**``Welcome to the Universal Tennis Style Guide**

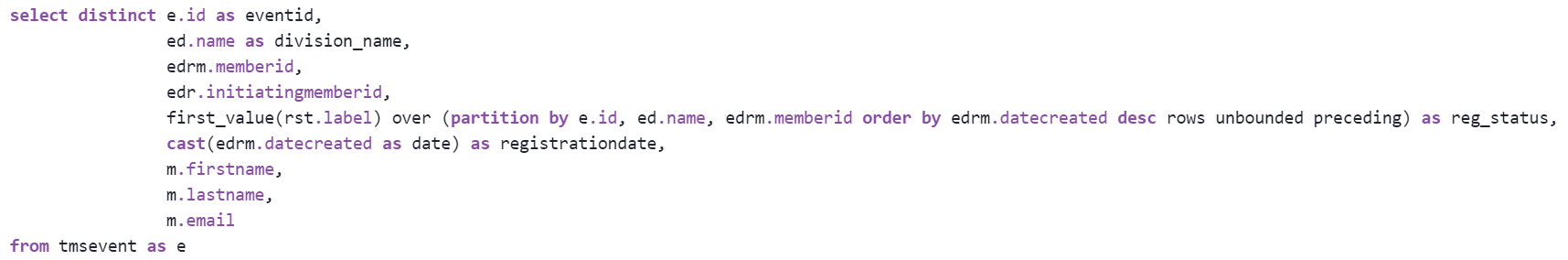
**This document will help answer your questions on how to write and format code, as well as interact with various tools you may or may not use**

# SQL Etiquette

## All

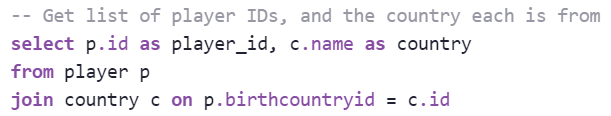
* + Put each clause on a separate line
    - **EXCEPTION:** if the query is very short and can fit on a single line, then separate lines are not necessary
  + Within a single clause, line up each new argument with the argument above it
  + **Optional:** Put SQL terms (select, from, where, and, or, as, group by, etc.) in ALL CAPS

## SELECT clause

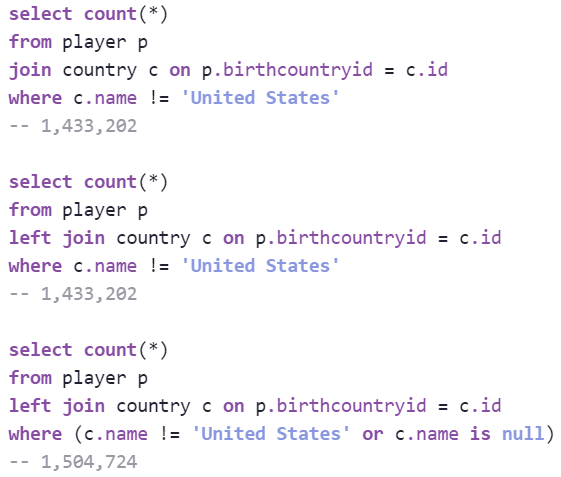
* + Put each field on a separate line
    - **EXCEPTION:** if all fields fit on one line, then new lines are not necessary
  + ALWAYS include table name/alias using dot notation, unless query only uses one table (see **ALIASES**)
  + Give each column name an alias (see **ALIASES**)
  + Example:

## FROM/JOIN clause

* + Any and all joins should be on their own lines
  + Use aliases for table names (see **ALIASES**)
  + **JOINS**
    - **JOIN** - inner join; takes only rows that appear in BOTH tables
    - **FULL JOIN** - outer join; takes rows that appear in either or both tables
    - **LEFT JOIN** - takes all rows that appear in first table, and any rows from second table that can be mapped to first table (all others are dropped)
    - **ON statement** - how to map the two tables to each other
      * Often an ID column that appears in both tables
  + Example:

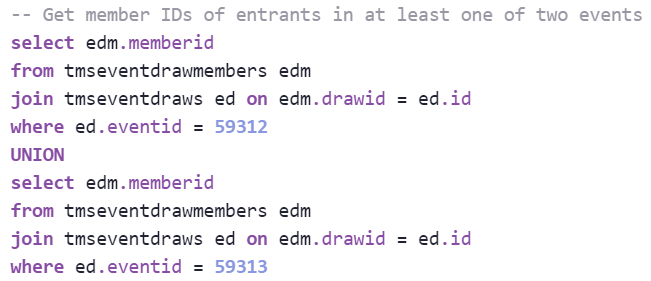


* + - **NOTE ON FULL AND LEFT JOINS:** if using a field from the right table in a WHERE clause, the condition **MUST** allow the field to be null. Otherwise, all rows with null values for the field will be removed from the output table, effectively turning the join into an inner join.
      * Example:



## UNION operators

* + Use on its own line, between two queries
  + **Optional:** add empty lines before and after UNION
  + Column names/aliases in all queries **MUST** match exactly, in order
  + ORDER BY can only be used after the final query (can only sort full output, not individual results along the way)
  + **Types of unions**
    - **UNION** - includes all rows from both queries, removing duplicates
    - **UNION ALL** - includes all rows from both queries, retaining duplicates
    - **INTERSECT** - includes only rows that appear in both queries, removing duplicates
    - **EXCEPT** - includes only rows that are in the first query, and not in the second query, removing duplicates
  + Example:



