	2020-12-11
ZAF	Southern Africa	FALSE	6,776.50	59,893,885	1,221,037.55	2019-07-19

Operators in DAX

> Operators in DAX are **symbols** or **keywords** that are used to create expressions that compare values, perform arithmetic calculations, or work with strings.

Each operator type has a **specific function**, like arithmetic operators for mathematical calculations, comparison operators for filtering and sorting data, etc. Understanding operators is **crucial for building expressions** and **formulas**.

Think about the different data types in our `Country_stats` dataset. Do you think we can use all the operators with all the different types of data? Why or why not?

Arithmetic operators

> Arithmetic operators perform **mathematical calculations** and **produce numeric results**. They allow us to create calculated columns or measures that involve summation, subtraction, or other mathematical transformations of data.

+	-	*	/	^
Used to **add** two **numbers**.	Used to **subtract** one **number** from another.	Used for **multiplying** two **numbers**.	Used to **divide** one **number** by another.	Raises one number to the **power** of another.
Example: `1,027.60 + 2,175.90`	Example: `580,367.59 - 446,300.34`	Example: `580,367.59 * 3`	Example: `110,990,103 / 123,379,924`	Example: `239,535.52 ^ 2`

Comparison operators

> Comparison operators in DAX are used to **compare values**. When two values are compared by using these operators, the **result** is a logical value, either **TRUE** or **FALSE**.

=	==	>	>=	<	<=	<>
Checks if **A = B** Returns TRUE if value A in a column **equals** a specified value B.	Checks if **A = B** Returns TRUE if value A in a column and a specified B are **strictly equal**, i.e. have equal values or are both BLANK.*	Checks if **A > B** Returns TRUE where value **A** in a column is **greater than** a specified value **B**.	Checks if **A ≥ B** Returns TRUE where value **A** in a column is **greater than or equal to** a specified value **B**.	Checks if **A < B** Returns TRUE where value **A** in a column is **less than** a specified value **B**.	Checks if **A ≤ B** Returns TRUE where value **A** in a column is **less than or equal to** a specified value **B**.	Checks if **A ≠ B** Returns TRUE where value **A** in a column is **not equal** to a specified value **B**.

** = and == differ in handling BLANK: = returns TRUE when BLANK is compared to 0, an empty string, or FALSE; == returns TRUE only when BLANK is compared to BLANK.*

String operators

> The primary string operator in DAX is the **concatenation** operator (**&**), which **joins** two or more text strings into a **single string**.

*This operator is especially useful in **creating calculated columns** where data from different text columns need to be combined for enhanced insights.*

We could create a new column in our dataset that combines the country's `Country_ID` and `Region` into a single string, with a dash (`-`) between them.

```
DAX: Country_region = [Country_ID] & " - " & [Region]
```

Country_region
EGY - Northern Africa
ETH - Eastern Africa
GHA - Western Africa
KEN - Eastern Africa
MAR - Northern Africa
RWA - Eastern Africa
NGA - Western Africa
ZAF - Southern Africa

Logical operators

Logical operator	Description	Example
&& (AND)	Returns TRUE if **both** conditions are TRUE; otherwise it returns FALSE.	**Check if a country is landlocked and has a population over 100 million:** `[Is_landlocked] && [Population_size] > 100000000`
|| (OR)	Returns TRUE if **at least one** of the expressions is TRUE; it's only FALSE if both expressions are FALSE.	**Check if a country is in either Eastern Africa or Northern Africa:** `[Region] = "Eastern Africa" || [Region] = "Northern Africa"`
IN	Checks if a value **exists** within a **specified list** or **table**, returning TRUE if a match is found or FALSE otherwise.	**Check if the country's region is in Northern Africa:** `[Region] IN {"Northern Africa"}`
NOT	**Reverses** the **logical value** of an expression, returning TRUE if the expression is FALSE, and vice versa. It is a unary operator, i.e. it operates on only one operand.	**Check if a country is not landlocked:** `NOT([Is_landlocked])`

Operators precedence

Order of operators

If we combine several operators in a single formula, the operations are **ordered** according to the table. If the operators have equal precedence value, they are ordered from **left** to **right**.

To change the order of evaluation, we can enclose the part of the formula that must be calculated first in parentheses.

Variables in DAX

> Variables are used to **store** the **result** of an expression as a **named** value. Defining a variable involves **assigning** an expression to a **name**, which can then be **used** throughout a DAX formula.

The need for variables in DAX

Variables are essential for **storing** and **reusing intermediate** results within expressions, eliminating the need for repeated calculations of the same expression within a formula.

