ICEHELLIONX SCRIPT GUIDE

A COLLABORATIVE GUIDE TO ADVANCED SCRIPTING PRACTICES FOR THE JANITORAI COMMUNITY

GROWTH THROUGH CREATION

v.1.0 · OCTOBER 2025 · JANITORAI COMMUNITY RESOURCE

Part I: Foundations

Building the Basics of Sandbox Scripting

Foreword

This guide was written to help people who aren't coders learn how to make scripts that bring their characters and worlds to life.

For many, scripting feels like a mysterious wall of symbols and rules. My goal is to show you that it's not magic — it's just building blocks stacked carefully, one step at a time.

Whether you're here to build roleplay characters, manage world lore, or just tinker for fun, I want this guide to be something you can flip through without feeling lost.

Remember: scripting isn't about writing "perfect" code. It's about creating something that feels alive, fun, and responsive. Start small, experiment, and let your characters grow with you.

— *Icehellionx*

Introduction

Welcome to the *Script Making Guide*, a beginner-to-advanced handbook for writing scripts in a sandbox environment.

This guide starts at the very beginning:

- What a script is.
- How it interacts with personality and scenario.
- Simple keyword checks.

Then it builds steadily into intermediate and advanced topics — probability, Lorebook, reaction systems, and full modular frameworks.

Think of this book as your roadmap. Start here at Chapter 1, take it one step at a time, and by the time you finish, you'll be writing full lore engines of your own.

The only requirement? **Curiosity.**
If you can read, experiment, and follow examples step by step, you can script.

Chapter 1 – What Is a Script?

Imagine your character as an actor in a play. Normally, they follow the "script" you wrote in their **Personality** and **Scenario** fields. But what if you wanted them to change their lines depending on what the user says? That's what scripting allows.

A **script** is a small set of instructions — a recipe card — that reacts to what's happening in the chat.

If this happens → do that.
If the user says "hello" → make the character smile.

🟡 When Do Scripts Run?

Scripts are automatic. They run:

- Before every bot reply.
- Right after the user sends a message.
- Every time the chat moves forward.

That means your script is always "listening" in the background.

🟡 What Can Scripts Change?

Only two things:

1. **Personality** → how the character acts or feels
2. **Scenario** → what's happening around them

Everything else — name, memories, chat history — is read-only.

🟡 The Context Object (Your Toolbox)

When your script runs, it's given a **context** object — a box of information about the current chat.

```
context.character.personality  // add traits here
context.character.scenario     // add scene details here
```

```
context.chat.last_message      // last user message
context.chat.message_count     // total messages exchanged
```

In plain terms:

- *personality* is the actor's mood.
- *scenario* is the stage set.
- *last_message* is what the user just said.
- *message_count* is how long the "play" has been running.

🟡 A Tiny First Example

```
if (context.chat.last_message.toLowerCase().indexOf("hello") !== -1) {
    context.character.scenario += "They greet you warmly.";
    context.character.personality += "Friendly and welcoming.";
}
```

What it does:

- Checks the user's message for "hello."
- If found, adds short notes to the scene and personality.

```
🟡 Key Takeaways
```

- Scripts are *if-this-then-that* instructions.
- They only modify Personality and Scenario.
- They run automatically each message.
- The **context** object is your toolbox.

Chapter 2 –The Context Object (Your Toolbox)

Every time your script runs, it doesn't start from scratch — it's handed a **context** object. Think of this as a backpack full of useful information.

🟡 Inside context.character

Property	Description	Editable?
`name`	The character's actual name	❌ Read-only
`chat_name`	Nickname shown in chat	❌ Read-only
`example_dialogs`	Practice lines	❌ Read-only
`personality`	The actor's mood	✅ You can edit
`scenario`	The stage set	✅ You can edit

You can only *add to* personality and scenario.

🟡 Inside context.chat

Property	Description
`message_count`	Number of messages so far
`last_message`	Latest user input
`first_message_date`	When the chat began
`last_bot_message_date`	When the bot last replied

Most scripts will only use the first two.