## WHERE clause

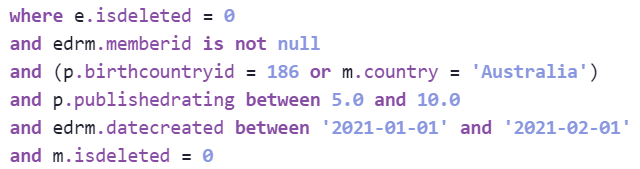
* + **General**
    - Will often cover multiple lines
    - Cannot filter on aggregate fields (see **HAVING clause**)
  + **AND statements**
    - Split into separate lines
      * No additional indentation on new lines (including the first one after the WHERE)
  + **OR statements**
    - Enclosed by parentheses
    - Keep in one line, unless it won’t all fit on one line
      * Nested AND/OR conditions inside parentheses can be kept in one line, or split into multiple, to make it as easy to read as possible
      * If splitting across multiple lines, each subsequent line should line up with the first character inside the parentheses
  + **Between**
    - Used for a value that falls within two **numbers**, or two **dates**
    - INCLUSIVE - values that touch either side of the range are considered to be inside the range, and will be included in the results
      * **NOTE:** a date without a timestamp (eg, ‘2021-07-01’) is assigned the timestamp ‘00:00:00’, so if one is used as the upper limit in a BETWEEN statement, the query will NOT include values on the upper limit’s date (unless a value also doesn’t have a timestamp, or it has the exact timestamp ‘00:00:00’)
        + Example: WHERE startdate BETWEEN ‘2021-01-01’ AND ‘2021-02-01’

‘2021-02-01’ is in the range

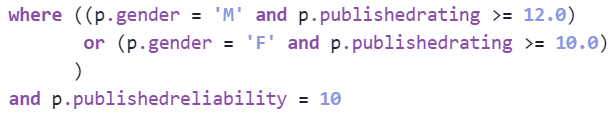
‘2021-02-01 00:00:00’ is in the range

‘2021-02-01 05:17:30’ is NOT in the range

* + **IN**
    - Used to check if a value is in a list
    - Case-sensitive
    - Values in list can be any format (character, integer, date, etc.), though it is most often used for character values
    - Useful alternative for multiple AND statements
  + **LIKE**
    - Used for fuzzy matching
    - character fields only
    - NOT case-sensitive
    - Wildcard symbol (%) can match zero, one, or multiple characters
    - Example:
      * “eventname LIKE ‘%flex%’” will match the eventnames ‘Flex Leagues’, ‘UTR Flex Leagues’, etc.
      * “eventname LIKE ‘flex%’” will match ‘Flex Leagues’, but it will NOT match ‘UTR Flex Leagues’
  + Example 1:



* + Example 2:

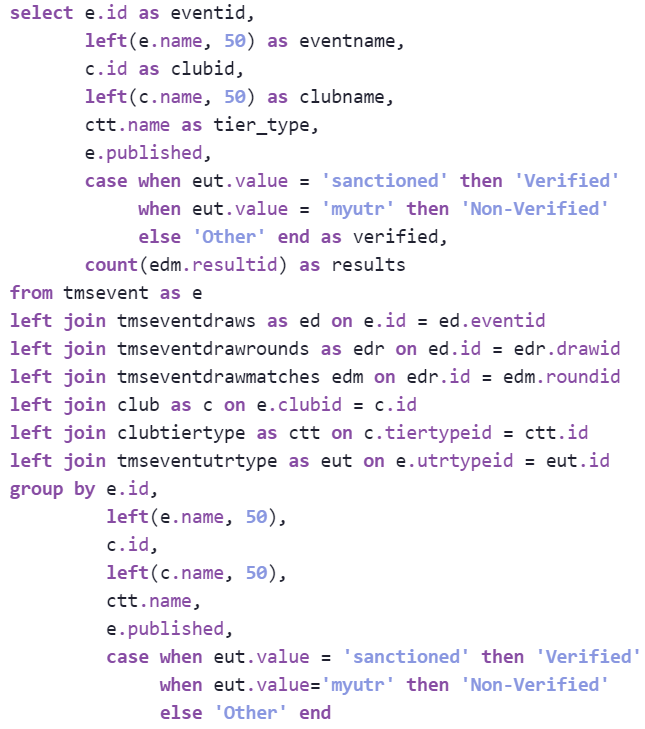


## Aliases

* + A temporary name for a column or table, that can be referenced later (instead of the full name)
  + Column names
    - Proceeded with “as” (or “AS”)
    - Ideally a full name, separated by underscores (snake case), all lowercase
    - Optional when taking a field name directly from the table, but recommended if it will make the column more clear (such as turning a column called **id** into **member\_id**)
    - Should always be used when the column is a formula
    - Do not use an alias if the column name is not changing
    - Examples:
      * datecreated
      * id as member\_id
      * datediff(‘day’, playerbirthdate, getdate())/365.25 as player\_age
      * ~~eventstartdate as eventstartdate~~
  + Table names
    - Optionally proceeded with “as” (or “AS”)
    - Single letter or initials are usually best
    - **All tables should have an alias, unless the query only pulls from one table**
    - Examples:
      * player as p
      * clubtiertype as ctt
    - Use dot notation (alias.field; eg, p.id) for **ALL** fields in all parts of query
      * **EXCEPTION:** if no table alias is used (ie, the query only pulls from one table), then dot notation is not necessary

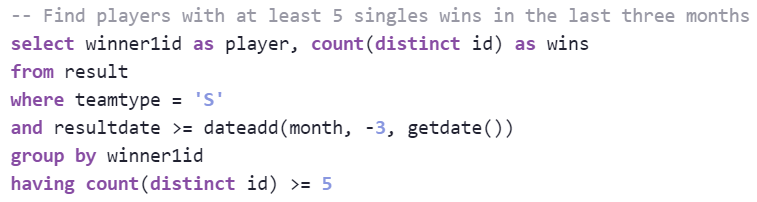
## GROUP BY clause

* + Creates categories by which to aggregate values
  + Required when mixing aggregate functions with standalone columns
  + All non-aggregated columns must appear in the group by clause, exactly as they appear in the SELECT clause
    - Cannot use column aliases; must use original column name or formula used to define it
    - Can often copy/paste the whole SELECT clause (except for the aggregated columns), but be sure to delete any aliases assigned to columns
  + Put the columns in the same order that they appear in the SELECT clause
  + Put each field on a separate line, and indent subsequent lines so they all line up with the field in the first line
    - **EXCEPTION:** if all fields fit on one line, then new lines are not necessary
  + Do not use DISTINCT in the SELECT clause, if using a GROUP BY (the DISTINCT is implied in this case, and is therefore not necessary)
  + Example:



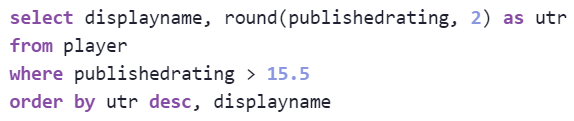
## HAVING clause

* + Used when filtering on an aggregate condition (eg, players with at least three matches, teams with a minimum UTR of 8.0 or higher, etc.)
  + Must appear immediately after GROUP BY clause
  + Example:



## ORDER BY clause

* + Determines how to sort the output rows
  + Can specify multiple columns
    - First column to be specified is used to determine the order, with ties being resolved using the second column specified, and then the third, etc.
  + Use “desc” immediately after a column to sort it from high to low
  + Can use column indices instead of column names
    - eg, ORDER BY 1 sorts by the first column in the output
    - Do not mix column names and numbers; use only names or only numbers
  + When using column names, okay to use an alias (see examples)
  + Can sort by columns that are not in the output, though usually this is not recommended
  + **NOTE:** ORDER BY is expensive, and can slow down large queries significantly. Do not sort by more columns than necessary
  + Examples (both of these queries are equivalent):
    - Example 1:



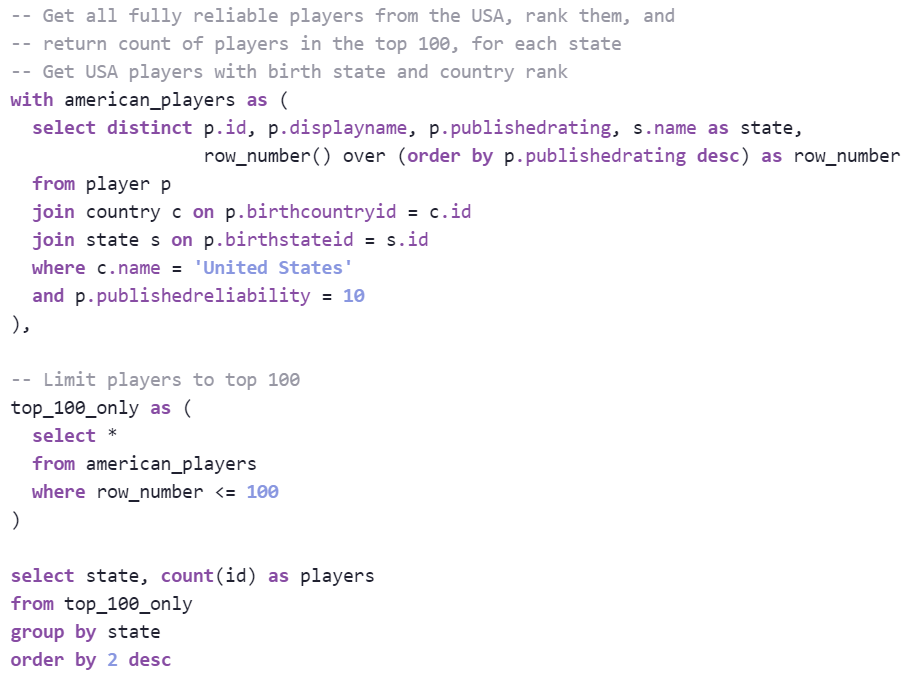
* + - Example 2:



## CTEs and Subqueries

### Common Table Expressions (CTEs)

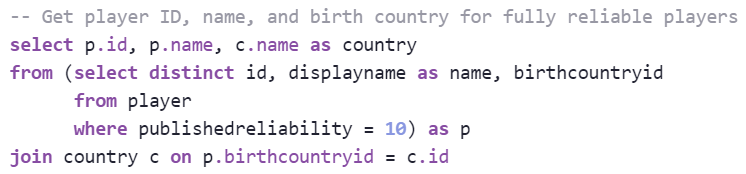
* + - A temporary table that is immediately used in a following query
    - Must appear at the start of a query
    - Multiple temporary tables/CTEs are allowed, and later ones can query from earlier ones
    - All tables/CTEs MUST have an alias
      * Unlike other uses, CTE aliases appear BEFORE the expression
    - **Notation:**
      * First table/CTE is preceded by “with” - all subsequent tables/CTEs (if any) are not
      * Table/CTE name (alias), followed by “as” and then open parenthesis
        + Alias is ideally a full name, separated by underscores (snake case), all lowercase
      * Table query starts on next line; all lines of query indented two spaces (or one tab)
      * At end of query, put a closed parenthesis on a new line, with no indentation
      * If creating multiple tables, the end of every expression (the closed parenthesis) must be followed by a comma, except for the last expression
      * Add an empty line between CTEs, and after the last one
      * Cannot use ORDER BY within a CTE
      * Example:
        + WITH <table\_1\_alias> AS (  
           <query>  
          ),  
            
          <table\_2\_alias> AS (  
           <query>  
          )  
            
          SELECT …  
          FROM <table\_1\_alias> as t1  
          JOIN <table\_2\_alias> as t2
    - Example:



* + - **WARNING:** be very careful when using CTEs in combination with the LIMIT PREVIEW option. The limit will be applied to the *final* SELECT statement in the query, which may be hidden inside a view or a subquery

