REDCap Programming & Logic

About
This guide is designed to help you make sense of branching logic, calculated fields, and the core logic patterns used throughout REDCap. Whether you are building a new project or troubleshooting an existing one, this page provides a practical framework for understanding how logic works and how to use it effectively. Refer to the Appendix A and B at the end of this guide for a deep dive of REDCap branching logic and calculated values.

Logic is one of the most powerful features in REDCap. It controls when fields appear, how values are calculated, how reports are filtered, when notifications are triggered, and how workflows behave across classic, longitudinal, and repeating projects.

In REDCap, logic always evaluates to true or false. The key is to build logic that is clear, testable, and appropriate for the field type and project structure.

The same general logic language is used across many REDCap features, but the behavior may differ depending on where the logic is used. Branching logic, calculated fields, ASIs, Alerts, Survey Queue, reports, and dashboards all use related syntax, but not always in exactly the same way.

Where Logic Is Used in REDCap
Logic is used across multiple parts of REDCap, not just branching logic and calculations. It can be used in:

- Branching logic
- Calculated fields
- Automated Survey Invitations
- Alerts & Notifications
- Survey Queue
- Data Quality rules
- Report filters
- Project dashboards

Simple rule: Before writing logic, make sure you know where the logic will be used, because that affects how the logic behaves and what features are compatible.

Logic Building Blocks

Most REDCap logic uses the same core syntax elements:

- Variables: [variable_name]
- Operators: =, <>, >, <, >=, <=
- Boolean logic: and, or
- Parentheses: used to group statements
- Strings vs Numbers: Use quotes when comparing strings or coded values that should be treated as strings. Leave quotes off for true numeric comparisons. Use the "" pattern with no spaces when checking for blank or null values. A space inside the quotes changes the meaning.

```
[gender] = '1'
[weight] > 100
[last_name] <> ""
```

- Combining Statements: Use parentheses to control grouping and make complex logic easier to read.

```
([variable1] = '1' or [variable2] = '1') and [variable3] = '2'
```

- Checkbox Logic: use different syntax because each answer choice behaves like its own yes/no field.

```
[race(2)] = '1'   checked
[race(4)] = '0'   unchecked
```

Design Tip: For more complex branching logic, it can help to build the basic statement with the drag-and-drop builder and then refine it manually using advanced syntax.

Branching Logic

Branching logic hides or shows a field based on conditions. In REDCap, the logic is placed on the destination field — the field you want to show or hide — not on the source field. Branching Logic shows a field only if a prior answer meets certain criteria, hides irrelevant questions, and creates cleaner, more dynamic instruments. Branching Logic cannot skip or hide an entire instrument, cannot be applied at the section level, and it must be applied field by field

Section headers are hidden only when all fields in that section are hidden. Entire instruments remain visible even if every field on the form is hidden by branching logic.

Calculated Fields

Calculated fields perform real-time numeric calculations on forms and surveys. They use variable names in brackets and mathematical functions similar to Excel.

Core Rules:

- Calculated fields can return numbers only
- They cannot directly return text or dates
- Use mathematical order of operations carefully
- Examples

```
[bpi_q1] + [bpi_q2]
sum([q1],[q2],[q3],[q4])
if([weight] > 100, 44, 11)
```

Use sum() vs + Carefully. Sum() ignores blank values, whereas + requires all fields to have values

Which function to use depends on how the instrument should be scored. A missing value may need to be ignored or may need to invalidate the total.

- Checkbox calculations are more complex because each answer choice is scored as checked or unchecked rather than by its visible choice code.

```
if([exercise(1)] = 1, 1, 0)
```

Important: If possible, avoid using complex calculations on checkbox fields unless the scoring structure is very clear.

- Special Functions: REDCap supports many functions in calculated fields, including:

```
round(), roundup(), rounddown()
sqrt(), abs()
min(), max(), mean(), median(), sum(), stdev()
```

- Working with Dates and Time: Dates are commonly used in both branching logic and calculations. The most common function is datediff().

```
Datediff([dob],[visit_date],"y","mdy")
datediff("today",[screen_date],"M")
```

Common Units include "y" years, "M" months, "d" days, "h" hours, "m" minutes, and "s" seconds.

