# **Analysis of Scratch 3.0 Block Architecture and Functionality**

## **Executive Summary**

Scratch 3.0 revolutionizes introductory programming through its intuitive visual, block-based interface, making complex computational concepts accessible to a broad audience. The platform's design centers on a drag-and-drop paradigm, where programming blocks interlock like physical components to construct scripts.1 This system organizes its extensive library of blocks into nine primary categories, distinguished by a color-coding scheme. Crucially, these blocks are further classified by six unique shapes, each signaling a distinct function and dictating how they connect within a script.1 This visual grammar is fundamental to understanding the construction and execution flow of Scratch programs.

The six fundamental block shapes in Scratch 3.0 are Hat, Stack, Boolean, Reporter, C, and Cap blocks.1 Each shape plays a specific role in controlling program flow, managing data, or initiating events. Hat blocks serve as script initiators, responding to specific triggers. Stack blocks execute sequential commands, forming the core actions within a script. Boolean blocks are hexagonal, representing conditions that evaluate to either true or false. Reporter blocks, with their rounded edges, output values (numbers or strings) for use in other blocks. C blocks, shaped like the letter "C," control loops and conditional execution. Finally, Cap blocks signify the termination point of a script, preventing further blocks from being added below them.1 This systematic categorization provides a clear and immediate understanding of each block's behavior and its role within the broader programming logic.

## **Introduction to Scratch 3.0 Block Architecture**

Scratch 3.0 employs a meticulously designed block architecture that underpins its effectiveness as a visual programming environment. The system's primary organizational principle involves nine distinct color-coded categories, each representing a specific functional domain. For instance, Motion blocks are dark blue, Looks blocks are purple, Events blocks are light yellow, Control blocks are amber, Operators blocks are light green, Data blocks are orange, and My Blocks are typically red or pink.1 This visual differentiation allows users to quickly identify and locate blocks pertinent to their desired functionality.

Beyond color, the unique physical shapes of the blocks are paramount in conveying their purpose and how they integrate into a script.1 This visual syntax is a cornerstone of Scratch's accessibility and pedagogical efficacy. The shape of a block directly communicates its role: Hat blocks initiate scripts, Stack blocks perform sequential actions, C blocks manage control flow, Boolean blocks provide conditional logic, Reporter blocks supply data, and Cap blocks conclude sequences.1 This visual language serves as a powerful abstraction of fundamental programming concepts. The interlocking nature of these shapes further ensures syntactical correctness, guiding users to construct valid code structures and significantly reducing common programming errors for beginners.1 This "snap-together" design is a key feature that simplifies the learning process.

The consistent use of color-coding and distinct block shapes in Scratch 3.0 functions as a highly effective visual grammar. This design choice implicitly teaches fundamental programming concepts—such as sequence, iteration, selection, event handling, and data representation—without requiring explicit memorization of syntax. For example, the hexagonal shape of a Boolean block visually indicates that it fits into a corresponding hexagonal slot within an if or repeat until block, thereby reinforcing logical connections through physical compatibility. This approach significantly lowers the barrier to entry for programming, allowing users to concentrate on algorithmic logic and problem-solving rather than syntactic rules. The result is an environment that fosters computational thinking skills and promotes early engagement and success in coding.

## **Hat Blocks: Script Entry Points**

Hat blocks are distinct in Scratch 3.0, characterized by their rounded top and a prominent bump at the bottom.1 This unique shape immediately signifies their role as the starting point for any script.1 Functioning as event listeners, these blocks initiate the execution of a sequence of interconnected blocks when a specific condition is met, such as a user interaction, a system event, or a change in the program state.1 Without Hat blocks, projects would be unable to begin automatically, underscoring their fundamental importance in defining the entry points for program execution.1

The prevalence of Hat blocks in Scratch 3.0 underscores the platform's foundation in an event-driven programming paradigm. These blocks serve as the sole entry points for scripts, responding to diverse occurrences like mouse clicks, key presses, broadcast messages, backdrop changes, or when a value exceeds a predefined threshold.1 This design means that scripts do not merely execute linearly from start to finish; rather, they react dynamically to occurrences within the program or from user interaction. This approach implicitly introduces users to event handling, a core concept in modern software development, particularly in graphical user interfaces and web applications. It also introduces the concept of concurrency, as multiple scripts can be triggered and run simultaneously in response to different events. The

broadcast and when I receive blocks, also part of the Events category, further extend this model by demonstrating message-passing and coordination between independent program components.1 By making event-driven logic intuitive and visually clear, Scratch effectively prepares users for more complex asynchronous programming paradigms they may encounter in advanced languages, shifting their thinking from purely sequential execution to a reactive model crucial for interactive applications and games.

A powerful application of Hat blocks is found in "My Blocks," or Custom Blocks, which are defined using a special define Hat block.1 This feature is instrumental in promoting advanced programming principles such as abstraction and modularity. Users can encapsulate complex sequences of code into a single, custom-named block, which can also accept inputs.1 This directly introduces the concept of abstraction by hiding intricate details behind a simple, reusable interface. Modularity is achieved by enabling users to break down large problems into smaller, more manageable sub-problems, each represented by a custom block. The ability to define inputs for these custom blocks teaches parameterization, a fundamental aspect of function design in all programming languages. The clear distinction between the

define block (serving as the blueprint) and its callable instance (representing its usage) reinforces the concept of function definition versus function call. This capability bridges the gap between simple block-based scripting and more structured, organized programming practices, empowering users to build more complex and maintainable projects and fostering good software engineering habits early in their programming journey. It also subtly introduces the idea of local scope for input variables within the custom block's definition.1

### **Table 1: Hat Block Details**

| Block Name | Block Type (Category) | Block Shape | Description | Inputs | Example (Standalone) | Example (With Other Blocks) |
| --- | --- | --- | --- | --- | --- | --- |
| when green flag clicked | Events | Hat Block | Triggers the script when the green flag, serving as the project's "start" button, is clicked. | None | when green flag clicked | when green flag clicked\n go to x: (0) y: (0)\n forever\n move (10) steps\n turn right (15) degrees\n end (Centers sprite, then moves and turns continuously) 2 |
| when () key pressed | Events | Hat Block | Initiates the script when a specified keyboard key is pressed. | key (dropdown: space, up arrow, a, etc.) | when [space v] key pressed | when [right arrow v] key pressed\n point in direction (90)\n move (10) steps (Moves sprite right when right arrow key is pressed) 3 |
| when this sprite clicked | Events | Hat Block | Starts the script when the sprite itself is clicked. | None | when this sprite clicked | when this sprite clicked\n say [Ouch!] for (1) seconds\n change [score v] by (-1) (Sprite says "Ouch!" and score decreases on click) |
| when backdrop switches to () | Events | Hat Block | Triggers the script when the stage backdrop changes to a specified backdrop. | backdrop name (dropdown) | when backdrop switches to [game over v] | when backdrop switches to [game over v]\n stop [all v] (Stops all scripts when backdrop changes to "game over") |
| when () > () | Events | Hat Block | Starts the script when a certain value (e.g., loudness from a microphone, or the timer) exceeds a defined threshold. | value type (dropdown: loudness, timer), threshold (number) | when [loudness v] > (70) | when [loudness v] > (70)\n start sound [scream v] (Plays scream sound if loudness exceeds 70) |
| when I receive () | Events | Hat Block | Initiates the script upon the reception of a specific broadcast message, facilitating indirect communication between sprites or the stage. | message name (dropdown) | when I receive [start game v] | when I receive [start game v]\n show\n go to x: (0) y: (0) (Shows sprite and moves it to center upon receiving "start game" message) 1 |
| define [my custom block] | My Blocks | Hat Block | Serves as the definition header for a custom block's script, allowing users to define reusable sequences of code, promoting modularity and abstraction. | Placeholders for inputs (e.g., height) | define jump (height) | define jump (height)\n change y by (height)\n wait (0.1) seconds\n change y by (0 - (height)) (Defines a custom "jump" block that moves sprite up and down) 1 |

### **JSON for Hat Blocks**

JSON

{  
 "block\_category": "Hat Blocks",  
 "description": "Hat blocks are characterized by a rounded top and a bump at the bottom. They initiate scripts, meaning they are the starting point for a sequence of interconnected blocks.",  
 "blocks":,  
 "example\_standalone": "when green flag clicked",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "when () key pressed",  
 "block\_type": "Events",  
 "block\_shape": "Hat Block",  
 "functionality": "This Hat block initiates the script when a specified keyboard key is pressed.",  
 "inputs": [  
 {"name": "key", "type": "dropdown", "options": ["space", "up arrow", "a", "..."]}  
 ],  
 "example\_standalone": "when [space v] key pressed",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when [right arrow v] key pressed\n point in direction (90)\n move (10) steps",  
 "explanation": "This script moves the sprite right when the right arrow key is pressed. (Implied from [3])"  
 }  
 ]  
 },  
 {  
 "block\_name": "when this sprite clicked",  
 "block\_type": "Events",  
 "block\_shape": "Hat Block",  
 "functionality": "This Hat block starts the script when the sprite itself is clicked.",  
 "inputs":,  
 "example\_standalone": "when this sprite clicked",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when this sprite clicked\n say [Ouch!] for (1) seconds\n change [score v] by (-1)",  
 "explanation": "This script makes the sprite say 'Ouch!' and decreases the score by 1 when the sprite is clicked."  
 }  
 ]  
 },  
 {  
 "block\_name": "when backdrop switches to ()",  
 "block\_type": "Events",  
 "block\_shape": "Hat Block",  
 "functionality": "This Hat block triggers the script when the stage backdrop changes to a specified backdrop.",  
 "inputs": [  
 {"name": "backdrop name", "type": "dropdown"}  
 ],  
 "example\_standalone": "when backdrop switches to [game over v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when backdrop switches to [game over v]\n stop [all v]",  
 "explanation": "This script stops all running processes when the backdrop changes to 'game over'."  
 }  
 ]  
 },  
 {  
 "block\_name": "when () > ()",  
 "block\_type": "Events",  
 "block\_shape": "Hat Block",  
 "functionality": "This Hat block starts the script when a certain value (e.g., loudness from a microphone, or the timer) exceeds a defined threshold.",  
 "inputs": [  
 {"name": "value type", "type": "dropdown", "options": ["loudness", "timer"]},  
 {"name": "threshold", "type": "number"}  
 ],  
 "example\_standalone": "when [loudness v] > (70)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when [loudness v] > (70)\n start sound [scream v]",  
 "explanation": "This script starts a 'scream' sound when the microphone loudness exceeds 70."  
 }  
 ]  
 },  
 {  
 "block\_name": "when I receive ()",  
 "block\_type": "Events",  
 "block\_shape": "Hat Block",  
 "functionality": "This Hat block initiates the script upon the reception of a specific broadcast message. This mechanism facilitates indirect communication between sprites or the stage.",  
 "inputs": [  
 {"name": "message name", "type": "dropdown"}  
 ],  
 "example\_standalone": "when I receive [start game v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when I receive [start game v]\n show\n go to x: (0) y: (0)",  
 "explanation": "This script makes the sprite visible and moves it to the center of the stage when it receives the 'start game' broadcast. [1, 3]"  
 }  
 ]  
 },  
 {  
 "block\_name": "define [my custom block]",  
 "block\_type": "My Blocks",  
 "block\_shape": "Hat Block",  
 "functionality": "This Hat block serves as the definition header for a custom block's script, allowing users to define reusable sequences of code. It promotes modularity and abstraction.",  
 "inputs": [  
 {"name": "placeholder", "type": "text/number/boolean"}  
 ],  
 "example\_standalone": "define jump (height)",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 }  
 ]  
}

## **Stack Blocks: Sequential Commands**

Stack blocks represent the most common block shape in Scratch 3.0, characterized by a notch at the top and a bump at the bottom.1 This design allows them to connect seamlessly both above and below, forming linear sequences of commands that execute in a top-down, sequential order.1 They are the workhorses of Scratch, performing the main actions within any given script.