### Subqueries

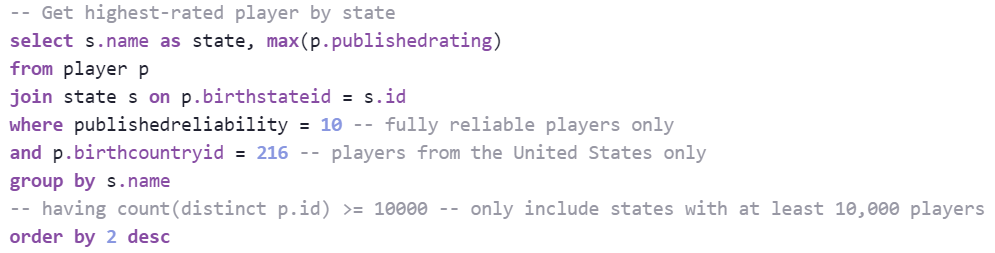
* + - A query defining a table, that appears within a FROM or JOIN statement in another query
    - Notation:
      * Start subquery on same line as FROM/JOIN statement
      * Surround the whole query with parentheses
      * Put each clause on a separate line, unless the whole query fits on one line (this is rare)
      * Indent lines the same way as a standard query is indented, with the start of each line lining up with the first character after the query’s opening parenthesis
      * End the query with a closed parenthesis, alias for the table, and ON statement, all on the same line
        + Put these on the same line as the final line of the subquery, unless it doesn’t fit on one line, in which case line up the closed parenthesis with the opening parenthesis
    - Only include necessary columns in subquery’s SELECT statement - do not use \*, or include any fields that will not be used in the main query
    - Cannot use ORDER BY in a subquery
    - Example:



* + **When to use a CTE or Subquery - which one is better?**
    - **Use a CTE if:**
      * The query is long, or has lots of conditions
      * Multiple temporary tables are required
      * The query is NOT being used for a view (views don’t like CTEs)
    - **Use a subquery if:**
      * The query is short
      * The query is being used for a view (views don’t like CTEs)
    - In general, **CTEs** are cleaner and easier to read, and are usually the preferred method
    - ~~Try not to combine CTEs and subqueries - use one or the other for all temporary tables in a query~~

## Comments

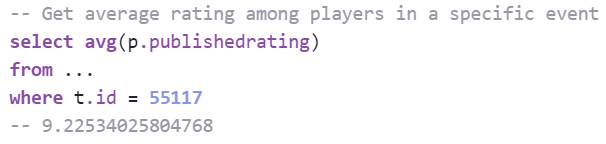
* + Two dashes, followed by a space
    - -- <commented text>
  + Query will ignore everything from the dashes and space to the end of the line
  + **Shortcut:** click on one or more lines, and push **ctrl-/**, and all selected lines in full will be commented out, or uncommented
  + **NOTE:** SiSense sometimes has trouble with comments in a query, especially if they appear at the bottom of the workspace. Best practice is to delete any unnecessary comments once the query is completed.
  + **Comment etiquette**
    - Start of query: put the comment one line above start of query, to provide a description of what the code does
    - Mid-query: put the comment at the end of the line, to describe what the line does
    - Commenting out full lines: comment out an entire line to “turn it off,” so the rest of the query can still be run
      * Be careful of code sections that span multiple lines
  + Example:



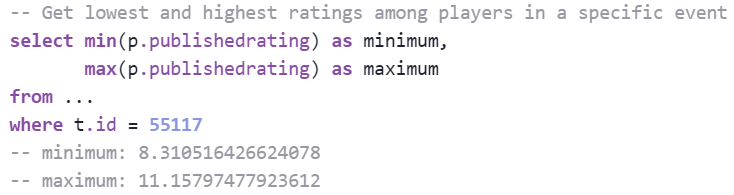
## General tips, and commonly-used functions

### Aggregate functions

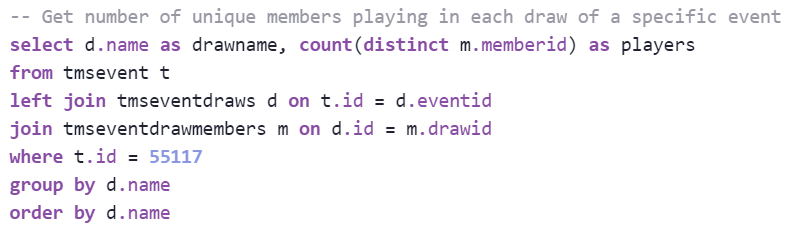
* + - Aggregate over a column or columns, to get summary statistics
    - **Must include a GROUP BY clause when aggregating** (see **GROUP BY clause**)
      * **EXCEPTION:** if not including any non-aggregated columns, then GROUP BY clause is not required
    - Available functions:
      * count (number of rows in group)
      * min/max (lowest/highest numeric value in group)
      * avg (arithmetic mean of numeric values in group)
      * sum (combined sum of numeric values in group)
    - Can use DISTINCT inside an aggregate function, most commonly in “count” (see **DISTINCT**)
    - Character values can also be aggregated, though they may give unexpected results (except for count)
    - Example 1:



* + - Example 2:



* + - Example 3:



### DISTINCT

* + - Removes any exact copies of a row (all columns must match exactly)
    - “DISTINCT” appears immediately after “SELECT” (eg, SELECT DISTINCT id, …)
    - Can also be used in aggregate functions
      * “SELECT count(id)” counts all rows that have a nonnull id value
        + If a single id appears five times in the table, it is counted five times
      * “SELECT count(distinct id)” counts all unique nonnull id values
        + If a single id appears five times in the table, it is only counted once
    - Should only appear in the SELECT clause, or in an aggregate function (including one in a HAVING clause)
    - **Do not start a SELECT clause with DISTINCT when using a GROUP BY clause**

### COALESCE

* + - Accepts list of fields or values as input, and returns the first one that is not null
    - Example 1:
      * coalesce(p.birthdate, m.birthdate)
      * Returns the birthdate value from the p table if it is not null, otherwise returns the birthdate value from the m table (returns null if both values are null)
    - Example 2:
      * coalesce(gender, ‘Unknown’)
      * Returns the gender value if it exists, otherwise returns ‘Unknown’
      * Effectively turns nulls into the character value ‘Unknown’

### CASE WHEN

* + - Takes list of criteria, and values to assign for each, and selects the value of the first condition to match
    - Used primarily in SELECT statements, but can also be used in aggregates and window functions, and in ORDER BY clauses
    - Format:
      * CASE WHEN <condition 1>