Variables allow for a more **organised** and **efficient** way of handling complex calculations, thereby **simplifying** our DAX expressions and improving **performance**.

Defining variables

VAR and RETURN

To **define** a variable in DAX, we use the "**VAR**" keyword, followed by the **variable name** and the **expression** we want to assign to it. **The syntax is:** `VAR VariableName = Expression`

Once variables are defined, we use "**RETURN**" to **output** the formula's result, incorporating the defined variables.

Define a variable `Max_population` to store the maximum value from the `Population_size` column, and then return this value.

```
VAR Max_population = MAX([Population_size])
RETURN
Max_population
```

Advantages of using variables

Readability and clarity

Variables make DAX formulas **easier** to **read** and **understand**, especially when dealing with lengthy and complex expressions.

Code reusability

Once defined, a variable can be **reused** multiple times within a formula, which helps in keeping the code **concise** and **manageable**.

Optimise performance

By storing intermediate results, variables can **reduce** the **number** of calculations required, thus **optimising** the **performance** of the DAX expression.

Easier debugging

Variables **simplify** the debugging process, as they allow us to **isolate** and **inspect** parts of the formula, making it easier to **identify** and fix errors

Diving deeper into DAX functions

> There are roughly **250 different functions** within Power BI, but the most popular ones can be grouped into the following **broad categories**.

Aggregation	These functions are used for **summarising** or **aggregating** data in columns or tables into a **single value**. They correspond closely to spreadsheet functions.
Count	**Count functions** are used for returning the number of rows/values in a column or table. Columns are the only accepted input for count functions.
Logical	**Logical** functions return **values or sets** based on whether an **expression** is true or false (i.e. whether a **condition** has been met or not).
Information	These functions are useful when **examining data** quality or types. A cell/row of data is provided as input and the function returns whether the cell/row value **matches** the **expected** type.

Diving deeper into DAX functions

Text	These functions are similar to **string** functions found in spreadsheets, with the ability to be applied to **columns** and **tables.**
Temporal	Used when doing calculations based on **dates** and **time**. These functions use a **datetime** data type and can use values from a column as input.
Filter	**Filter** functions in Power BI allow us to **manipulate data context**, resulting in dynamic calculations.
Other	**Power BI documentation** contains detailed information on **mathematical, financial, statistical,** and many other categories of functions that could be of use.

Aggregation functions

> These functions perform **calculations** on a set of values to produce a **single, summarising value,** often used in measures.

Syntax

```
Measure = FUNCTION(Table_name[column_name])
Measure_with_expression = FUNCTIONX(Table_name, Expression)
```

Common aggregation functions

`SUM()` Aggregates the values in a column.

`SUMX()` Aggregates the values in an expression.

`MIN()/MAX()` Returns the min/max value in a column.

`MINX()/MAXX()` Returns the min/max value of an expression.

`AVERAGE()` Returns the arithmetic mean of a column.

`AVERAGEX()` Returns the arithmetic mean of an expression.

Power BI has extensive documentation explaining syntax for all the DAX functions.

X for expression

The **X** at the end of these aggregation functions refers to **expression** and it means that we can add any expression which will then be used as the **'input column'.**

Aggregation functions

> In this example, we are interested in the **number of males living in rural areas** in the different countries in our `Gender_parity_2022` table. By multiplying the **population size** with the **proportion** of **rural male population**, we can easily see this metric.

```
Total_pop = SUM(Gender_parity_2022[Population size])
```

```
Rural_male_pop = SUMX(Gender_parity_2022, Gender_parity_2022[Population size] * Gender_parity_2022[Rural population, male (% of total)])
```

Country	Sum of Population size	Sum of Rural population, male (% of total)	Rural_male_pop
Egypt	110990103	28.85	3,201,844,210.58
Ethiopia	123379924	40.38	4,981,485,318.49
Ghana	33475870	23.18	775,831,354.42
Kenya	54027487	36.85	1,990,681,006.19
Morocco	37457971	19.89	744,989,333.09
Nigeria	218541212	25.97	5,676,327,071.01
Rwanda	13776698	33.10	456,008,814.70
South Africa	59893885	16.55	991,028,146.42
Total	**651543150**	**224.75**	**18,818,195,254.91**

Count functions

> We use **count functions** to count certain elements in a table or column. These functions follow the same format as the common aggregation functions.