The distinction between synchronous and asynchronous operations is a fundamental concept introduced through Stack blocks like play sound () until done and start sound () 1, and similarly with

broadcast () and wait versus broadcast ().1 Some Stack blocks, such as

play sound until done or broadcast and wait, halt the execution of the current script until their designated action is fully completed. This behavior ensures sequential execution and predictability in program flow. Conversely, blocks like start sound or broadcast initiate their actions without pausing the current script, allowing other operations to proceed concurrently. This design choice implicitly teaches users about blocking versus non-blocking operations, a critical concept in concurrent programming. The selection between these two modes depends entirely on the desired program flow; for instance, one might choose a blocking sound to ensure an animation completes before the next action, or a non-blocking sound for continuous background music. Understanding this distinction is vital for designing complex interactive applications where responsiveness is key, laying a foundational concept for later learning about threads, promises, and asynchronous functions in more advanced programming environments.

Another important consideration for Stack blocks, particularly those manipulating data structures, involves performance implications. For example, the replace item () of () with () block in Scratch 3.0 has been observed to exhibit significantly slower performance compared to its Scratch 2.0 counterpart.1 This difference is attributed to the underlying JavaScript implementation, specifically the use of the

splice() method, which results in an O(N) runtime complexity for this operation. In contrast, direct index assignment (list.value[index - 1] = item) offers an O(1) operation, yielding a substantial performance improvement for projects involving large lists.1 This highlights that despite Scratch's visual simplicity, it operates on a sophisticated virtual machine with real-world computational limitations. It introduces the concept of algorithmic complexity (O(N) vs. O(1)) in a practical, albeit abstracted, manner, demonstrating that the "block-based" nature does not negate the importance of performance considerations. For advanced users or those developing performance-critical projects, such as complex games with extensive data structures, this knowledge is crucial for debugging and optimizing their code. It encourages thinking about efficient data structures and algorithms even within a visual programming context, and it illustrates that the Scratch Team actively optimizes the VM for common use cases.

### **Table 2: Stack Block Details**

| Block Name | Block Type (Category) | Block Shape | Description | Inputs | Example (Standalone) | Example (With Other Blocks) |
| --- | --- | --- | --- | --- | --- | --- |
| move () steps | Motion | Stack Block | Moves the sprite forward or backward in its current direction by a specified number of "steps." | steps (number) | move (10) steps | when green flag clicked\n forever\n move (10) steps\n if <touching edge?> then\n turn (180) degrees\n end\n end (Sprite moves continuously, turning when it touches the edge) 4 |
| say () for () seconds | Looks | Stack Block | Displays a speech bubble containing specified text for a set duration. | text (string), seconds (number) | say [Hello!] for (2) seconds | when green flag clicked\n say [Grr] for (3) seconds\n say [Have you seen my honey?] for (3) seconds (Sprite delivers sequential dialogue) 6 |
| switch costume to () | Looks | Stack Block | Alters the sprite's appearance to a designated costume. | costume name/number (dropdown/number) | switch costume to (costume1) | when I receive [explosion v]\n repeat (5)\n next costume\n end\n hide (Changes costume repeatedly to animate an explosion, then hides sprite) 3 |
| show | Looks | Stack Block | Makes the sprite visible on the stage. | None | show | when green flag clicked\n hide\nwhen I receive [start game v]\n show (Hides sprite at start, shows it when game begins) |
| hide | Looks | Stack Block | Makes the sprite invisible on the stage. | None | hide | when green flag clicked\n hide (Hides sprite immediately when the green flag is clicked) |
| play sound () until done | Sound | Stack Block | Plays a specified sound and pauses the script's execution until the sound has completed. | sound name (dropdown) | play sound [Meow v] until done | forever\n play sound [Music v] until done (Creates a continuous music loop) 8 |
| start sound () | Sound | Stack Block | Initiates playback of a specified sound without pausing the script, allowing other actions to proceed concurrently. | sound name (dropdown) | start sound [Pop v] | when this sprite clicked\n start sound [Pop v]\n change [score v] by (1) (Plays sound and changes score simultaneously on click) 1 |
| broadcast () | Events | Stack Block | Sends a broadcast message throughout the Scratch program, activating any when I receive () blocks that are set to listen for that message, enabling indirect communication. | message name (string/dropdown) | broadcast [start game v] | if <key [space v] pressed?> then\n broadcast [jump v]\nend (Broadcasts 'jump' message when space key is pressed) 9 |
| broadcast () and wait | Events | Stack Block | Sends a broadcast message and pauses the current script until all other scripts activated by that broadcast have completed their execution, ensuring sequential coordination. | message name (string/dropdown) | broadcast [initialize sprites v] and wait | broadcast [initialize sprites v] and wait\n say for (2) seconds (Initializes sprites then says "Game Started!" after all initialization scripts complete) 1 |
| wait () seconds | Control | Stack Block | Pauses the script for a specified duration. | seconds (number) | wait (1) seconds | say [Hello!] for (1) seconds\n wait (0.5) seconds\n say [Goodbye!] for (1) seconds (Creates a timed dialogue sequence) 1 |
| create clone of () | Control | Stack Block | Generates a copy, or clone, of a specified sprite. | sprite name (dropdown: myself, other sprites) | create clone of [myself v] | when green flag clicked\n hide\n forever\n create clone of [myself v]\n wait (1) seconds\n end (Continuously creates clones of the sprite every second) 1 |
| set [my variable v] to () | Data | Stack Block | Assigns a specific value (number, string, or boolean) to a variable. | variable name (dropdown), value (any type) | set [score v] to (0) | when green flag clicked\n set [score v] to (0)\n set [player name v] to [Guest] (Initializes score and player name variables at the start of the project) 1 |
| change [my variable v] by () | Data | Stack Block | Increases or decreases a variable's numerical value by a specified amount. | variable name (dropdown), value (number) | change [score v] by (1) | when this sprite clicked\n change [score v] by (1) (Increases score by 1 each time the sprite is clicked) 1 |
| add () to [my list v] | Data | Stack Block | Appends an item to the end of a list. | item (any type), list name (dropdown) | add [apple] to [shopping list v] | when green flag clicked\n add [apple] to [shopping list v]\n add [banana] to [shopping list v] (Adds "apple" and "banana" to a shopping list) 1 |
| delete () of [my list v] | Data | Stack Block | Removes an item from a list by its index or by selecting "all" items. | index/option (number/dropdown: all, last, random), list name (dropdown) | delete (1) of [my list v] | when green flag clicked\n delete (all) of [my list v] (Clears all items from a list when the green flag is clicked) 1 |
| insert () at () of [my list v] | Data | Stack Block | Inserts an item at a specific position within a list. | item (any type), index (number), list name (dropdown) | insert [orange] at (2) of [fruits v] | insert [orange] at (2) of [fruits v] (Inserts "orange" as the second item in the "fruits" list) 1 |
| replace item () of [my list v] with () | Data | Stack Block | Replaces an item at a specific position in a list with a new value. | index (number), list name (dropdown), new item (any type) | replace item (1) of [colors v] with [blue] | replace item (1) of [colors v] with [blue] (Changes the first item in the "colors" list to "blue") 1 |
| show variable [my variable v] | Data | Stack Block | Makes a variable's monitor visible on the stage. | variable name (dropdown) | show variable [score v] | when green flag clicked\n show variable [score v] (Displays the score variable on the stage at project start) 1 |
| hide variable [my variable v] | Data | Stack Block | Hides a variable's monitor from the stage. | variable name (dropdown) | hide variable [score v] | when I receive [game over v]\n hide variable [score v] (Hides the score variable when the game ends) 1 |
| show list [my list v] | Data | Stack Block | Makes a list's monitor visible on the stage. | list name (dropdown) | show list [shopping list v] | when green flag clicked\n show list [shopping list v] (Displays the shopping list on the stage at project start) 1 |
| hide list [my list v] | Data | Stack Block | Hides a list's monitor from the stage. | list name (dropdown) | hide list [shopping list v] | when I receive [game over v]\n hide list [shopping list v] (Hides the shopping list when the game ends) 1 |
| Ask () and Wait | Sensing | Stack Block | Displays an input box with specified text at the bottom of the screen, allowing users to input text, which is stored in the "Answer" block. | question (text) | ask [What is your name?] and wait | ask [What is your name?] and wait\n say join [Hello ] (answer) for (2) seconds (Asks for name, then greets the user using their input) 12 |
| Reset Timer | Sensing | Stack Block | Sets the timer’s value back to 0.0. | None | reset timer | when green flag clicked\n reset timer\n forever\n say (timer) (Resets timer at start, then continuously reports elapsed time) 12 |