🟡 Example: Checking Context

```
console.log("Last message was:", context.chat.last_message);
console.log("Total messages:", context.chat.message_count);
console.log("Current personality:", context.character.personality);
```

Logs are your testing friend — they don't appear in normal chat, just in debug.

🟡 Why Only Two Editable Fields?

Because safety matters. You can write on the **whiteboards** (personality and scenario), but you can't tear down the stage.

🟡 **Key Takeaways**

- The **context** object provides everything your script needs.
- You can *read* from chat data, *write* to personality/scenario.
- Keep changes small and focused for best results.

Chapter 3 –The Sandbox Rules (Safe Javascript)

Now that you've met your tools, let's talk about where you can use them safely.
You're working inside a **sandbox**, a simplified JavaScript environment that protects both your bot and your sanity.

🟡 **The Golden Rule**

You can **only use the tools** the sandbox provides.
If you try something unsupported, the script just fails silently.

🟢 **Safe Tools**

Category	Examples
Strings	`.toLowerCase(), .indexOf("word"), .trim()`
Math	`+, -, *, /, Math.random(), Math.floor()`
Arrays	`.length, .indexOf(), for loops`
Dates	`new Date(), .getHours()`
Regex	`/\bhello\b/i.test(text)`

Category	Examples
Debugging	`console.log()`

✅ These always work.

🔴 Unsafe Tools

- Arrow functions () => {}
- Template strings `Hello ${name}`
- Spread operators ...arr
- .map(), .filter(), .reduce(), .forEach()
- Async functions, fetch calls, timers

❌ These will fail silently.

⚠️ Gray Area Tools

```
.includes()
.repeat()
.padStart() / .padEnd()
```

They sometimes work, but not everywhere — stick to safe methods.

🟡 Example: Safe vs Unsafe

❌ Unsafe:

```
if (context.chat.last_message.includes("hello")) { ... }
```

✅ Safe:

```
if (context.chat.last_message.toLowerCase().indexOf("hello") !== -1) { ... }
```

🟡 Key Takeaways

- The sandbox keeps things fast, safe, and simple.
- Use older ES5-style JavaScript — it's guaranteed to work.

- Always test features before relying on them.

Chapter 4 – Safe Matching & String Handling

Now that you know what works, let's talk about how to make your scripts *recognize* words properly.

Keyword matching is one of the most common (and trickiest) beginner tasks.

🟡 Step 1: Normalize

Always lowercase the user's message.

```
var last = context.chat.last_message.toLowerCase();
```

🟡 Step 2: Pad with Spaces

Add a space at the start and end:

```
var padded = " " + last + " ";
```

This ensures " hello " won't trigger inside "shellows".

🟡 Step 3: Check Safely

```
if (padded.indexOf(" hello ") !== -1) {
    context.character.personality += "Friendly and welcoming.";
    context.character.scenario += "They greet you warmly.";
}
```

🟡 Step 4: Match Multiple Words

```
var greetings = ["hi", "hello", "hey"];
for (var i = 0; i < greetings.length; i++) {
    if (padded.indexOf(" " + greetings[i] + " ") !== -1) {
        context.character.personality += "Friendly and welcoming.";
        context.character.scenario += "They greet you warmly.";
        break;
    }
}
```

🟡 Step 5: Optional Regex

For multiple synonyms:

```
if (/\b(help|assist|aid)\b/i.test(last)) {
    context.character.personality += "Eager to be helpful.";
}
```

🟡 Key Takeaways

- Always lowercase and pad your messages.
- `.indexOf(" word ") !== -1` is your best friend.
- Keep matches simple; test often.
- Regex is optional — learn it later.

Chapter 5 – Progressive Mini-Examples (Building Fluency)

Now that you know how to write safe matches, let's build fluency — short, working examples that grow in complexity step by step.