Syntax

```
Count_of_something = FUNCTION(Table_name[column_name])
Count_of_some_expression = FUNCTIONX(Table_name, Expression)
```

Common count functions

`COUNT()` Returns the number of non-blank values in a column. Supports strings, integers, or dates.

`COUNTX()` Returns the number of non-blank values as the result of an expression. Uses table as input.

`COUNTROWS()` Counts the number of rows in a table. Uses table as input.

`DISTINCTCOUNT()` Counts the number of distinct values in a column, including the BLANK value.

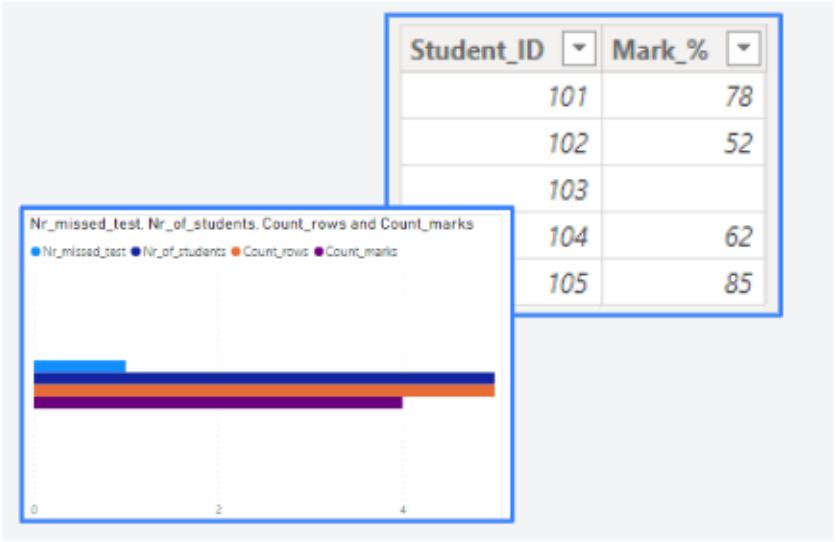
`COUNTBLANK()` Counts the number of BLANK cells in a column.

`COUNTA()` Counts the number of non-blank cells in a column. In contrast to **`COUNT()`**, it also supports Boolean values.

Count functions

> In the dataset below, we have a very small class with 5 students and a test mark per student. We want to know **how many** students didn't write the test (with a valid excuse).

```
Total_blanks = COUNTBLANK(Test_data[Mark_%])
Total_students = COUNT(Test_data[Student_ID])
Count_rows = COUNTROWS(Test_data)
Count_marks = COUNT(Test_data[Mark_%])
```



Why care about BLANKS?

Missing values might influence our dashboard's reliability. If a large proportion of data points are missing, the **summary** might present a **skewed view.**

Logical and conditional functions

> These functions enable us to make use of **conditional logic** when creating new columns.

Common logical functions

`AND()` Returns TRUE if both arguments are TRUE.

`OR()` Returns TRUE if one of the arguments is TRUE.

`NOT()` Returns TRUE if the condition is not met.

`IF()` Evaluates a condition, if TRUE, it returns one value, otherwise it returns the second value.

`IFERROR()` Evaluates an expression and returns an alternative pre-specified value if the expression results in an error.

`SWITCH()` Evaluates an expression (or column) against a list of values and returns one of the pre-specified results. By using Boolean values in the expression, we can also use this instead of nested `IF()` statements.

Important

Logical functions can only be used on **columns.**

If/ Switch example

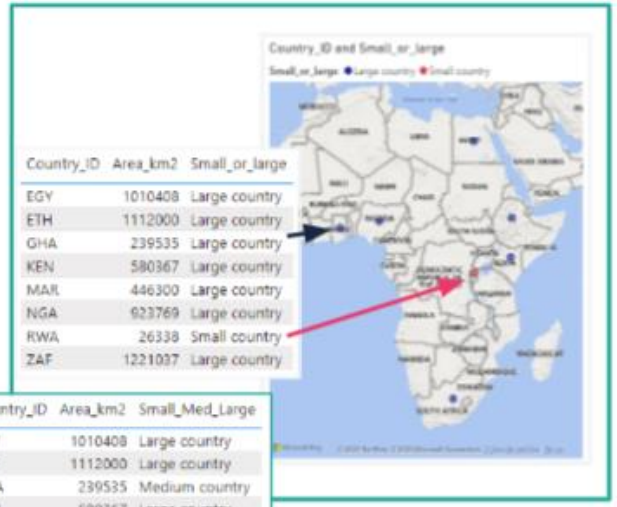
> Using our Demographic_info dataset, we're interested in creating a column that would **categorise countries as small or large,** based on the `Area_km2` column. The second code block shows a **nested** version of **`IF()`**, based on three categories for the country sizes.

```
Small_or_large = IF(Demographic_info[Area_km2] > 100000,
"Large country", "Small country")
```

```
Small_Med_Large = IF(Demographic_info[Area_km2] > 500000,
"Large country",  IF(Demographic_info[Area_km2] > 100000,
"Medium country", "Small country"))
```