### **JSON for Stack Blocks**

JSON

{  
 "block\_category": "Stack Blocks",  
 "description": "Stack blocks are the most common block shape, featuring a notch at the top and a bump at the bottom. They perform the main commands within a script and can connect both above and below them.",  
 "blocks":,  
 "example\_standalone": "move (10) steps",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "say () for () seconds",  
 "block\_type": "Looks",  
 "block\_shape": "Stack Block",  
 "functionality": "Displays a speech bubble containing specified text for a set duration.",  
 "inputs": [  
 {"name": "text", "type": "string"},  
 {"name": "seconds", "type": "number"}  
 ],  
 "example\_standalone": "say [Hello!] for (2) seconds",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when green flag clicked\n say [Grr] for (3) seconds\n say [Have you seen my honey?] for (3) seconds",  
 "explanation": "This script makes the sprite display two sequential speech bubbles with different messages and durations. [6, 7]"  
 }  
 ]  
 },  
 {  
 "block\_name": "switch costume to ()",  
 "block\_type": "Looks",  
 "block\_shape": "Stack Block",  
 "functionality": "Alters the sprite's appearance to a designated costume.",  
 "inputs": [  
 {"name": "costume", "type": "dropdown/number"}  
 ],  
 "example\_standalone": "switch costume to (costume1)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when I receive [explosion v]\n repeat (5)\n next costume\n end\n hide",  
 "explanation": "This script animates an explosion by rapidly switching costumes, then hides the sprite. [3]"  
 }  
 ]  
 },  
 {  
 "block\_name": "show",  
 "block\_type": "Looks",  
 "block\_shape": "Stack Block",  
 "functionality": "Makes the sprite visible on the stage.",  
 "inputs":,  
 "example\_standalone": "show",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when green flag clicked\n hide\nwhen I receive [start game v]\n show",  
 "explanation": "This script hides the sprite at the beginning of the project and makes it visible when a 'start game' broadcast is received."  
 }  
 ]  
 },  
 {  
 "block\_name": "hide",  
 "block\_type": "Looks",  
 "block\_shape": "Stack Block",  
 "functionality": "Makes the sprite invisible on the stage.",  
 "inputs":,  
 "example\_standalone": "hide",  
 "example\_with\_other\_blocks":  
 },  
 {  
 "block\_name": "play sound () until done",  
 "block\_type": "Sound",  
 "block\_shape": "Stack Block",  
 "functionality": "Plays a specified sound and pauses the script's execution until the sound has completed.",  
 "inputs": [  
 {"name": "sound name", "type": "dropdown"}  
 ],  
 "example\_standalone": "play sound [Meow v] until done",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "forever\n play sound [Music v] until done",  
 "explanation": "This script creates a continuous loop for background music, playing the 'Music' sound repeatedly. [8]"  
 }  
 ]  
 },  
 {  
 "block\_name": "start sound ()",  
 "block\_type": "Sound",  
 "block\_shape": "Stack Block",  
 "functionality": "Initiates playback of a specified sound without pausing the script, allowing other actions to proceed concurrently.",  
 "inputs": [  
 {"name": "sound name", "type": "dropdown"}  
 ],  
 "example\_standalone": "start sound [Pop v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when this sprite clicked\n start sound [Pop v]\n change [score v] by (1)",  
 "explanation": "This script plays a 'Pop' sound and increments the score simultaneously when the sprite is clicked. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "broadcast ()",  
 "block\_type": "Events",  
 "block\_shape": "Stack Block",  
 "functionality": "Sends a broadcast message throughout the Scratch program, activating any 'when I receive ()' blocks that are set to listen for that message, enabling indirect communication.",  
 "inputs": [  
 {"name": "message name", "type": "string/dropdown"}  
 ],  
 "example\_standalone": "broadcast [start game v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <key [space v] pressed?> then\n broadcast [jump v]\nend",  
 "explanation": "This script sends a 'jump' message to other scripts or sprites when the space key is pressed. [9]"  
 }  
 ]  
 },  
 {  
 "block\_name": "broadcast () and wait",  
 "block\_type": "Events",  
 "block\_shape": "Stack Block",  
 "functionality": "Sends a broadcast message and pauses the current script until all other scripts activated by that broadcast have completed their execution, ensuring sequential coordination.",  
 "inputs": [  
 {"name": "message name", "type": "string/dropdown"}  
 ],  
 "example\_standalone": "broadcast [initialize sprites v] and wait",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "broadcast [initialize sprites v] and wait\n say for (2) seconds",  
 "explanation": "This script ensures all sprite initialization routines complete before displaying 'Game Started!' for 2 seconds. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "wait () seconds",  
 "block\_type": "Control",  
 "block\_shape": "Stack Block",  
 "functionality": "Pauses the script for a specified duration.",  
 "inputs": [  
 {"name": "seconds", "type": "number"}  
 ],  
 "example\_standalone": "wait (1) seconds",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "say [Hello!] for (1) seconds\n wait (0.5) seconds\n say [Goodbye!] for (1) seconds",  
 "explanation": "This script creates a timed dialogue sequence, pausing for 0.5 seconds between two speech bubbles. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "create clone of ()",  
 "block\_type": "Control",  
 "block\_shape": "Stack Block",  
 "functionality": "Generates a copy, or clone, of a specified sprite.",  
 "inputs":}  
 ],  
 "example\_standalone": "create clone of [myself v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when green flag clicked\n hide\n forever\n create clone of [myself v]\n wait (1) seconds\n end",  
 "explanation": "This script continuously creates new clones of the current sprite every second after the original sprite hides itself. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "set [my variable v] to ()",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Assigns a specific value (number, string, or boolean) to a variable.",  
 "inputs": [  
 {"name": "variable name", "type": "dropdown"},  
 {"name": "value", "type": "any"}  
 ],  
 "example\_standalone": "set [score v] to (0)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when green flag clicked\n set [score v] to (0)\n set [player name v] to [Guest]",  
 "explanation": "This script initializes the 'score' variable to 0 and the 'player name' variable to 'Guest' when the project starts. [1, 10, 11]"  
 }  
 ]  
 },  
 {  
 "block\_name": "change [my variable v] by ()",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Increases or decreases a variable's numerical value by a specified amount.",  
 "inputs": [  
 {"name": "variable name", "type": "dropdown"},  
 {"name": "value", "type": "number"}  
 ],  
 "example\_standalone": "change [score v] by (1)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when this sprite clicked\n change [score v] by (1)",  
 "explanation": "This script increments the 'score' variable by 1 each time the sprite is clicked. [1, 10]"  
 }  
 ]  
 },  
 {  
 "block\_name": "add () to [my list v]",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Appends an item to the end of a list.",  
 "inputs": [  
 {"name": "item", "type": "any"},  
 {"name": "list name", "type": "dropdown"}  
 ],  
 "example\_standalone": "add [apple] to [shopping list v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when green flag clicked\n add [apple] to [shopping list v]\n add [banana] to [shopping list v]",  
 "explanation": "This script adds 'apple' and 'banana' as new items to the 'shopping list' when the project starts. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "delete () of [my list v]",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Removes an item from a list by its index or by selecting 'all' items.",  
 "inputs": [  
 {"name": "index/option", "type": "number/dropdown", "options": ["all", "last", "random"]},  
 {"name": "list name", "type": "dropdown"}  
 ],  
 "example\_standalone": "delete (1) of [my list v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when green flag clicked\n delete (all) of [my list v]",  
 "explanation": "This script clears all items from 'my list' when the green flag is clicked. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "insert () at () of [my list v]",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Inserts an item at a specific position within a list.",  
 "inputs": [  
 {"name": "item", "type": "any"},  
 {"name": "index", "type": "number"},  
 {"name": "list name", "type": "dropdown"}  
 ],  
 "example\_standalone": "insert [orange] at (2) of [fruits v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "insert [orange] at (2) of [fruits v]",  
 "explanation": "This script inserts 'orange' as the second item in the 'fruits' list, shifting subsequent items. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "replace item () of [my list v] with ()",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Replaces an item at a specific position in a list with a new value.",  
 "inputs": [  
 {"name": "index", "type": "number"},  
 {"name": "list name", "type": "dropdown"},  
 {"name": "new item", "type": "any"}  
 ],  
 "example\_standalone": "replace item (1) of [colors v] with [blue]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "replace item (1) of [colors v] with [blue]",  
 "explanation": "This script changes the first item in the 'colors' list to 'blue'. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "show variable [my variable v]",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Makes a variable's monitor visible on the stage.",  
 "inputs": [  
 {"name": "variable name", "type": "dropdown"}  
 ],  
 "example\_standalone": "show variable [score v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when green flag clicked\n show variable [score v]",  
 "explanation": "This script displays the 'score' variable on the stage when the project starts. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "hide variable [my variable v]",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Hides a variable's monitor from the stage.",  
 "inputs": [  
 {"name": "variable name", "type": "dropdown"}  
 ],  
 "example\_standalone": "hide variable [score v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when I receive [game over v]\n hide variable [score v]",  
 "explanation": "This script hides the 'score' variable when the 'game over' broadcast is received. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "show list [my list v]",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Makes a list's monitor visible on the stage.",  
 "inputs": [  
 {"name": "list name", "type": "dropdown"}  
 ],  
 "example\_standalone": "show list [shopping list v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when green flag clicked\n show list [shopping list v]",  
 "explanation": "This script displays the 'shopping list' on the stage when the project starts. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "hide list [my list v]",  
 "block\_type": "Data",  
 "block\_shape": "Stack Block",  
 "functionality": "Hides a list's monitor from the stage.",  
 "inputs": [  
 {"name": "list name", "type": "dropdown"}  
 ],  
 "example\_standalone": "hide list [shopping list v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when I receive [game over v]\n hide list [shopping list v]",  
 "explanation": "This script hides the 'shopping list' when the 'game over' broadcast is received. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "Ask () and Wait",  
 "block\_type": "Sensing",  
 "block\_shape": "Stack Block",  
 "functionality": "Displays an input box with specified text at the bottom of the screen, allowing users to input text, which is stored in the 'Answer' block.",  
 "inputs": [  
 {"name": "question", "type": "text"}  
 ],  
 "example\_standalone": "ask [What is your name?] and wait",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "ask [What is your name?] and wait\n say join [Hello ] (answer) for (2) seconds",  
 "explanation": "This script prompts the user for their name, waits for input, then greets them using the provided answer. [12, 13, 14]"  
 }  
 ]  
 },  
 {  
 "block\_name": "Reset Timer",  
 "block\_type": "Sensing",  
 "block\_shape": "Stack Block",  
 "functionality": "Sets the timer’s value back to 0.0.",  
 "inputs":,  
 "example\_standalone": "reset timer",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 }  
 ]  
}

## **Boolean Blocks: Logical Conditions**

Boolean blocks are instantly recognizable by their hexagonal shape.1 Their fundamental purpose is to represent conditions that evaluate to one of two states:

true or false.1 These blocks are typically integrated into the input slots of control flow blocks, such as

if <> then or repeat until <>, enabling decision-making and conditional execution within scripts.1 They are the logical "gates" that determine the path of a program's execution.