🟢 Level 1: Single Trigger

```
if (padded.indexOf(" hello ") !== -1) {
    context.character.personality += "Friendly and welcoming.";
    context.character.scenario += "They greet you warmly.";
}
```

🟢 Level 2: Multi-Keyword Reaction

```
var greetings = ["hi", "hello", "hey"];
for (var i = 0; i < greetings.length; i++) {
    if (padded.indexOf(" " + greetings[i] + " ") !== -1) {
        context.character.scenario += "They greet you warmly.";
        break;
    }
}
```

🟢 Level 3: Emotion Detection

```
var emotions = ["happy", "sad", "angry"];
for (var i = 0; i < emotions.length; i++) {
    if (padded.indexOf(" " + emotions[i] + " ") !== -1) {
        context.character.scenario += "The user seems " + emotions[i] + ".";
        break;
    }
}
```

🟢 Level 4: Message Count Progression

```
if (context.chat.message_count > 10) {
    context.character.personality += ", more comfortable now.";
}
```

🟢 Level 5: Combining Concepts

Try combining emotion + timing for flavor:

```
if (padded.indexOf(" secret ") !== -1 && context.chat.message_count > 15) {
    context.character.personality += ", mysterious and cautious.";
}
```

🟡 Key Takeaways

- Build small, test often.
- Each example teaches one new trick.
- The goal here isn't fancy logic — it's *comfort* and *confidence*.

Part II – Interaction Design

Teaching scripts to listen, react, and grow with conversation.

Chapter 6 – Progressive Reactions & Pacing

In Part I you learned how to make scripts *react*.
Now we'll make them *evolve*—so your character feels like they're getting to know the user.

🟡 Growing Friendships (Message Count Progression)

Let your character warm up over time:

```
var count = context.chat.message_count;

if (count < 5) {
    context.character.personality += ", polite and formal";
    context.character.scenario += " This feels like a cautious first meeting.";
} else if (count < 15) {
    context.character.personality += ", becoming more casual";
    context.character.scenario += " The atmosphere is loosening up.";
} else if (count < 30) {
    context.character.personality += ", open and friendly";
    context.character.scenario += " You've both settled into an easy rhythm.";
} else {
    context.character.personality += ", deeply connected";
    context.character.scenario += " The bond feels strong and genuine.";
}
```

This simple trick gives the sense of a deepening relationship.

🟡 Event Beats for Pacing

You can drop in story-like moments to make conversations feel alive.

```
if (context.chat.message_count === 10) {
    context.character.scenario += " A phone rings in the distance.";
}
if (context.chat.message_count === 25) {
    context.character.scenario += " The weather shifts suddenly.";
}
```

Each beat marks a little milestone, just like acts in a show.

🟡 Tips for Smooth Pacing

- Think in "acts" of 5–10 messages.
- Keep events small and suggestive, not cinematic walls of text.
- Fewer but more meaningful changes feel more organic.

🟡 Key Takeaways

- Use message count to simulate growing familiarity.
- Sprinkle simple events to mark time.
- Keep outputs short—small touches build immersion.

Chapter 7 – Fake Memory & Context Recall

Real memory isn't available in sandbox scripts—but we can *fake* it convincingly.
All we need is clever use of the **scenario** field.

🟡 Remembering Names

```
var last = context.chat.last_message.toLowerCase();
if (last.indexOf("my name is") !== -1) {
    var match = context.chat.last_message.match(/my name is (\w+)/i);
    if (match) {
        context.character.scenario += " Remember: the user's name is " +
match[1] + ".";
    }
}
```

Now your bot will act like it knows the user's name later.

🟡 Remembering Likes and Dislikes

```
var last = context.chat.last_message.toLowerCase();
var likes = ["pizza", "music", "movies"];
var dislikes = ["spiders", "crowds"];

for (var i = 0; i < likes.length; i++) {
    if (last.indexOf(likes[i]) !== -1) {
        context.character.personality += ", remembers the user likes " +
likes[i];
    }
}
for (var j = 0; j < dislikes.length; j++) {
    if (last.indexOf(dislikes[j]) !== -1) {
        context.character.personality += ", avoids mentioning " + dislikes[j];
    }
}
```

These quick notes cue the AI to act consistently.