This exact statement can also be written with a **`SWITCH()`** function:

```
Small_Med_Large = SWITCH(TRUE,
Demographic_info[Area_km2] > 500000, "Large country",
Demographic_info[Area_km2] > 100000, "Medium country",
Demographic_info[Area_km2] > 0, "Small country")
```



Information functions

> These functions operate on a cell/row level and evaluate whether the input argument is what is expected. If it is, the function returns **TRUE**, otherwise **FALSE**. They are often used to perform calculations wrapped within an **IF()** statement, to prevent errors in the results.

Common information functions

`ISBLANK()` Checks whether a value/expression is BLANK.

`ISNUMBER()` Checks whether a value/expression is numeric.

`ISTEXT()` Checks whether a value is of type text.

`ISNONTEXT()` Checks whether a value is non-text.

`ISERROR()` Checks whether value/expression results in error.

`CONSTAINSSTRING()` Checks if the specified text is found in a string.

Case sensitive?

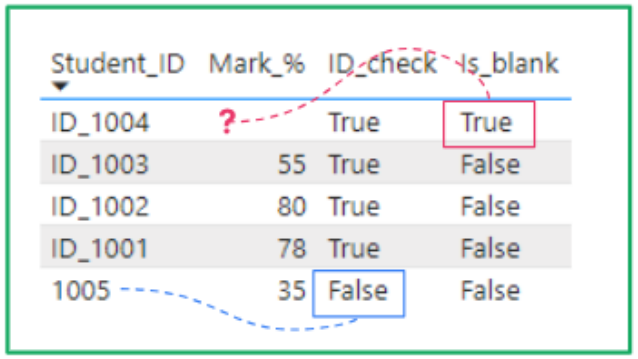
Certain functions are not case sensitive, such as **`CONTAINSSTRING()`**. If we are trying to find a case sensitive string, we'd use **`CONTAINSSTRINGEXACT()`** instead.

Information functions

Example

> We need to determine whether all the **Student_IDs** in our data are in the **right format** (contains "ID_" as a prefix), and also whether the **marks** are **populated**. We can use two information functions to solve this issue.

```
ID_check = CONTAINSSTRING(Test_data2[Student_ID],"ID")

Is_blank = ISBLANK(Test_data2[Mark_%])
```



Student_ID	Mark_%	ID_check	Is_blank
ID_1004	?	True	True
ID_1003	55	True	False
ID_1002	80	True	False
ID_1001	78	True	False
1005	35	False	False

Power BI BLANK functions

BLANK cells in Power BI can display different behaviours depending on column types. It is good practice to check that **ISBLANK()** and **COUNTBLANK()** functions behave in the way assumed, especially if used for quality checks.

Text functions

> **Text columns** frequently need to be **manipulated** in some way in order to be used in **visualisations**. Power BI functions can help us create columns in a **specific format**, with the benefit that they correspond closely to string functions found in spreadsheets and SQL.

Common text functions

`SUBSTITUTE()` Replaces existing text with new text in a text string. Text to be replaced is explicitly stated.

`REPLACE()` Replaces part of a string with new text, based on the number of characters and position specified.

`SEARCH()` Returns the number of the character at which a specific string/character first occurs in text, from left to right. Can specify starting position for the search.

`FIXED()` Rounds a number to specified decimals, then converts to string.

`CONCATENATE()` Joins two text strings together into one.

`UPPER/LOWER()` Converts the case of a text string to all uppercase or all lowercase.

`LEFT()` Returns the specified number of characters from the start of a text string.

`RIGHT()` Returns the last character or characters in a text string, based on the number of characters specified.