Important:

- Both dates must match the format you specify in the function. Test carefully whenever dates are part of logic or calculations.
- While "today" can be used, it should be used carefully in calculated fields because the result changes whenever the form is re-opened and saved. A fixed project date field is often safer.

Longitudinal, Repeating, and Smart Variables

- **Longitudinal Event References.** In longitudinal projects, fields from other events must include the unique event name.

  [baseline_arm_1][weight]

- **Repeating Instruments.** You can reference a specific repeating instance by appending the instance number to the variable.

  [variable1][4]

- **Smart Variables.** Smart Variables make logic more flexible by referencing contextual concepts such as first event, previous event, or first instance.

  [first-event-name][variable]
  [variable][first-instance]

  Design Tip: Event names and smart variables are powerful, but they add complexity quickly. Use them intentionally and test them thoroughly.

Logic and Action Tags

Logic can interact with action tags, but action tags do not always behave like branching logic or calculations.

Important Interactions:

@HIDDEN overrides branching logic and hides the field regardless of logic result
@DEFAULT works only if the field is shown when the form first loads
@IF allows conditional action-tag behavior
@CALCDATE and @CALCTEXT allow text/date outputs that standard calculated fields cannot return

Action tags generally work when the form first loads and may not update in real time as data are entered.

Testing and Troubleshooting

Logic should always be tested with real test records, not just by previewing a form. Follow these tips:

- Preview mode does not fully test branching logic or calculations
- Use realistic test records and enter data directly
- Use reports or Data Quality rules to validate complex logic
- Diagram complicated logic flows before building them

Best Practice: For complex logic, draw out the logic in a flow chart before building it. This makes troubleshooting much easier.

Common Troubleshooting Tips

- Wrong variable name or wrong event reference
- Using a checkbox field as if it were a radio field
- Blank value behavior not handled properly
- Calculated field hidden by branching logic but still returning data

Important: If a calculated field is hidden by branching logic but still evaluates to a value, REDCap may show an error or warning. The fix is often to build the branching condition into the calculation itself using an `if()` statement that returns `""` when hidden.

Updating Calculation Values

When a calculated field is added or changed, the new calculation may appear on screen, but the values are not automatically saved to the server. This is why updated values may not appear in reports or exports right away.

To update saved values:

*Go to **Data Quality** ➔ Run **Rule H** ➔ Review the listed fields and values ➔ Use **Fix calcs now** if appropriate*

Rule H updates all listed calculation values unless you explicitly exclude specific items first. Review carefully before fixing values in bulk. Run Rule H soon after adding or changing calculated fields so the list stays short and easier to review.

Key Best Practices

- Use clear variable names and consistent coding
- Keep logic as simple as possible
- Use parentheses intentionally in multi-part statements
- Know whether your field is a regular multiple-choice field or a checkbox field
- Test with realistic records and actual saved data
- Use event references and smart variables carefully in longitudinal or repeating projects
- Do not overuse calculated fields, especially on large forms
- Run Rule H after adding or changing calculations

- Final Takeaway: Good logic should be readable, predictable, and easy to test. If the logic is difficult to explain, it is usually worth simplifying before putting it into production.

Appendix A
REDCap Branching Logic Deep Dive

[REDCap Branching Logic: A Practical Guide to Dynamic Forms | Clinical Research Center]

Branching logic allows you to show or hide fields based on participant responses or existing data. It is one of the most powerful tools in REDCap for creating clean, efficient, and dynamic instruments.