🟡 Hint of Continuity

Even without specifics, you can imply memory:

```
context.character.personality += ", seems to remember details from earlier.";
```

That small phrase encourages consistent tone.

🟡 **Key Takeaways**

- True memory isn't possible—*fake it with notes*.
- Store small facts in scenario or personality.
- Use this sparingly: a few reminders feel natural; dozens feel robotic.

Chapter 8 – Dynamic Triggers & Combined Conditions

Scripts become much richer when they respond to more than one cue.
We can layer conditions to detect combined contexts.

🟡 **Emotion + Topic Pairs**

```
if (padded.indexOf(" painting ") !== -1 && padded.indexOf(" happy ") !== -1) {
    context.character.scenario += " They joyfully describe their love of
painting.";
}
```

The bot now reacts differently when moods and subjects overlap.

🟡 **Time + Keyword**

```
if (context.chat.message_count > 15 && padded.indexOf(" secret ") !== -1) {
    context.character.personality += ", mysterious and cautious";
    context.character.scenario += " They whisper as if revealing something
hidden.";
}
```

Timing gates add story progression to your triggers.

🟡 **Combining Multiple Signals**

You can nest conditions to produce special flavor text:

```
if (padded.indexOf(" forest ") !== -1 && padded.indexOf(" night ") !== -1) {
    context.character.scenario += " The forest feels dark and alive with
distant sounds.";
}
```

🟡 Key Takeaways

- Combine conditions for richer logic.
- Keep each combination focused on one idea.
- Test edge cases—compound logic grows fast!

Chapter 9 – Event Lore (Randomized Story Beats)

Sometimes the world should move on its own.
Event Lore gives your chat background motion—like stage props quietly shifting.

🟡 Timed Events

```
if (context.chat.message_count === 10) {
    context.character.scenario += " A distant bell tolls softly.";
}
if (context.chat.message_count === 25) {
    context.character.scenario += " A warm breeze passes through.";
}
```

These happen automatically at certain milestones.

🟡 Random Ambient Events

```
if (Math.random() < 0.2) {
    context.character.scenario += " A bird flutters past, scattering dust
motes.";
}
```

Only a 20 % chance each message—perfect for atmosphere.

🟡 Event Pools for Variety

```
var events = [
```

```
 " A knock echoes faintly.",
 " Leaves rustle somewhere nearby.",
 " A clock chimes once, then falls silent."
];
if (Math.random() < 0.15) {
    var pick = events[Math.floor(Math.random() * events.length)];
    context.character.scenario += pick;
}
```

This creates gentle, unpredictable world motion.

🟡 **Best Practices**

- Keep events short—one line or sentence.
- Space them out (e.g., 10 % chance per turn).
- Use them to reinforce mood, not distract from dialogue.

🟡 **Key Takeaways**

- Event Lore keeps worlds feeling alive.
- Use timed beats for structure, random beats for texture.
- Less is more—one or two small surprises every dozen lines works best.

Transition to Part III

You've now mastered interaction: your scripts can respond, remember, and breathe. Next, we'll add *life's unpredictability*—probabilities, pacing windows, and environmental context—to make your worlds feel truly dynamic.

Part III – Growth & Dynamics

Adding realism, rhythm, and unpredictability to your scripts.

Chapter 10 – Weighted Lore & Probability

So far, your reactions have been predictable: if a keyword appears, something always happens. But real conversation is never that mechanical. Sometimes characters hesitate or surprise you. Let's add a touch of randomness.

🟡 The Digital Dice Roll

`Math.random()` returns a number between 0 and 1.
Use it to decide when something happens.

```
if (Math.random() < 0.5) {
    // 50% chance to trigger
    context.character.personality += ", remembering something fondly.";
}
```

Half the time this code will run; half the time it won't.
That unpredictability makes dialogue feel human.