Boolean blocks serve as the direct visual representation of conditional statements (like if/else structures) and loop conditions (such as repeat until) found in all programming languages. Their hexagonal shape visually indicates where they can be placed, specifically within the conditional slots of C-blocks like if then or repeat until.1 This design concretely teaches propositional logic and the concept of a "condition" that must be met for a program to branch its execution or iterate through a sequence. The ability to combine multiple Boolean blocks using logical operators such as

and, or, and not 1 introduces Boolean algebra and the construction of complex logical expressions. These are fundamental for creating intelligent and responsive programs that can adapt their behavior based on multiple criteria. By making logical conditions tangible and visually distinct, Scratch demystifies control flow for beginners, laying the groundwork for understanding decision trees, state machines, and more complex algorithm design in advanced programming contexts.

Furthermore, many Boolean blocks, particularly those within the Sensing category, directly link user input and environmental detection to program logic.12 Blocks like

mouse down?, key pressed?, and touching? are frequently used to check for user interactions or sprite-to-sprite/color interactions.12 This demonstrates a core principle of interactive programming: a program's behavior is dynamically influenced by external events and data. It emphasizes the importance of continuous sensing, often implemented within

forever loops, to maintain responsiveness to user actions or changes in the game environment.3 The detailed behavior of blocks like

mouse down? 15 even introduces nuances related to different input devices (e.g., trackpads vs. mice) and the distinction between a quick click and a sustained press. This hands-on experience with input-driven logic is crucial for developing any interactive application, from games to user interfaces, and highlights the essential feedback loop between user action and program response, a cornerstone of user experience design.

### **Table 3: Boolean Block Details**

| Block Name | Block Type (Category) | Block Shape | Description | Inputs | Example (Standalone) | Example (With Other Blocks) |
| --- | --- | --- | --- | --- | --- | --- |
| <() < ()> | Operators | Boolean Block | Checks if the first value is less than the second. | value1 (any type), value2 (any type) | <(score) < (10)> | if <(score) < (10)> then\n say [Keep trying!] (Says "Keep trying!" if score is less than 10) 1 |
| <() = ()> | Operators | Boolean Block | Checks if two values are equal. | value1 (any type), value2 (any type) | <(answer) = > | if <(answer) = > then\n say [Correct!] (Says "Correct!" if the answer variable equals 5) 9 |
| <() > ()> | Operators | Boolean Block | Checks if the first value is greater than the second. | value1 (any type), value2 (any type) | <(health) > (0)> | if <(health) > (0)> then\n move (10) steps\nelse\n stop [all v]\nend (Moves sprite if health is positive, otherwise stops all scripts) |
| <<> and <>> | Operators | Boolean Block | Returns true if both provided Boolean conditions are true. | condition1 (Boolean), condition2 (Boolean) | <<mouse down?> and <touching [mouse-pointer]?> > | if <<mouse down?> and <touching [mouse-pointer]?> > then\n say [You're clicking me!] (Says "You're clicking me!" if mouse is down and touching sprite) 9 |
| <<> or <>> | Operators | Boolean Block | Returns true if at least one of the provided Boolean conditions is true. | condition1 (Boolean), condition2 (Boolean) | <<key [left arrow v] pressed?> or <key [a v] pressed?>> | if <<key [left arrow v] pressed?> or <key [a v] pressed?>> then\n change x by (-10)\nend (Moves sprite left if either left arrow or 'a' key is pressed) 3 |
| <not <>> | Operators | Boolean Block | Returns true if the provided Boolean condition is false, and false if it is true. | condition (Boolean) | <not <mouse down?>> | wait until <not <mouse down?>> (Pauses script until the mouse button is released) 17 |
| () contains ()? | Operators | Boolean Block | Checks if one string contains another string. | string1 (string), string2 (string) | [apple] contains [a]? | if <[answer] contains [yes]?> then\n say [Great!] (Says "Great!" if the answer contains "yes") 1 |
| <touching ()?> | Sensing | Boolean Block | Checks if its sprite is touching the mouse-pointer, edge, or another specified sprite. | object (dropdown: mouse-pointer, edge, sprite name) | <touching [edge v]?> | if <touching [edge v]?> then\n bounce off edge\nend (Makes sprite bounce if it touches the edge) 12 |
| <touching color ()?> | Sensing | Boolean Block | Checks whether its sprite is touching a specified color. | color (color picker) | <touching color [#FF0000]?> | if <touching color [#FF0000]?> then\n change [health v] by (-1)\nend (Decreases health if sprite touches red color) 12 |
| <color () is touching ()?> | Sensing | Boolean Block | Checks whether a color on its sprite is touching another color. | color1 (color picker), color2 (color picker) | <color [#00FF00] is touching [#FF0000]?> | if <color [#00FF00] is touching [#FF0000]?> then\n say [Collision!]\nend (Says "Collision!" if a green part of the sprite touches a red color) 12 |
| <key () pressed?> | Sensing | Boolean Block | Checks if a specified keyboard key is pressed. | key (dropdown: space, arrow keys, letters, etc.) | <key [space v] pressed?> | forever\n if <key [space v] pressed?> then\n broadcast [shoot v]\n end\nend (Continuously checks for space key press to broadcast "shoot") 3 |
| <mouse down?> | Sensing | Boolean Block | Checks if the computer mouse’s primary button is being clicked while the cursor is over the stage. | None | <mouse down?> | forever\n if <mouse down?> then\n go to (mouse-pointer v)\n end\nend (Sprite continuously follows mouse pointer while mouse button is held down) 1 |
| [my list v] contains ()? | Data | Boolean Block | Checks if a list includes a specific item. | list name (dropdown), item (any type) | [inventory v] contains [key]? | if <[inventory v] contains [key]?> then\n say [You have the key!] (Says "You have the key!" if the inventory list contains "key") 1 |

### **JSON for Boolean Blocks**

JSON

{  
 "block\_category": "Boolean Blocks",  
 "description": "Boolean blocks are hexagonal in shape. They represent conditions that evaluate to either 'true' or 'false' and are typically used as inputs for control flow blocks.",  
 "blocks":,  
 "example\_standalone": "<(score) < (10)>",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <(score) < (10)> then\n say [Keep trying!]",  
 "explanation": "This script causes the sprite to say 'Keep trying!' if the 'score' variable is less than 10. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "<() = ()>",  
 "block\_type": "Operators",  
 "block\_shape": "Boolean Block",  
 "functionality": "Checks if two values are equal.",  
 "inputs": [  
 {"name": "value1", "type": "any"},  
 {"name": "value2", "type": "any"}  
 ],  
 "example\_standalone": "<(answer) = >",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <(answer) = > then\n say [Correct!]",  
 "explanation": "This script makes the sprite say 'Correct!' if the value of the 'answer' variable is exactly 5. [9, 13, 14]"  
 }  
 ]  
 },  
 {  
 "block\_name": "<() > ()>",  
 "block\_type": "Operators",  
 "block\_shape": "Boolean Block",  
 "functionality": "Checks if the first value is greater than the second.",  
 "inputs": [  
 {"name": "value1", "type": "any"},  
 {"name": "value2", "type": "any"}  
 ],  
 "example\_standalone": "<(health) > (0)>",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <(health) > (0)> then\n move (10) steps\nelse\n stop [all v]\nend",  
 "explanation": "This script moves the sprite if its 'health' is greater than 0; otherwise, it stops all scripts. (Implied by [1])"  
 }  
 ]  
 },  
 {  
 "block\_name": "<<> and <>>",  
 "block\_type": "Operators",  
 "block\_shape": "Boolean Block",  
 "functionality": "Returns 'true' if both provided Boolean conditions are 'true'.",  
 "inputs":,  
 "example\_standalone": "<<mouse down?> and <touching [mouse-pointer]?> >",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <<mouse down?> and <touching [mouse-pointer]?> > then\n say [You're clicking me!]",  
 "explanation": "This script makes the sprite say 'You're clicking me!' only if the mouse button is pressed AND the mouse pointer is touching the sprite. [9, 17]"  
 }  
 ]  
 },  
 {  
 "block\_name": "<<> or <>>",  
 "block\_type": "Operators",  
 "block\_shape": "Boolean Block",  
 "functionality": "Returns 'true' if at least one of the provided Boolean conditions is 'true'.",  
 "inputs":,  
 "example\_standalone": "<<key [left arrow v] pressed?> or <key [a v] pressed?>>",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <<key [left arrow v] pressed?> or <key [a v] pressed?>> then\n change x by (-10)\nend",  
 "explanation": "This script moves the sprite left if either the left arrow key OR the 'a' key is pressed. [3]"  
 }  
 ]  
 },  
 {  
 "block\_name": "<not <>>",  
 "block\_type": "Operators",  
 "block\_shape": "Boolean Block",  
 "functionality": "Returns 'true' if the provided Boolean condition is 'false', and 'false' if it is 'true'.",  
 "inputs":,  
 "example\_standalone": "<not <mouse down?>>",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "() contains ()?",  
 "block\_type": "Operators",  
 "block\_shape": "Boolean Block",  
 "functionality": "Checks if one string contains another string.",  
 "inputs": [  
 {"name": "string1", "type": "string"},  
 {"name": "string2", "type": "string"}  
 ],  
 "example\_standalone": "[apple] contains [a]?",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <[answer] contains [yes]?> then\n say [Great!]",  
 "explanation": "This script makes the sprite say 'Great!' if the 'answer' variable contains the substring 'yes'. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "<touching ()?>",  
 "block\_type": "Sensing",  
 "block\_shape": "Boolean Block",  
 "functionality": "Checks if its sprite is touching the mouse-pointer, edge, or another specified sprite.",  
 "inputs":}  
 ],  
 "example\_standalone": "<touching [edge v]?>",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <touching [edge v]?> then\n bounce off edge\nend",  
 "explanation": "This script makes the sprite reverse direction if it comes into contact with the edge of the stage. [12]"  
 }  
 ]  
 },  
 {  
 "block\_name": "<touching color ()?>",  
 "block\_type": "Sensing",  
 "block\_shape": "Boolean Block",  
 "functionality": "Checks whether its sprite is touching a specified color.",  
 "inputs": [  
 {"name": "color", "type": "color"}  
 ],  
 "example\_standalone": "<touching color [#FF0000]?>",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <touching color [#FF0000]?> then\n change [health v] by (-1)\nend",  
 "explanation": "This script decreases the 'health' variable by 1 if the sprite touches any red color on the stage. [12]"  
 }  
 ]  
 },  
 {  
 "block\_name": "<color () is touching ()?>",  
 "block\_type": "Sensing",  
 "block\_shape": "Boolean Block",  
 "functionality": "Checks whether a specific color on its sprite is touching another specified color on the stage or another sprite.",  
 "inputs": [  
 {"name": "color1", "type": "color"},  
 {"name": "color2", "type": "color"}  
 ],  
 "example\_standalone": "<color [#00FF00] is touching [#FF0000]?>",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <color [#00FF00] is touching [#FF0000]?> then\n say [Collision!]\nend",  
 "explanation": "This script makes the sprite say 'Collision!' if a green part of the sprite touches a red color elsewhere in the project. [12]"  
 }  
 ]  
 },  
 {  
 "block\_name": "<key () pressed?>",  
 "block\_type": "Sensing",  
 "block\_shape": "Boolean Block",  
 "functionality": "Checks if a specified keyboard key is currently being pressed.",  
 "inputs": [  
 {"name": "key", "type": "dropdown", "options": ["space", "up arrow", "a", "..."]}  
 ],  
 "example\_standalone": "<key [space v] pressed?>",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "forever\n if <key [space v] pressed?> then\n broadcast [shoot v]\n end\nend",  
 "explanation": "This script continuously checks if the space key is pressed and, if so, sends a 'shoot' broadcast. [3, 12, 16, 18]"  
 }  
 ]  
 },  
 {  
 "block\_name": "<mouse down?>",  
 "block\_type": "Sensing",  
 "block\_shape": "Boolean Block",  
 "functionality": "Checks if the computer mouse's primary button is being clicked while the cursor is over the stage.",  
 "inputs":,  
 "example\_standalone": "<mouse down?>",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "[my list v] contains ()?",  
 "block\_type": "Data",  
 "block\_shape": "Boolean Block",  
 "functionality": "Checks if a list includes a specific item.",  
 "inputs": [  
 {"name": "list name", "type": "dropdown"},  
 {"name": "item", "type": "any"}  
 ],  
 "example\_standalone": "[inventory v] contains [key]?",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <[inventory v] contains [key]?> then\n say [You have the key!]",  
 "explanation": "This script makes the sprite say 'You have the key!' if the 'inventory' list contains the item 'key'. [1]"  
 }  
 ]  
 }  
 ]  
}