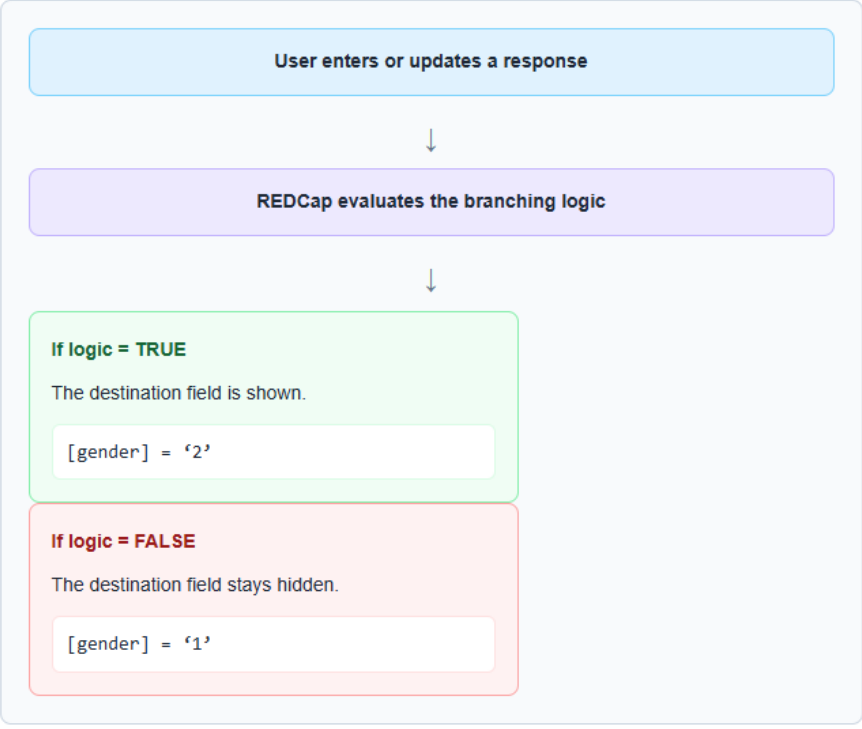
Branching logic determines **whether a field is displayed**. If the logic evaluates to TRUE, the field is shown. If FALSE, the field is hidden. Also, branching logic is applied to the **field you want to show or hide** (the destination field), not the field that contains the triggering value.

Branching Logic:

- Shows fields only when relevant
- Hides unnecessary questions
- Improves data quality
- Creates a guided user experience

How Branching Logic Flows

At its core, branching logic follows a simple decision path: REDCap checks a condition, then either shows or hides the destination field.



The logic belongs on the field you want to show or hide. REDCap checks that field's rule every time the page loads or a triggering value changes. As an example, if you only want to show a pregnancy question when the response to sex is Female, the pregnancy field would contain the branching logic, such as [sex] = '0' if Female is coded as 0.

How to Set It Up

Go to **Online Designer** ➔ Edit a field ➔ Enter logic in the **Branching Logic** box ➔ Or use the **logic builder**

Tip: Use the builder first, then refine manually.

Basic format:

```
[variable] = 'value'
```

Examples:

```
[gender] = '1'
[age] > 18
[consent_complete] = 2
[field_name] <> ""
```

⚠ **Important:** Use "" (no space) to check for blank values.

Simple vs. Complex Branching Logic
Branching logic can range from very simple conditions to more complex, multi-part rules. Understanding how logic builds step-by-step makes it much easier to write and troubleshoot.

Simple Logic

One condition controls whether the field appears.

```
[gender] = '2'
```

→ Show field only if gender = Female

Use case: Basic eligibility or demographic filtering

AND Logic

All conditions must be TRUE.

```
[age] > 18 and [consent] = '1'
```

→ Show field only if BOTH conditions are met

Use case: Eligibility + consent requirements

OR Logic

Only one condition must be TRUE.

```
[symptom1] = '1' or [symptom2] = '1'
```

→ Show field if ANY condition is met

Use case: Trigger follow-up questions

Complex Logic (Grouped Conditions)

Use parentheses to control how logic is evaluated.

```
([q1] = '1' or [q2] = '1') and [q3] = '2'
```

→ Show field if **(q1 OR q2)** AND q3 is true

Key Tip: Without parentheses, REDCap may evaluate logic in an unexpected order.

Checkbox Logic Example

Checkboxes require special syntax for each option:

```
[symptoms(1)] = '1' or [symptoms(2)] = '1'
```

→ Show field if either checkbox option is selected

⚠️ **Important:** Each checkbox choice is its own variable—do not treat it like a dropdown or radio field.

✅ **Best Practice:** Start simple. Build and test one condition at a time, then layer additional logic.

Checkbox Logic. Checkboxes use special syntax:

[field(1)] = '1' → checked

[field(1)] = '0' → unchecked

Each checkbox option is its own variable.