🟡 Weighted Options

Instead of yes/no, you can roll for multiple possibilities.

```
var options = [
    { chance: 0.6, text: " They mention an old friend." },
    { chance: 0.3, text: " They grow thoughtful, lost in memory." },
    { chance: 0.1, text: " They fall silent, eyes distant." }
];

var roll = Math.random();
var total = 0;
for (var i = 0; i < options.length; i++) {
    total += options[i].chance;
    if (roll < total) {
        context.character.scenario += options[i].text;
        break;
    }
}
```

Now most chats get the common line, but rare moments add sparkle.

🟡 Controlled Surprises

Use randomness like seasoning—lightly.
Important lore should be certain; small details can vary.

🟡 Key Takeaways

- `Math.random()` is your digital dice.
- Weighted outcomes create variety.
- Randomness should add *flavor*, not confusion.

Chapter 11 – Min/Max Message Gating (Unlockable Content)

Gating lets you control *when* events unfold.
It's how you make secrets reveal themselves naturally over longer chats.

🟡 The Basic Gate

```
var count = context.chat.message_count;
if (count >= 5 && count <= 15) {
    context.character.scenario += " They still seem guarded.";
}
```

This note only appears while message count is between 5 and 15.

🟡 Layered Reveals

```
var count = context.chat.message_count;
if (count <= 15 && padded.indexOf(" secret ") !== -1) {
    context.character.personality += ", cautious about their secrets.";
}
if (count >= 16 && count <= 30 && padded.indexOf(" secret ") !== -1) {
    context.character.personality += ", finally ready to open up.";
}
if (count > 30 && padded.indexOf(" secret ") !== -1) {
    context.character.personality += ", burdened by secrets too heavy to
hide.";
}
```

The longer the chat, the deeper the disclosure.

🟡 Event Windows

```
if (count === 10) {
    context.character.scenario += " A distant bell marks a turning point.";
}
```

```
if (count > 20 && count < 25) {
    context.character.personality += ", nostalgic.";
}
```

Use equality (===) for precise beats and ranges (> && <) for flexible arcs.

🟡 Key Takeaways

- Gates create pacing and “chapters.”
- Combine gates with keywords for evolving lore.
- Use ranges, not single numbers, to keep flow natural.

Chapter 12 – Time & Environment Awareness

You can also tie behavior to the *real-world clock.*
Time adds atmosphere—your bot feels awake when you are.

🟡 Day vs Night Behavior

```
var hour = new Date().getHours();
if (hour < 6 || hour > 22) {
    context.character.personality += ", a bit sleepy";
    context.character.scenario += " It's late, and the world feels quiet.";
} else {
    context.character.personality += ", bright and energetic";
    context.character.scenario += " Daylight spills across the scene.";
}
```

🟡 Combining Time and Lore

```
var hour = new Date().getHours();
if (padded.indexOf(" forest ") !== -1) {
    if (hour > 6 && hour < 20) {
        context.character.scenario += " Sunlight filters through the canopy.";
    } else {
        context.character.scenario += " Moonlight paints silver shapes among
the trees.";
    }
}
```

Now the same keyword feels completely different depending on the hour.

🟡 Realism Through Rhythm

Small environmental shifts make the world feel *alive* even when no one mentions them.

🟡 Key Takeaways

- `new Date().getHours()` gives local time.
- Split personality/scenario based on hour ranges.
- Combine time with keywords for richer scenes.

Transition to Part IV

You now control time, pacing, and chance.
Next, we'll scale up from individual reactions to full **systems**—Lorebook, reaction engines, and adaptive frameworks that organize entire worlds.

Part IV – Systems Thinking

Designing modular, scalable, and adaptive scripting systems.

Chapter 13 – Lorebooks & Hierarchies

Up to now, you've been writing isolated reactions — small, self-contained "if this then that" rules.
That's powerful, but as projects grow, you'll want a way to manage dozens or hundreds of triggers cleanly.

That's where **Lorebooks** come in.

Think of a Lorebook as a **library of modular reactions** — each entry describes one aspect of your world or character, ready to be referenced when needed.