String manipulation example

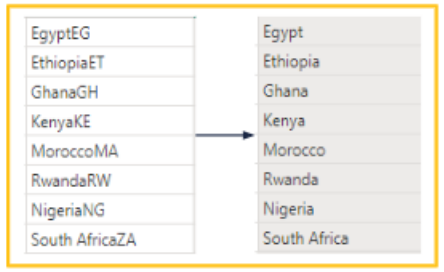
> Our `Country` column in our dataset accidentally includes the first two letters of the `Country_ID` column, as **something went wrong** during the data processing phase. We need to **clean the data** by making use of some **text manipulation functions.**

Column of interest

```
Country_new = REPLACE(Country_simple[Country],
LEN(Country_simple[Country]) - 1 ,2 ,"")
```

Getting the position of where the replacement text should go, in this case the length of the name, -1 position

Specifying that we want to replace 2 characters, with nothing ("")



EgyptEG	Egypt
EthiopiaET	Ethiopia
GhanaGH	Ghana
KenyaKE	Kenya
MoroccoMA	Morocco
RwandaRW	Rwanda
NigeriaNG	Nigeria
South AfricaZA	South Africa

Power BI documentation

Here we used **`LEN()`** (returning the length of a string) within our text manipulation formula. **Power BI documentation** contains a full list of useful functions not directly covered here.

Temporal functions

> By using temporal functions in Power BI, we can do **time-based analysis** on columns. It is useful for productivity measures, comparisons over different periods, and determining person-level indicators such as age.

Common date functions

`DATE()` Returns a specified date in datetime format.

`HOUR()` Returns the hour as a number from 0 to 23, based on 24-hour time notation (00:00 - 23:59).

`WEEKDAY()` Returns the day of the week as a number (1-7).

`NOW()` Returns the current date and time in a datetime format.

`EOMONTH()` Returns the date in datetime format of the last day of the month, before or after a specified number of months.

`DATEDIFF()` Counts the number of intervals between two dates. Intervals can be seconds up to years.

Date and time example

> We have a small dataset (`Grants`), but the day, month, and year have been captured in **separate columns**. Convert these columns to a **single date field**, showing the **actual day** of the week, and then calculate **how many months** have gone by between the end and start date.

```
Start_date = DATE(Grants[Start_year], Grants[Start_month], Grants[Start_day])
Weekday_nr = WEEKDAY(Grants[Start_date], 1)
Weekday_format = FORMAT(Grants[Weekday_nr], "DDD")
Diff_months = DATEDIFF(Grants[Start_date], Grants[End Date], MONTH)
```

Start_day	Start_month	Start_year	Start_date	Weekday_nr	Weekday_format	End Date	Diff_months
1	6	2022	6/1/2022 12:00:00 AM	4	Wed	Saturday, December 31, 2022	6
1	7	2023	7/1/2023 12:00:00 AM	7	Sat	Friday, December 01, 2023	5
3	8	2022	8/3/2022 12:00:00 AM	4	Wed	Thursday, December 01, 2022	4
4	6	2023	6/4/2023 12:00:00 AM	1	Sun	Friday, December 01, 2023	6
7	7	2020	7/7/2020 12:00:00 AM	3	Tue	Friday, January 01, 2021	6

Filter and relationships function

> **Filter functions** in DAX enable us to manipulate the data context to create dynamic calculations, whereas **relationship functions** help to manage and utilise relationships between tables.

Common filter functions

`ALL()` Returns all rows in a table, ignoring any filters that may have been applied.

`CALCULATE()` Evaluates an expression in a modified filter context.

`CALCULATETABLE()` Evaluates a table expression in a modified filter context.

`FILTER()` Returns a table that is a subset of another table or expression.

`LOOKUPVALUE()` Returns a row based on criteria specified in a search condition. There can be multiple search conditions.

`USERELATIONSHIP()` The function specifies a relationship to be used in a specific calculation.

`RELATED()` Returns a related value from another table.

Filter Example

> We are interested in creating a table (E_Africa) that contains a **subset** of the data in our original table. We also want to add the Country name, a column from another table. The linking columns have already been identified and relationships are set up.

```
E_Africa = FILTER(Demographic_info,
Demographic_info[Region] == "Eastern Africa")
```

Country_ID	Region
ETH	Eastern Africa
KEN	Eastern Africa
RWA	Eastern Africa
EGY	Northern Africa
MAR	Northern Africa
ZAF	Southern Africa
GHA	Western Africa
NGA	Western Africa

Country_ID	Region
ETH	Eastern Africa
KEN	Eastern Africa
RWA	Eastern Africa

```
Country_name =
RELATED(Country_simple[Country_name])
```

Country_ID	Region
ETH	Eastern Africa
KEN	Eastern Africa
RWA	Eastern Africa

Country_ID	Region	Country_name
ETH	Eastern Africa	Ethiopia
KEN	Eastern Africa	Kenya
RWA	Eastern Africa	Rwanda