## **Reporter Blocks: Value Outputs**

Reporter blocks are distinguished by their rounded edges, a visual cue that immediately identifies them as blocks that output or "report" values.1 These values can be numerical, string-based, or other data types, and they are specifically designed to fit into the input slots of other blocks.1 This capability allows them to provide dynamic data for calculations, conditional evaluations, or direct display within a Scratch project.

Reporter blocks are crucial for enabling data flow between different components of a Scratch project, making programs dynamic and data-driven. Their design explicitly teaches the concept of "returning a value" or "reporting data," a fundamental aspect of function design in text-based programming languages. This allows data generated by one operation, such as a sprite's (x position) 1, the result of an arithmetic calculation like

() + () 1, or user input captured by

(answer) 12, to be seamlessly consumed as input by another block, such as

set to () 1,

if <> then 1, or

say ().1 This interoperability enables complex calculations, dynamic decision-making processes, and interactive displays. The versatility of the

() of () block 12, which can report various properties of different sprites or the stage, further highlights cross-sprite data access, a form of inter-object communication essential for multi-sprite projects. Ultimately, Reporter blocks serve as the "glue" that elevates Scratch projects beyond simple linear scripts to truly interactive and intelligent applications. They are indispensable for implementing core game mechanics like scoring and health, running simulations involving physics and movement, and creating dynamic data visualizations, thereby laying a practical foundation for understanding variables, functions, and data structures.

The comprehensive set of Operator Reporter blocks demonstrates Scratch's commitment to teaching core computational thinking skills related to data manipulation.1 This category includes standard arithmetic operations (

+, -, \*, /), string manipulation functions (join, letter of, length of), and advanced mathematical functions (abs of, sqrt of, trigonometric functions).1 This provides a visual and interactive introduction to fundamental data types (numbers, strings) and the various operations that can be performed on them. Users can build functional calculators 13, manage game scores 10, and process text inputs, directly engaging with concepts like variable assignment, expression evaluation, and data transformation. The

pick random block 1 further introduces the concepts of randomness and probability, which are crucial elements for designing engaging games and realistic simulations. These blocks are essential for creating dynamic and responsive programs that can process information and perform calculations, laying the groundwork for understanding more complex mathematical and logical reasoning, data structures, and algorithms in advanced programming contexts.

### **Table 4: Reporter Block Details**

| Block Name | Block Type (Category) | Block Shape | Description | Inputs | Example (Standalone) | Example (With Other Blocks) |
| --- | --- | --- | --- | --- | --- | --- |
| (x position) | Motion | Reporter Block | Reports the current X coordinate of the sprite on the stage. | None | (x position) | forever\n wait until <(x position) > (220)>\n change [Hits v] by (1)\nend (Increments 'Hits' variable when sprite's X position exceeds 220) 19 |
| (y position) | Motion | Reporter Block | Reports the current Y coordinate of the sprite on the stage. | None | (y position) | set [worms v] to (y position) (Sets 'worms' variable to the sprite's current Y position) 19 |
| (costume #) | Looks | Reporter Block | Reports the current costume number of the sprite. | None | (costume #) | say join [I am costume number ] (costume #) (Sprite says its current costume number) |
| (costume name) | Looks | Reporter Block | Reports the current costume name of the sprite. | None | (costume name) | say join [My costume is ] (costume name) (Sprite says its current costume name) |
| (volume) | Sound | Reporter Block | Reports the current volume level of the sprite. | None | (volume) | say join [Current volume: ] (volume) (Sprite says its current volume level) 1 |
| () + () | Operators | Reporter Block | Adds two numerical values. | number1 (number), number2 (number) | (5) + (3) | set [total v] to ( (number 1) + (number 2) ) (Calculates sum of two variables and stores in 'total') 13 |
| () - () | Operators | Reporter Block | Subtracts the second numerical value from the first. | number1 (number), number2 (number) | (10) - (4) | say join ( (num1) - (num2) ) (Reports the difference between two numbers) 1 |
| () \* () | Operators | Reporter Block | Multiplies two numerical values. | number1 (number), number2 (number) | (6) \* (7) | set [area v] to ( (length) \* (width) ) (Calculates area from length and width) 1 |
| () / () | Operators | Reporter Block | Divides the first numerical value by the second. | number1 (number), number2 (number) | (20) / (5) | set [average v] to ( (total score) / (number of students) ) (Calculates average) 1 |
| pick random () to () | Operators | Reporter Block | Generates a random integer within a specified inclusive range. | min (number), max (number) | pick random (1) to (10) | go to x: (pick random (-240) to (240)) y: (pick random (-180) to (180)) (Moves sprite to a random position on the stage) 1 |
| join ()() | Operators | Reporter Block | Concatenates two strings or values into a single string. | string1 (string/number), string2 (string/number) | join [Hello ][World!] | say join [Hello ][World!] (Sprite says "Hello World!") 1 |
| letter () of () | Operators | Reporter Block | Reports the character at a specific numerical position within a string. | index (number), text (string) | letter (1) of [apple] | say letter (1) of [apple] (Sprite says "a") 1 |
| length of () | Operators | Reporter Block | Reports the total number of characters in a given string. | text (string) | length of [banana] | say length of [banana] (Sprite says "6") 1 |
| () mod () | Operators | Reporter Block | Reports the remainder when the first number is divided by the second. | number1 (number), number2 (number) | (10) mod (3) | if <(number) mod (2) = (0)> then\n say [Even number] (Checks if a number is even) 1 |
| round () | Operators | Reporter Block | Rounds a numerical value to the nearest integer. | number (number) | round (3.7) | set [rounded score v] to (round (score)) (Rounds the score to the nearest whole number) 1 |
| ([abs v] of ()) | Operators | Reporter Block | Performs various mathematical functions (e.g., absolute value, square root, trigonometric functions). | function type (dropdown), value (number) | ([sqrt v] of (25)) | set [distance v] to ([sqrt v] of ( ( (x position) \* (x position) ) + ( (y position) \* (y position) ) )) (Calculates distance using Pythagorean theorem) 1 |
| (my variable) | Data | Reporter Block | Provides the current value stored in a variable. | variable name (dropdown) | (score) | say (score) (Sprite says the current value of the 'score' variable) 1 |
| item () of [my list v] | Data | Reporter Block | Reports the item located at a specific position in a list. | index/option (number/dropdown: last, random), list name (dropdown) | item (1) of [shopping list v] | say item (1) of [shopping list v] (Sprite says the first item in the 'shopping list') 1 |
| length of [my list v] | Data | Reporter Block | Provides the total number of items contained in a list. | list name (dropdown) | length of [my list v] | say join join (length of [shopping list v]) [ items in the list.] (Reports the number of items in a list) 1 |
| Answer | Sensing | Reporter Block | Holds the most recent text inputted using the Ask () and Wait block. | None | (answer) | ask [What is your name?] and wait\n say join [Hello ] (answer) (Greets the user using the text they typed in response to a question) 12 |
| Mouse X | Sensing | Reporter Block | Reports the mouse-pointer’s current X position on the stage. | None | (mouse x) | forever\n go to x: (mouse x) y: (mouse y) (Makes sprite follow the mouse pointer) 12 |
| Mouse Y | Sensing | Reporter Block | Reports the mouse-pointer’s current Y position on the stage. | None | (mouse y) | forever\n go to x: (mouse x) y: (mouse y) (Makes sprite follow the mouse pointer) 12 |
| Loudness | Sensing | Reporter Block | Reports the loudness of noise received by a microphone on a scale of 0 to 100. | None | (loudness) | if <(loudness) > (50)> then\n change size by (10)\nend (Increases sprite size if microphone detects loud noise) 12 |
| Timer | Sensing | Reporter Block | Reports the elapsed time since Scratch was launched or the timer was reset, increasing by 1 every second. | None | (timer) | when green flag clicked\n reset timer\n forever\n say (timer) (Continuously displays the elapsed time) 12 |
| () of () | Sensing | Reporter Block | Reports a specified value (e.g., x position, direction, costume number) of a specified sprite or the Stage. | value to report (dropdown), sprite/stage (dropdown) | (x position) of | say join ( (x position) of ) (Reports the X position of another sprite) 12 |
| Current () | Sensing | Reporter Block | Reports the current local year, month, date, day of the week, hour, minutes, or seconds. | time unit (dropdown) | current [hour v] | say join (current [hour v]) (Reports the current hour of the day) 12 |
| Days Since 2000 | Sensing | Reporter Block | Reports the number of days (and fractions of a day) since 00:00:00 UTC on January 1, 2000. | None | (days since 2000) | say join (days since 2000) (Reports the total days elapsed since January 1, 2000) 12 |