Building Complex Logic

([q1] = '1' or [q2] = '1') and [q3] = '2'

Always use parentheses for clarity.

Longitudinal Considerations

[baseline_arm_1][weight]

- Reference events explicitly
- Test across events

Branching Logic Limitations:

- Cannot hide entire instruments
- Cannot skip surveys
- Works only at field level
- NOT event-relative by default

Common Mistakes:

- Applying logic to wrong field
- Treating checkbox like radio
- Forgetting quotes
- Not testing with real data
- **Common Mistake:** Assuming preview mode fully tests logic.

Common Real-World Examples

Branching logic is most useful when it mirrors how people actually move through a form or survey. Below are some common ways it is used in real REDCap projects.

Eligibility Screening

Only show follow-up eligibility questions if the participant meets initial screening criteria.

```
[age] >= 18 and [consent_interest] = '1'
```

Use case: Recruitment and screening forms

Sex-Specific Questions

Only display fields when they are relevant to the participant.

```
[sex] = '2'
```

Use case: Pregnancy history, prostate-related questions, etc.

"Other, Specify" Fields

Show a text box only when a participant selects "Other."

```
[race] = '88'
```

Use case: Dropdown or radio fields with an "Other" choice

Medication Lists

Show the next medication field only when the previous one has been completed.

```
[med1] <> ""
```

Use case: Repeating-like entry without repeating instruments

Symptom Follow-Up

Show severity or details only if a symptom is present.

```
[symptom_present] = '1'
```

Use case: Adverse events, symptom screens, clinical intake

Checkbox-Based Follow-Up

Show additional questions if one or more checkbox options are selected.

```
[symptoms(1)] = '1' or [symptoms(2)] = '1'
```

Use case: Multi-select symptom or exposure lists

Branching logic works best when it follows the natural decision process of the user. If the logic feels hard to explain in plain language, it is often worth simplifying.

Best Practices:

- Keep logic simple
- Use consistent coding
- Test thoroughly
- Use parentheses
- Write down the plain-English rule first, then convert it into REDCap logic. For example: "Show this field only if the participant is over 18 and consented."

Appendix B
REDCap Calculated Fields Deep Dive

Calculated fields allow REDCap to automatically compute values in real time using data from other fields. They are commonly used for scoring instruments, calculating age or time intervals, and generating derived variables for analysis. Calculated fields always return a numeric value based on a formula using other fields.

Tip: Calculated values are not automatically saved for existing records when you change a formula. You must run Data Quality Rule H to update stored values.

Calculated Fields:

- Automatically compute scores
- Calculate age or time differences
- Generate derived variables
- Reduce manual calculation errors

Use calculated fields for simple, real-time calculations—not complex data processing.

Calculation Syntax

Basic format:

```
[field1] + [field2]
```

Examples:

```
[bp_sys] + [bp_dia]
[bmi_weight] / ([height]^2)
if([age] > 65, 1, 0)
```

Tip: Use square brackets for all variables.

Common Functions:

sum(), mean(), min(), max(), round(), if()

```
sum([q1],[q2],[q3])
round([value],1)
if([score] > 10, 1, 0)
```

sum() vs. "+"	
+	Fails if any field is blank
sum()	Ignores blanks

Important: Choose based on scoring rules.

Working with Dates

```
datediff([dob],[visit_date],"y")
datediff("today",[visit_date],"d")
```

y = years, d = days, M = months

Important: “today” updates each time the record is opened.

Checkbox Calculations

```
if([exercise(1)] = 1, 1, 0)
```

Checkbox values are binary (0/1), not choice codes.

Real-World Examples:

Total Score

```
sum([q1],[q2],[q3],[q4])
```

Age

```
datediff([dob],"today","y")
```

Eligibility Flag

```
if([age] > 18,1,0)
```

Rule H

Use Rule H to update values: Go to Data Quality ➔ Run Rule H ➔ Click “Fix calcs now”

Common Mistakes:

- Using text instead of numeric output
- Not running Rule H
- Using checkbox incorrectly
- Overcomplicating formulas

Best Practices

- Keep formulas simple
- Use sum() when appropriate
- Test thoroughly
- Avoid heavy use in large projects