THEN <value 1>

WHEN <condition 2>

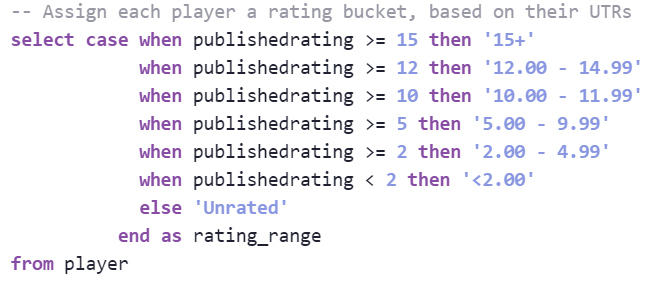
THEN <value 2>

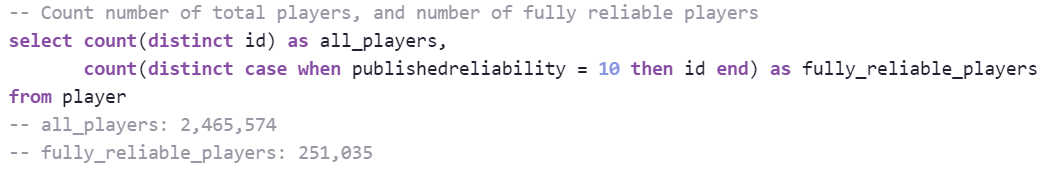
…

[ELSE <value 3>]

END as alias

* Note the indentation - line up all the WHENs
  + THEN can be on the same line as the WHENs if **ALL** cases can fit on one line
    - ELSE is an optional catchall; any values that don’t match any of the WHEN conditions will be given the ELSE value
      * Indent the ELSE as if it were another WHEN
    - Statement **MUST** be closed out with “END” (or else the query will fail)
    - Example 1:



* + - Example 2:
      * 

### Window functions

* + - Used to perform calculations over a set or sets of rows
    - When calculating value for one row, can access values in other rows for comparisons or calculations
    - Written in two parts: **aggregation** function and **over** statement
      * **Aggregation**
        + **row\_number()** - assigns an integer value, starting with 1 and increasing (ties are broken randomly)
        + **rank()** - assigns an integer value, starting with 1, and increasing (ties get same value, and next value(s) skipped)
        + **dense\_rank()** - assigns an integer value, starting with 1, and increasing (ties get same value, and next value(s) NOT skipped)
        + Most aggregate functions will work within a window function, including sum, count, etc., to get a running value (see **Aggregate functions**)
        + **lag** and **lead** can be used to get value in previous or next row
      * **Over statement**
        + Determines how to perform the calculation
        + Must be surrounded by parentheses (starting after the “over”)
        + **Partition by (optional)**

Splits rows into groups, and performs the aggregation on each group separately

Can partition by multiple fields

Can partition by fields that are not in the SELECT clause, though this is usually not recommended

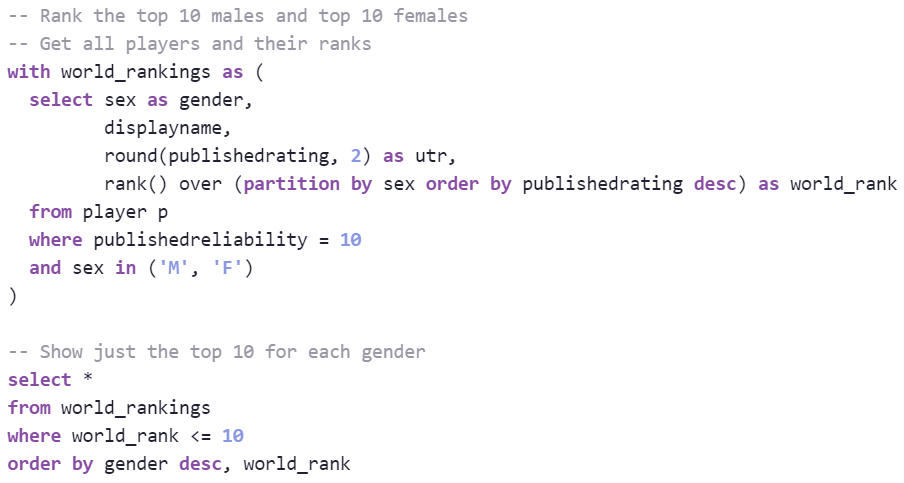
* + - * + **Order by**

Used to declare how to sort the values, before performing the calculation

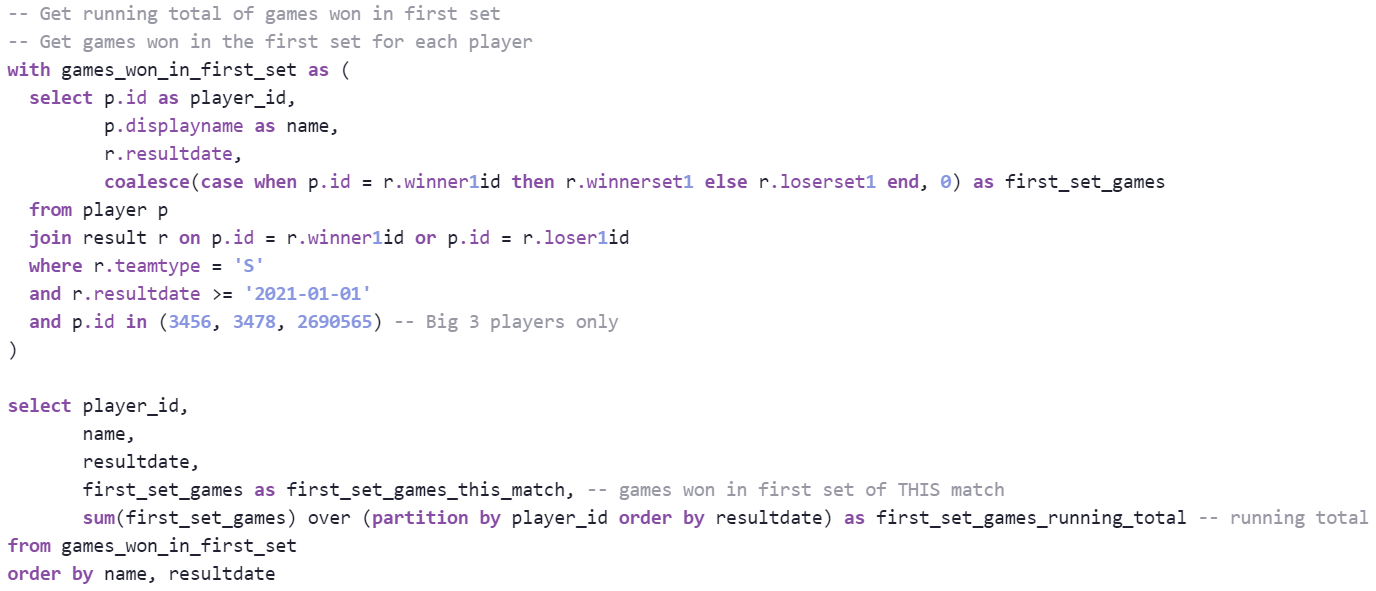
Can sort by multiple values, and can use desc, similar to standard ORDER BY clause (see **ORDER BY clause**)