### **JSON for Reporter Blocks**

JSON

{  
 "block\_category": "Reporter Blocks",  
 "description": "Reporter blocks have rounded edges. Their purpose is to report values, which can be numbers or strings, and are designed to fit into input slots of other blocks.",  
 "blocks":,  
 "example\_standalone": "(x position)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "forever\n wait until <(x position) > (220)>\n change [Hits v] by (1)\nend",  
 "explanation": "This script continuously checks the sprite's X position and increments a 'Hits' variable once it moves beyond 220. [19]"  
 }  
 ]  
 },  
 {  
 "block\_name": "(y position)",  
 "block\_type": "Motion",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the current Y coordinate of the sprite on the stage.",  
 "inputs":,  
 "example\_standalone": "(y position)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "set [worms v] to (y position)",  
 "explanation": "This script assigns the sprite's current Y position to the 'worms' variable. [19]"  
 }  
 ]  
 },  
 {  
 "block\_name": "(costume #)",  
 "block\_type": "Looks",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the current costume number of the sprite.",  
 "inputs":,  
 "example\_standalone": "(costume #)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "say join [I am costume number ] (costume #)",  
 "explanation": "This script makes the sprite display its current costume number in a speech bubble."  
 }  
 ]  
 },  
 {  
 "block\_name": "(costume name)",  
 "block\_type": "Looks",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the current costume name of the sprite.",  
 "inputs":,  
 "example\_standalone": "(costume name)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "say join [My costume is ] (costume name)",  
 "explanation": "This script makes the sprite display its current costume name in a speech bubble."  
 }  
 ]  
 },  
 {  
 "block\_name": "(volume)",  
 "block\_type": "Sound",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the current volume level of the sprite.",  
 "inputs":,  
 "example\_standalone": "(volume)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "say join [Current volume: ] (volume)",  
 "explanation": "This script makes the sprite display its current volume level in a speech bubble. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "() + ()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Adds two numerical values.",  
 "inputs": [  
 {"name": "number1", "type": "number"},  
 {"name": "number2", "type": "number"}  
 ],  
 "example\_standalone": "(5) + (3)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "set [total v] to ( (number 1) + (number 2) )",  
 "explanation": "This script calculates the sum of 'number 1' and 'number 2' and stores the result in the 'total' variable. [13, 14]"  
 }  
 ]  
 },  
 {  
 "block\_name": "() - ()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Subtracts the second numerical value from the first.",  
 "inputs": [  
 {"name": "number1", "type": "number"},  
 {"name": "number2", "type": "number"}  
 ],  
 "example\_standalone": "(10) - (4)",  
 "example\_with\_other\_blocks": ( (num1) - (num2) )",  
 "explanation": "This script makes the sprite display the result of subtracting 'num2' from 'num1'. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "() \* ()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Multiplies two numerical values.",  
 "inputs": [  
 {"name": "number1", "type": "number"},  
 {"name": "number2", "type": "number"}  
 ],  
 "example\_standalone": "(6) \* (7)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "set [area v] to ( (length) \* (width) )",  
 "explanation": "This script calculates the area by multiplying 'length' and 'width' variables and stores it in the 'area' variable. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "() / ()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Divides the first numerical value by the second.",  
 "inputs": [  
 {"name": "number1", "type": "number"},  
 {"name": "number2", "type": "number"}  
 ],  
 "example\_standalone": "(20) / (5)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "set [average v] to ( (total score) / (number of students) )",  
 "explanation": "This script calculates the average by dividing 'total score' by 'number of students' and stores it in the 'average' variable. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "pick random () to ()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Generates a random integer within a specified inclusive range.",  
 "inputs": [  
 {"name": "min", "type": "number"},  
 {"name": "max", "type": "number"}  
 ],  
 "example\_standalone": "pick random (1) to (10)",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "join ()()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Concatenates two strings or values into a single string.",  
 "inputs": [  
 {"name": "string1", "type": "string/number"},  
 {"name": "string2", "type": "string/number"}  
 ],  
 "example\_standalone": "join [Hello ][World!]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "say join [Hello ][World!]",  
 "explanation": "This script makes the sprite display 'Hello World!' in a speech bubble by joining two string literals. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "letter () of ()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the character at a specific numerical position within a string.",  
 "inputs": [  
 {"name": "index", "type": "number"},  
 {"name": "text", "type": "string"}  
 ],  
 "example\_standalone": "letter (1) of [apple]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "say letter (1) of [apple]",  
 "explanation": "This script makes the sprite display the first character of the string 'apple', which is 'a'. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "length of ()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the total number of characters in a given string.",  
 "inputs": [  
 {"name": "text", "type": "string"}  
 ],  
 "example\_standalone": "length of [banana]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "say length of [banana]",  
 "explanation": "This script makes the sprite display the length of the string 'banana', which is 6. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "() mod ()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the remainder when the first number is divided by the second.",  
 "inputs": [  
 {"name": "number1", "type": "number"},  
 {"name": "number2", "type": "number"}  
 ],  
 "example\_standalone": "(10) mod (3)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <(number) mod (2) = (0)> then\n say [Even number]",  
 "explanation": "This script checks if a 'number' variable is even by checking if its remainder when divided by 2 is 0. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "round ()",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Rounds a numerical value to the nearest integer.",  
 "inputs": [  
 {"name": "number", "type": "number"}  
 ],  
 "example\_standalone": "round (3.7)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "set [rounded score v] to (round (score))",  
 "explanation": "This script rounds the 'score' variable to the nearest whole number and stores it in 'rounded score'. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "([abs v] of ())",  
 "block\_type": "Operators",  
 "block\_shape": "Reporter Block",  
 "functionality": "Performs various mathematical functions (e.g., absolute value, square root, trigonometric functions).",  
 "inputs": [  
 {"name": "function type", "type": "dropdown", "options": ["abs", "sqrt", "sin", "cos", "tan", "..."]},  
 {"name": "value", "type": "number"}  
 ],  
 "example\_standalone": "([sqrt v] of (25))",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "set [distance v] to ([sqrt v] of ( ( (x position) \* (x position) ) + ( (y position) \* (y position) ) ))",  
 "explanation": "This script calculates the distance from the origin (0,0) using the Pythagorean theorem and stores it in 'distance'. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "(my variable)",  
 "block\_type": "Data",  
 "block\_shape": "Reporter Block",  
 "functionality": "Provides the current value stored in a variable.",  
 "inputs": [  
 {"name": "variable name", "type": "dropdown"}  
 ],  
 "example\_standalone": "(score)",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "item () of [my list v]",  
 "block\_type": "Data",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the item located at a specific position in a list.",  
 "inputs": [  
 {"name": "index/option", "type": "number/dropdown", "options": ["last", "random"]},  
 {"name": "list name", "type": "dropdown"}  
 ],  
 "example\_standalone": "item (1) of [shopping list v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "say item (1) of [shopping list v]",  
 "explanation": "This script makes the sprite display the first item from the 'shopping list'. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "length of [my list v]",  
 "block\_type": "Data",  
 "block\_shape": "Reporter Block",  
 "functionality": "Provides the total number of items contained in a list.",  
 "inputs": [  
 {"name": "list name", "type": "dropdown"}  
 ],  
 "example\_standalone": "length of [my list v]",  
 "example\_with\_other\_blocks": join (length of [shopping list v]) [ items in the list.]",  
 "explanation": "This script makes the sprite display the total number of items currently in the 'shopping list'. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "Answer",  
 "block\_type": "Sensing",  
 "block\_shape": "Reporter Block",  
 "functionality": "Holds the most recent text inputted using the 'Ask () and Wait' block.",  
 "inputs":,  
 "example\_standalone": "(answer)",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "ask [What is your name?] and wait\n say join [Hello ] (answer)",  
 "explanation": "This script prompts the user for their name and then uses the 'answer' block to incorporate their input into a greeting. [12]"  
 }  
 ]  
 },  
 {  
 "block\_name": "Mouse X",  
 "block\_type": "Sensing",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the mouse-pointer’s current X position on the stage.",  
 "inputs":,  
 "example\_standalone": "(mouse x)",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "Mouse Y",  
 "block\_type": "Sensing",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the mouse-pointer’s current Y position on the stage.",  
 "inputs":,  
 "example\_standalone": "(mouse y)",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "Loudness",  
 "block\_type": "Sensing",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the loudness of noise received by a microphone on a scale of 0 to 100.",  
 "inputs":,  
 "example\_standalone": "(loudness)",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "Timer",  
 "block\_type": "Sensing",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the elapsed time since Scratch was launched or the timer was reset, increasing by 1 every second.",  
 "inputs":,  
 "example\_standalone": "(timer)",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "() of ()",  
 "block\_type": "Sensing",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports a specified value (e.g., x position, direction, costume number) of a specified sprite or the Stage.",  
 "inputs": [  
 {"name": "value to report", "type": "dropdown", "options": ["x position", "y position", "direction", "..."]},  
 {"name": "sprite/stage", "type": "dropdown", "options":}  
 ],  
 "example\_standalone": "(x position) of",  
 "example\_with\_other\_blocks": ( (x position) of )",  
 "explanation": "This script makes the current sprite display the X position of 'Sprite1'. [12]"  
 }  
 ]  
 },  
 {  
 "block\_name": "Current ()",  
 "block\_type": "Sensing",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the current local year, month, date, day of the week, hour, minutes, or seconds.",  
 "inputs": [  
 {"name": "time unit", "type": "dropdown", "options": ["year", "month", "date", "day of week", "hour", "minute", "second"]}  
 ],  
 "example\_standalone": "current [hour v]",  
 "example\_with\_other\_blocks": (current [hour v])",  
 "explanation": "This script makes the sprite display the current hour of the day based on the user's computer clock. [12]"  
 }  
 ]  
 },  
 {  
 "block\_name": "Days Since 2000",  
 "block\_type": "Sensing",  
 "block\_shape": "Reporter Block",  
 "functionality": "Reports the number of days (and fractions of a day) since 00:00:00 UTC on January 1, 2000.",  
 "inputs":,  
 "example\_standalone": "(days since 2000)",  
 "example\_with\_other\_blocks": (days since 2000)",  
 "explanation": "This script makes the sprite display the total number of days elapsed since January 1, 2000. [12]"  
 }  
 ]  
 }  
 ]  
}

## **C Blocks: Control Flow Structures**

C blocks, named for their distinctive "C" shape, are fundamental to managing the flow of scripts within a Scratch project.1 Their design allows other blocks to be placed within their opening, enabling either repetitive execution or conditional branching.1 These blocks are essential for implementing core programming constructs such as looping, conditional execution, and dynamic script control.1

C blocks serve as the direct visual analogues of fundamental programming constructs like loops (e.g., for, while, do-while) and conditionals (e.g., if-then, if-then-else) found in all text-based programming languages. Their C-shaped form visually indicates that they enclose a sequence of other blocks, controlling their repetition or conditional execution.1 This design provides an intuitive, drag-and-drop method for implementing iterative and branching logic, thereby teaching the critical importance of control flow in determining program behavior. The

forever block, for instance, introduces the concept of an infinite loop, which is crucial for continuous processes in games or simulations, while also subtly highlighting the need for explicit termination conditions, such as a stop all block. Similarly, the if <> then else block 1 explicitly introduces the concept of alternative execution paths based on whether a given condition evaluates to true or false. By abstracting these complex control flow mechanisms into simple visual blocks, Scratch makes core programming concepts highly accessible to beginners. This approach builds a strong conceptual foundation for understanding algorithms, program efficiency, and debugging logical errors, preparing users for more advanced programming paradigms.

While C blocks are powerful, particularly the forever block, their usage requires careful consideration to avoid potential performance degradation. The continuous repetition of code within a forever loop, if not managed judiciously, can lead to a project lagging or slowing down, especially when multiple such loops run concurrently or contain computationally intensive operations. This observation highlights the concept of computational load and resource management, indicating that even though Scratch abstracts away many low-level details, the underlying execution environment still operates with finite processing power. Excessive concurrent infinite loops can exhaust these resources, leading to performance bottlenecks. This introduces the idea of optimization and efficient script design, encouraging users to consider the computational cost of their blocks. This understanding is crucial for developing robust and performant Scratch projects, particularly interactive games where smooth operation is paramount. It subtly introduces concepts like CPU cycles and frame rates, and the importance of efficient code for a seamless user experience. For users who might transition to other programming languages, it reinforces the universal need to consider algorithmic efficiency and resource consumption.

### **Table 5: C Block Details**

| Block Name | Block Type (Category) | Block Shape | Description | Inputs | Example (Standalone) | Example (With Other Blocks) |
| --- | --- | --- | --- | --- | --- | --- |
| repeat () | Control | C Block | Executes the blocks contained within its loop a specified number of times. | count (number) | repeat (10) | repeat (10)\n move (10) steps\n wait (0.1) seconds\nend (Moves sprite 10 steps, waits, repeats 10 times) 1 |
| forever | Control | C Block | Continuously loops the blocks inside until the project is manually stopped or a stop all block is activated. | None | forever | when green flag clicked\n forever\n go to (mouse-pointer v)\n end\nend (Sprite continuously follows the mouse pointer) |
| if <> then | Control | C Block | Executes the blocks contained within its conditional branch only if a Boolean condition evaluates to true. | condition (Boolean) | if <touching [mouse-pointer]?> then | if <touching [mouse-pointer]?> then\n say [I'm being touched!]\nend (Sprite says "I'm being touched!" if mouse pointer touches it) 1 |
| if <> then else | Control | C Block | Executes one set of blocks if a condition is true, and an alternative set of blocks if the condition is false. | condition (Boolean) | if <(score) > (10)> then else | if <(score) > (10)> then\n say [You win!]\nelse\n say [Keep trying!]\nend (Says "You win!" if score > 10, else "Keep trying!") 1 |
| repeat until <> | Control | C Block | Loops the blocks contained within its structure until a Boolean condition becomes true. | condition (Boolean) | repeat until <key [space v] pressed?> | repeat until <key [space v] pressed?>\n move (5) steps\nend (Moves sprite 5 steps repeatedly until space key is pressed) 1 |

### **JSON for C Blocks**

JSON

{  
 "block\_category": "C Blocks",  
 "description": "C blocks are shaped like the letter 'C'. They are used to loop or conditionally execute blocks that are placed within their opening, managing the flow of scripts.",  
 "blocks":,  
 "example\_standalone": "repeat (10)",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "forever",  
 "block\_type": "Control",  
 "block\_shape": "C Block",  
 "functionality": "Continuously loops the blocks inside until the project is manually stopped or a 'stop all' block is activated.",  
 "inputs":,  
 "example\_standalone": "forever",  
 "example\_with\_other\_blocks":"  
 }  
 ]  
 },  
 {  
 "block\_name": "if <> then",  
 "block\_type": "Control",  
 "block\_shape": "C Block",  
 "functionality": "Executes the blocks contained within its conditional branch only if a Boolean condition evaluates to 'true'.",  
 "inputs":,  
 "example\_standalone": "if <touching [mouse-pointer]?> then",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <touching [mouse-pointer]?> then\n say [I'm being touched!]\nend",  
 "explanation": "This script makes the sprite say 'I'm being touched!' only if the mouse pointer is touching the sprite. [1, 9]"  
 }  
 ]  
 },  
 {  
 "block\_name": "if <> then else",  
 "block\_type": "Control",  
 "block\_shape": "C Block",  
 "functionality": "Executes one set of blocks if a condition is 'true', and an alternative set of blocks if the condition is 'false'.",  
 "inputs":,  
 "example\_standalone": "if <(score) > (10)> then else",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <(score) > (10)> then\n say [You win!]\nelse\n say [Keep trying!]\nend",  
 "explanation": "This script makes the sprite say 'You win!' if the score is greater than 10; otherwise, it says 'Keep trying!'. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "repeat until <>",  
 "block\_type": "Control",  
 "block\_shape": "C Block",  
 "functionality": "Loops the blocks contained within its structure until a Boolean condition becomes 'true'.",  
 "inputs":,  
 "example\_standalone": "repeat until <key [space v] pressed?>",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "repeat until <key [space v] pressed?>\n move (5) steps\nend",  
 "explanation": "This script repeatedly moves the sprite 5 steps until the space key is pressed. [1]"  
 }  
 ]  
 }  
 ]  
}

## **Cap Blocks: Script Endings**

Cap blocks are distinguished by their notch at the top and a flat bottom.1 This unique shape signifies their role as the terminal point of a script, preventing any further blocks from being placed below them.1 They are specifically designed to provide explicit control over program execution flow by terminating scripts or specific actions.1

Cap blocks introduce the crucial concept of explicit program termination and fine-grained control over script execution. Blocks such as stop all, stop this script, and delete this clone 1 enable users to define clear endpoints for their programs or specific processes.

stop all 1 functions as a global program halt, effectively stopping all running scripts across the entire project. In contrast,

stop this script allows for localized script termination, affecting only the script in which it is placed without impacting other concurrent processes. The delete this clone block 1 is particularly vital for memory management and preventing an uncontrolled accumulation of unused sprite clones, which could otherwise lead to performance issues in complex projects. Understanding these termination conditions is essential for creating robust and well-behaved programs, preventing unintended infinite loops or resource leaks. This introduces concepts of process management and the lifecycle of program components, which are critical in more advanced programming environments.

A notable characteristic of the stop [v] block is its ability to dynamically change shape based on the selected option.1 While it typically functions as a Cap block when options like "all" or "this script" are chosen, it transforms into a Stack block if "other scripts in sprite" is selected.15 This dynamic behavior illustrates a more advanced, context-dependent syntax within Scratch, demonstrating that block properties, such as shape (which dictates connectivity), can be conditional based on internal parameters. When

stop [v] is set to stop "other scripts in sprite," it is not terminating the current script but rather performing an action *within* the current script's flow, hence its ability to have subsequent blocks stacked below it. This introduces the idea that even in highly abstracted visual programming environments, there can be nuanced, context-sensitive rules that govern how programming elements behave and connect. It encourages a deeper understanding of block semantics beyond just their primary function, hinting at the complexity of underlying language design and the flexibility it offers.

### **Table 6: Cap Block Details**

| Block Name | Block Type (Category) | Block Shape | Description | Inputs | Example (Standalone) | Example (With Other Blocks) |
| --- | --- | --- | --- | --- | --- | --- |
| stop [v] | Control | Cap Block (can be Stack) | Halts all scripts, only the current script, or other scripts within the same sprite. Its shape is dynamic. | option (dropdown: all, this script, other scripts in sprite) | stop [all v] | if <(health) = (0)> then\n stop [all v]\nend (Stops all scripts if health reaches zero, commonly used for game over) 9 |
| delete this clone | Control | Cap Block | Removes the clone that is executing it from the stage. | None | delete this clone | when I start as a clone\n wait until <touching [edge v]?>\n delete this clone (Clone deletes itself when it touches the edge of the stage) 1 |
| stop all sounds | Sound | Cap Block | Halts all sounds currently playing across all sprites and the Stage. | None | stop all sounds | when I receive [game over v]\n stop all sounds (Stops all sounds when a "game over" message is received) 1 |

### **JSON for Cap Blocks**

JSON

{  
 "block\_category": "Cap Blocks",  
 "description": "Cap blocks have a notch at the top and a flat bottom. They signify the end of a script, preventing any further blocks from being placed below them, and are used to terminate scripts or specific actions.",  
 "blocks": [  
 {  
 "block\_name": "stop [v]",  
 "block\_type": "Control",  
 "block\_shape": "Cap Block (dynamic: can be Stack)",  
 "functionality": "Halts all scripts, only the current script, or other scripts within the same sprite. Its shape can dynamically change based on the selected option.",  
 "inputs": [  
 {"name": "option", "type": "dropdown", "options": ["all", "this script", "other scripts in sprite"]}  
 ],  
 "example\_standalone": "stop [all v]",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "if <(health) = (0)> then\n stop [all v]\nend",  
 "explanation": "This script stops all running scripts in the project if the 'health' variable reaches 0, typically signifying a game over condition. [9, 15]"  
 }  
 ]  
 },  
 {  
 "block\_name": "delete this clone",  
 "block\_type": "Control",  
 "block\_shape": "Cap Block",  
 "functionality": "Removes the clone that is executing it from the stage.",  
 "inputs":,  
 "example\_standalone": "delete this clone",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when I start as a clone\n wait until <touching [edge v]?>\n delete this clone",  
 "explanation": "This script, run by a clone, causes the clone to disappear from the stage once it touches the edge. [1]"  
 }  
 ]  
 },  
 {  
 "block\_name": "stop all sounds",  
 "block\_type": "Sound",  
 "block\_shape": "Cap Block",  
 "functionality": "Halts all sounds currently playing across all sprites and the Stage.",  
 "inputs":,  
 "example\_standalone": "stop all sounds",  
 "example\_with\_other\_blocks": [  
 {  
 "script": "when I receive [game over v]\n stop all sounds",  
 "explanation": "This script immediately stops all audio playback in the project when a 'game over' message is broadcast. [1]"  
 }  
 ]  
 }  
 ]  
}

## **Advanced Considerations**

Beyond the fundamental categorization and functionality of Scratch 3.0 blocks, a deeper examination reveals several advanced considerations related to the platform's internal architecture and performance characteristics.