Can order by fields that are not in the SELECT clause, though this is usually not recommended

* + - **Notation:**
      * Try to write all on one line, if possible
      * If necessary, put the “over” and everything after it on a new line, indented two spaces (or one tab)
    - Example 1:



* + - Example 2:



## Bracket notation (SiSense only)

* + - **NOTE:** If running a query outside of SiSense, DO NOT use bracket notation
      * Click on “Query” (between the workspace and the results) to see a version of the query that translates the bracket notation to SQL code. Use this code outside of SiSense (copy/paste it)
    - **Dates**
      * **[<datefield>:<datepart>]** returns the <datefield> formatted to highlight the desired <datepart>
        + Example: [getdate():date] returns the current date, without its timestamp
        + Some accepted <datepart> arguments:

date

month

year

week

day\_of\_week

hour

* + - * **[<datefield>:<timezone>]** converts the timezone of <datefield> to <timezone>
        + Example: [getdate():utc] returns the current date, converted to UTC (7 hours ahead of PST)
        + **NOTE:** if using both datetype conversion and timezone conversion, **ALWAYS** do the timezone conversion first

[getdate():pst:date]

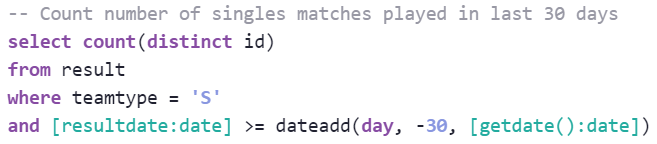
~~[getdate():date:pst]~~

* + - **Views**
      * A previously-defined query (saved elsewhere), that can be used in any query
      * [<view\_name> as <alias>]
      * Alias appears inside the brackets, and **MUST** use “as” (not optional in this case)

## Datediff and Dateadd

* + - **Datediff**
      * datediff(<datepart>, <date1>, <date2>)
      * **Returns a number:** the number of <dateparts> between <date1> and <date2>
      * Can be negative, if <date1> is later than <date2>
      * Some accepted <datepart> arguments:
        + day
        + month
        + year
        + week
        + hour
      * **NOTE:** datediff will truncate finer parts of the date (ie, day will remove timestamps, month will remove days, etc.)
      * Example 1: datediff(day, ‘2021-07-01’, ‘2021-08-01’) returns 31
      * Example 2: datediff(day, ‘2021-08-01’, ‘2021-07-01’) returns -31

### Dateadd

* + - * dateadd(<datepart>, <number of units to add>, <datefield>)
      * **Returns a date:** <datefield> with additional time units added to it
      * <number of units to add> can be negative (to look into the past)
      * Accepted <datepart> arguments are the same as for datediff
      * **NOTE:** timestamps can cause unexpected results, so it is recommended that they be removed using [<datefield>:date], if they are not needed
      * Example:
      * 

### When to use a datediff or dateadd - which one is better?

* + - * Often interchangeable
      * **Use datediff if:**
        + Function is being used in SELECT statement (desired outcome is a number, difference in dates)
        + Desired output is a number (time between two date fields)
      * **Use dateadd if:**
        + Function is being used in WHERE clause (desired outcome is a min/max date condition)
        + Desired output is a date that is some set amount of time before or after another date
        + Want to consider timestamps, or finer differences between date values

## GETDATE()

* + - Returns the current date and time, as a timestamp
    - Use [getdate():date] to get just the date

## Age calculation

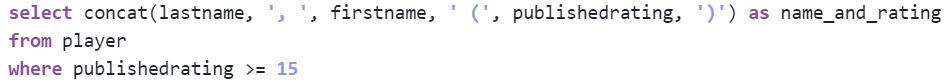
* + - datediff(day, [<birthdate>:date], [getdate():date]) / 365.25
    - Surround with **floor()** if whole numbers are preferred

## Completed matches only (from result table)

* + - In the **result** table, the “outcome” column has many different values
    - If outcome is “Completed,” null, or an empty string (different from null), then the match was completed
    - Statement to get all of these matches (merges null and empty string into “Completed”):
      * coalesce(nullif(outcome, ''), 'Completed')
  + Round all players’ ratings to nearest 0.5 (or similar)
    - round(publishedrating / 0.50, 0) \* 0.50 as rating\_rounded

## Concatenation

* + - Used to join two or more strings
    - Numbers and numeric fields will be converted to characters
    - **concat**
      * concat(<str1>, <str2>, <str3>, …)
      * Combines all inputs into one string, without adding spaces or anything in between the individual strings
      * Example:



* + - * + Sample value from output: “Murray, Andy (15.0866)”
    - **concat\_ws**
      * concat\_ws(<separator>, <str1>, <str2>, <str3>, …)
      * Combines all strings into one string, adding <separator> between every string
      * Example:



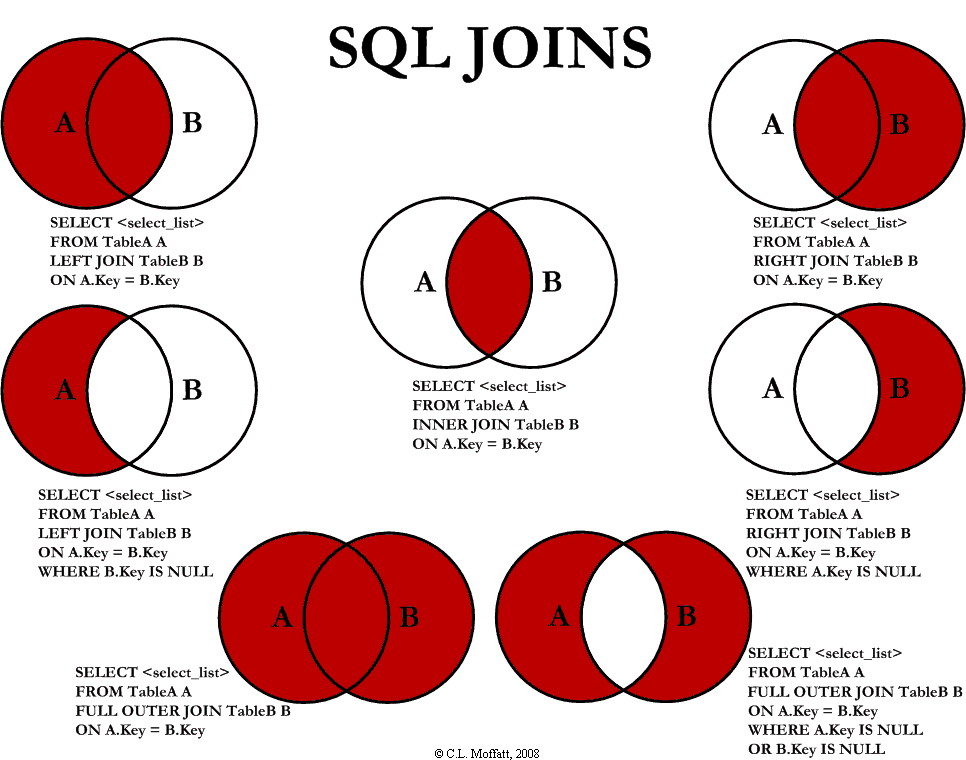
* + - * + Sample value from output: “Stanford, CA, United States”

## Lower (Upper)

* + - lower(<str>)
    - Converts string to all lowercase (uppercase) letters
    - Example: lower(‘Test String’) returns “test string”

## Left (Right)

* + - left(<str>, <n>)
    - Takes the first (last) <n> characters of a string
    - Example: SELECT left(‘test string’, 7) returns “test st”



# Bitbucket Etiquette

* Every project in bitbucket must have a virtual environment created along with a requirements.txt file listing out those packages installed in the virtual environment
* Creating a virtual environment for your project
  + Cd into your project folder
  + Py -m venv [name of your environment]
* Activating a virtual environment
  + Cd into your project folder
  + .\[name of your environment]\Scripts\activate
* Deactivating a virtual environment
  + Deactivate
* Installing packages into a virtual environment (must be activated)
  + Py -m pip install [package name]
* Creating a requirements.txt file
  + In your project folder, in your activated virtual environment: pip freeze > requirements.txt
* Installing packages from a requirements.txt file into a virtual environment
  + In your project folder with your virtual environment activated: pip install -r requirements.txt
* Cloning the bitbucket repo
  + Cd into project folder
  + Git clone [bitbucket repo URL]
* How to stage changes
  + Cd into project folder
  + Git add .
* How to commit
  + Cd into project folder
  + Git commit -m “your commit message”
* When we commit
* How we merge
* How to push committed changes
  + Cd into project folder
  + Git push

# Folder Structure

* Ad-hoc Analyses
  + Code folder
    - Analysis executable py module imports from
      * Functions py module
  + Data folder
    - csv
    - Database pulls if writing to a csv
  + Report folder
    - Written up concluding analysis whether its a
      * Notebook
      * Powerpoint
      * Doc
      * Etc
* Bitbucket Projects
  + Project Folder
    - Example: FlaskWebApps
    - Code Folder
      * Example: InsightsWebApp
      * All python modules
      * Requirements.txt file

# File Format

* Library Imports
* Data Reads
* Code

# Imports

* Numpy → np
* Pandas → pd
* Seaborn → sns
* Matplotlib.pyplot → plt
  + %matplotlib inline
* Scikitlearn → sklearn
* Networkx → nx
* Pyodbc → Pyodbc