### **Discussion of "Hidden Opcodes" and Their Internal Significance**

The research consistently references "hidden opcodes".1 These are internal commands or definitions utilized by the Scratch Virtual Machine (VM) that are not exposed as user-facing blocks in the palette. Examples include

procedures\_prototype for the underlying definition of custom blocks 1,

looks\_hideallsprites (an obsolete Looks block) 1,

music\_midiPlayDrumForBeats (related to an obsolete Music Extension) 1,

event\_whentouchingobject (an obsolete Events block) 1, and

control\_for\_each (an internal loop mechanism).1

The existence of these hidden opcodes demonstrates that even a high-level visual programming language like Scratch operates with multiple layers of abstraction. This reveals that user-friendly, drag-and-drop blocks are built upon a more complex, lower-level set of instructions. It illustrates the concept of an abstraction layer, where intricate details are deliberately concealed to simplify usage for the end-user. These hidden opcodes can be considered the "machine code" or "assembly language" of the Scratch VM, responsible for translating the visual blocks into executable instructions. The presence of obsolete opcodes also provides a glimpse into the evolutionary nature of software development, highlighting the ongoing refinement of the platform and the need for backward compatibility in certain instances. This understanding is crucial for users who may eventually transition to text-based programming, where they will directly interact with more granular commands and system calls. It demystifies how high-level languages function by showing that even visual blocks have an underlying, structured representation, such as the JSON format used for project files, as implied by the sophisticated internal representation of custom blocks.1 This reinforces that programming fundamentally involves translating human intent into machine-understandable instructions through various levels of abstraction.

### **Analysis of Performance Implications for Specific Block Operations**

While Scratch aims to abstract away computational complexities, certain block implementations can introduce non-obvious performance characteristics, challenging the assumption that visual programming inherently eliminates all computational overhead.

One notable example is the set volume to Sound block.1 Its use can introduce "imposed delays" and cause the game to "yield one frame," even when placed within a "wait without screen refresh block." This behavior is attributed to the internal implementation, specifically the use of

Promise.resolve() within the scratch3\_sound.js file, which indicates an intentional yielding mechanism.1 This suggests that the internal implementation of certain sound blocks can introduce unexpected performance characteristics, such as temporarily pausing the main thread, possibly due to the nature of audio processing or a deliberate choice to prevent the user interface from becoming unresponsive for extended periods.

Another critical performance consideration arises with the replace item () of () with () Data block.1 This block exhibits significantly slower performance in Scratch 3.0 compared to Scratch 2.0, particularly with large lists. This is due to an O(N) runtime complexity issue, attributed to the underlying JavaScript

splice() method.1 In contrast, a proposed solution involving direct index assignment (

list.value[index - 1] = item) offers an O(1) operation, resulting in a "huge performance impact" for projects manipulating extensive datasets.1 This reveals that even in a high-level, block-based language, underlying technical decisions regarding data manipulation directly influence script performance. It introduces the concept of

*runtime complexity* (O(N) vs. O(1)) in a practical context, demonstrating that the "block-based" nature does not imply "performance-agnosticism." For advanced users, understanding these subtle performance considerations becomes critical for debugging and optimizing complex projects. It teaches that even simple-looking operations can have significant computational costs under the hood, guiding decisions on how to structure data and select operations for optimal performance, especially in interactive applications or games where responsiveness is key. This also suggests a trade-off between simplicity of use and absolute performance optimization within the Scratch VM.

### **The Role of "My Blocks" (Custom Blocks) in Promoting Code Reusability and Abstraction**

"My Blocks," also known as Custom Blocks, represent a powerful feature in Scratch 3.0 that allows users to define reusable sequences of code, encapsulating complex logic into a single, custom-named block.1 These blocks are typically color-coded red, light red, or pink, and initially appear blank in the block palette until defined.1

This functionality directly introduces and encourages fundamental software engineering principles such as modularity, abstraction, and code reusability. Modularity is achieved by enabling users to break down large, intricate problems into smaller, self-contained custom blocks, making the overall project more manageable and easier to understand. Abstraction is demonstrated as the internal complexity of a custom block is hidden behind its simple name and inputs, allowing users to focus on *what* the block does rather than *how* it does it. Reusability is evident as the same custom block can be invoked multiple times throughout a project without the need to duplicate the underlying code sequence, analogous to writing functions or methods in text-based programming languages. Custom blocks can be designed to accept inputs (arguments) and can function as commands (performing actions), reporters (returning values), or predicates (returning Boolean true/false values), further enhancing their versatility.1 This feature serves as a critical bridge for users transitioning from simple, linear scripts to more structured and organized projects. By providing these tools, Scratch empowers users to build more scalable and maintainable projects, fostering good programming habits early in their development journey. It also subtly introduces the concept of local scope for variables defined as inputs within a custom block's definition.1

## **Conclusion**

Scratch 3.0's block design stands as a masterclass in visual programming, strategically leveraging distinct shapes and colors to intuitively convey functionality and connectivity.1 This meticulously crafted system guides users through fundamental programming concepts with remarkable clarity: Hat blocks introduce event handling, Stack blocks facilitate sequential execution, C blocks enable conditional logic and iteration, Boolean blocks provide logical conditions, Reporter blocks manage data flow, and Cap blocks ensure controlled script termination.1

The platform's design fosters computational thinking, problem-solving, and creative expression, making programming accessible and engaging for a wide audience, from novice learners to more advanced users. The inclusion of "My Blocks" further extends these capabilities, allowing users to practice advanced software engineering principles like abstraction and modularity, which are crucial for developing scalable and maintainable projects.1 Moreover, the underlying technical considerations, such as the presence of hidden opcodes and the nuanced performance characteristics of specific block operations, offer valuable insights for advanced users. These details bridge the conceptual gap between visual and text-based programming paradigms, demonstrating that even in a highly abstracted environment, an understanding of computational efficiency and internal architecture remains relevant for optimizing complex projects. Ultimately, Scratch 3.0 serves as an exceptional introductory platform that not only simplifies programming syntax but also instills core logical constructs, preparing users for future endeavors in diverse programming landscapes.

#### Works cited

1. Scratch 3.0 Block Descriptions\_.docx
2. Scratch Tutorial: When Green Flag Clicked Block - YouTube, accessed on July 1, 2025, <https://www.youtube.com/watch?v=JdWz4_hBwyQ>
3. Advanced Game Development Using Scratch 3.0 : 10 Steps (with Pictures) - Instructables, accessed on July 1, 2025, <https://www.instructables.com/Advanced-Game-Development-Using-Scratch-30/>
4. Scratch Blocks in 60 Seconds - The "move steps" Block - YouTube, accessed on July 1, 2025, <https://www.youtube.com/watch?v=ZCVb_EKTwQM>
5. Move 10 Steps in Scratch Programming - YouTube, accessed on July 1, 2025, <https://www.youtube.com/watch?v=zcJUnrhEhNg>
6. Scratch Basics - Episode 3: Make Your Sprite Say Something! - YouTube, accessed on July 1, 2025, <https://www.youtube.com/watch?v=lTVpjvPGrAE>
7. Say () - Scratch Wiki, accessed on July 1, 2025, <https://en.scratch-wiki.info/wiki/Say_()>
8. Play Sound () Until Done - Scratch Wiki, accessed on July 1, 2025, <https://en.scratch-wiki.info/wiki/Play_Sound_()_Until_Done_(block)>
9. If () Then - Scratch Wiki, accessed on July 1, 2025, <https://en.scratch-wiki.info/wiki/If_()_Then_(block)>
10. Scratching the Surface: What Is a Variable in Scratch? - JetLearn, accessed on July 1, 2025, <https://www.jetlearn.com/blog/what-is-a-variable-in-scratch>
11. How to Define Variables in Scratch 3.0 | Tutorial - YouTube, accessed on July 1, 2025, <https://www.youtube.com/watch?v=5vK9IL4tCbk>
12. Sensing Blocks and Examples in PictoBlox - STEMpedia Education, accessed on July 1, 2025, <https://ai.thestempedia.com/extension/sensing/>
13. 3. Calculator - Scratch Programming - Alps Academy, accessed on July 1, 2025, <https://www.alps.academy/wp-content/uploads/2024/07/3-calculator.pdf>
14. How do i use Operator blocks? - Discuss Scratch, accessed on July 1, 2025, <https://scratch.mit.edu/discuss/topic/406350/?page=1>
15. Mouse Down? - Scratch Wiki, accessed on July 1, 2025, <https://en.scratch-wiki.info/wiki/Mouse_Down%3F_(block)>
16. Calling a Custom Block from within another script? - scratch - Reddit, accessed on July 1, 2025, <https://www.reddit.com/r/scratch/comments/mutg16/calling_a_custom_block_from_within_another_script/>
17. The "Stop ( )" Block | CreatiCode Scratch Forum, accessed on July 1, 2025, <https://forum.creaticode.com/topic/227/the-stop-block>
18. Sensing Blocks | PDF | Scratch (Programming Language) | Variable ..., accessed on July 1, 2025, <https://fr.scribd.com/document/346142028/sensing-blocks>
19. X Position (block) - Scratch Wiki, accessed on July 1, 2025, <https://en.scratch-wiki.info/wiki/X_Position_(block)>
20. Control Blocks - Scratch Wiki, accessed on July 1, 2025, <https://en.scratch-wiki.info/wiki/Control_Blocks>