# Jupyter Notebook Magic Functions

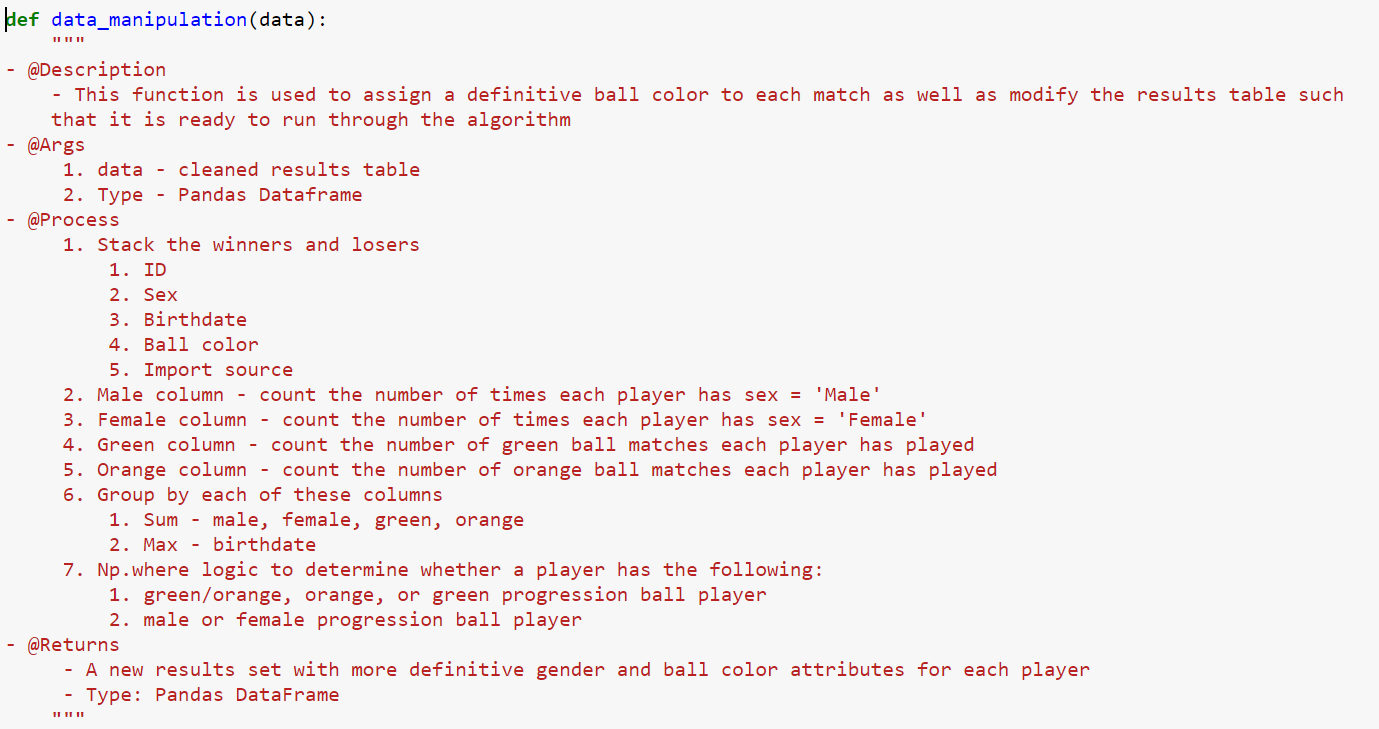
* %%time → will print out the time it took to run the cell at the end of the run
* %matplotlib inline → Graphical design magic function that is written after the matplotlib.pyplot import

# Jupyter Notebook Shortcuts

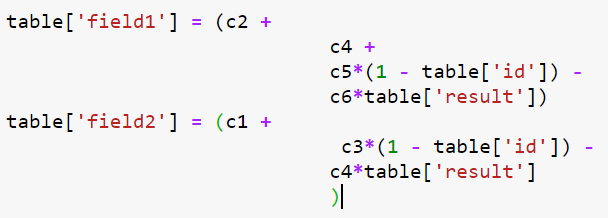
* Multi-cursor → ctrl + Left Click
* Auto-bracket → highlight text to be bracketed + [ or highlight text + { or highlight text + (
* To go from edit mode to command mode → esc
* To go from command mode to edit mode → Enter
* Add a cell above
  + If in command mode → a
  + If in edit mode → esc + a
* Add a cell below
  + If in command mode → b
  + If in edit mode → esc + b
* Delete the current cell
  + If in command mode → d + d
  + If in edit mode → esc + d + d
* Splitting a cell at the cursor
  + If in command mode → Enter + cursor + ctrl + shift + -
  + If in edit mode → cursor + ctrl + shift + -
* Merge cells (merges with the cell below the selected cell)
  + If in command mode → shift + m
  + If in edit mode → esc + shift + m
* Selecting multiple cells downward
  + If in command mode → shift + j
  + If in edit mode → esc + shift + j
  + Can apply delete/merge/copy/cut/paste/run to the selected cells
* Selecting multiple cells upward
  + If in command mode → shift + k
  + If in edit mode → esc + shift + k
  + Can apply delete/merge/copy/cut/paste/run to the selected cells
* Get line numbers
  + If in command mode → l
  + If in edit mode → esc + l
* Change code cell to markdown cell
  + If in command mode → m
  + If in edit mode → esc + m
* Change markdown cell to code cell
  + If in command mode → y
  + If in edit mode → esc + y

# Functions

## Documentation

* + @Description - what does this function do at a high level, 1 - 2 sentences
  + @Args - what are the inputs to this function and what do they represent, 1 descriptive statement per argument
  + @Process - what does the function do at a deeper level, should have an algorithmic tone, longest section
  + @Returns - describe what the function returns and its data type
    - Example taken from the Progression Ball Algo
    - 

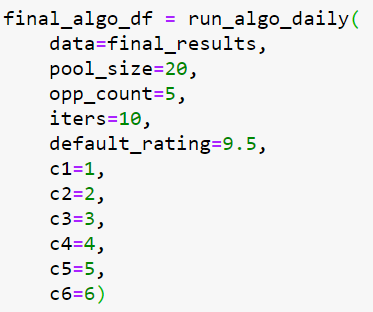
## Line style

* + As much as possible, try to not have lines in the function bleed off the edge of the page for sake of readability
    - Exceptions include:
      * Multiple nested np.where statements
  + Equations should be wrapped in parentheses and broken up if needed to ensure that it does not bleed off the edge of the page. They should be broken up at their operator i.e. +, -, \* , \. Powers (\*\*) and modulus (%) however should stay on the same line.
    - Example, assuming c1, c2, c3, c4, c5, c6 are integers:
      * 

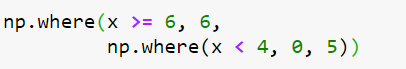
## Length limits

* + There isn’t a specific limit but if be cognizant of making functions readable (less than 100 lines in the most extreme cases)

## Calling a function

* + Function calls should consist of all of the arguments written out and their inputted values
    - Only time which this does not apply is when your argument names are identical to the names of the inputs
    - Example from the progression ball algo
    - 

## Library-specific functions

* + Generally
    - Library-specific functions can be broken out line by line at their arguments.
    - If edge bleeding is occurring in a library-specific call you should break it out at the furthest argument that doesn’t bleed off the page
  + Numpy
    - Numpy.where or np.where
      * For nested np.where functions, each subsequent nested np.where statement should be indented one tab to the right of the last np.where statement and should be lined up with the condition of the previous np.where statement
        + Example
        + 
      * Individual conditions inside of the np.where statements should be on one line regardless of edge bleeding
      * In the case of multiple conditions, they should be broken up if not doing so will cause edge bleeding. The conditions should be broken up at their operator i.e. “&” or “|”