🟡 The Flat Lorebook

A simple Lorebook is just a collection of keywords and their effects.

```
var lore = [
```

```
    { key: "forest", text: " Tall trees surround you." },
    { key: "river", text: " Water murmurs softly nearby." },
    { key: "mountain", text: " The peaks cut sharply into the sky." }
];

for (var i = 0; i < lore.length; i++) {
    if (padded.indexOf(" " + lore[i].key + " ") !== -1) {
        context.character.scenario += lore[i].text;
        break;
    }
}
```

This lets you scale from three to thirty reactions easily.

🟡 Hierarchical Lorebooks

You can group related entries into categories for organization.

```
var lorebook = {
    places: [
        { key: "forest", text: " The air smells of pine." },
        { key: "river", text: " A cool mist rises from the water." }
    ],
    emotions: [
        { key: "happy", text: " Their smile seems effortless." },
        { key: "sad", text: " They speak softly, eyes downcast." }
    ]
};

for (var group in lorebook) {
    var entries = lorebook[group];
    for (var i = 0; i < entries.length; i++) {
        if (padded.indexOf(" " + entries[i].key + " ") !== -1) {
            context.character.scenario += entries[i].text;
            break;
        }
    }
}
```

This is the start of scalable scripting.

🟡 Priorities and Overlaps

When multiple matches occur, decide which should take precedence.
For example, emotion might override place.

🟡 Key Takeaways

- Lorebooks store many triggers in clean data form.
- Hierarchies add organization.
- Priorities prevent conflicts.
- This is your first real *system* instead of a collection of lines.

Chapter 14 – Shifts & Conditional Layers

Lore shouldn't stay static — moods, times, and settings shift.
Conditional layers let your world adapt dynamically.

🟡 Mood-Based Lore Shifts

```
var mood = "happy"; // this could be set elsewhere

if (mood === "happy") {
    context.character.scenario += " Sunlight feels warmer today.";
} else if (mood === "sad") {
    context.character.scenario += " The light seems muted and distant.";
}
```

Now your lore reacts to emotion states instead of keywords alone.

🟡 Environmental Layers

```
var isNight = (new Date().getHours() < 6 || new Date().getHours() > 21);

if (isNight) {
    context.character.scenario += " The stars shimmer faintly overhead.";
} else {
    context.character.scenario += " A soft breeze moves through the daylight.";
}
```

You can layer this with other systems (like Chapter 13's Lorebook) for deep immersion.

🟡 Nested Conditions (Advanced)

Combine multiple contexts to produce complex shifts.

```
if (padded.indexOf(" forest ") !== -1) {
    if (isNight) {
        context.character.scenario += " Crickets sing through the dark trees.";
    } else {
```

```
        context.character.scenario += " Shafts of light spill between green
leaves.";
    }
}
```

Each layer modifies the same base idea differently.

🟡 Key Takeaways

- Shifts create contextual depth.
- Combine mood, time, and place for realism.
- Keep logic modular—avoid giant “if” chains when possible.

Chapter 15 – Reaction Engines & Scoring Systems

A **reaction engine** turns scattered logic into a cohesive system.
Instead of writing one-off triggers, you calculate *how strongly* a reaction should fire.

🟡 Scoring System Basics

```
var score = 0;

if (padded.indexOf(" angry ") !== -1) score += 2;
if (padded.indexOf(" shout ") !== -1) score += 1;
if (padded.indexOf(" calm ") !== -1) score -= 1;

if (score >= 2) {
    context.character.personality += ", visibly irritated.";
} else if (score <= -1) {
    context.character.personality += ", serene and patient.";
}
```

This lets reactions scale in intensity.

🟡 Weighted Emotion Systems

You can also roll emotions randomly but with bias:

```
var emotions = [
    { name: "happy", weight: 5 },
    { name: "curious", weight: 3 },
    { name: "tired", weight: 2 }
];
```

```
var total = 0;
for (var i = 0; i < emotions.length; i++) total += emotions[i].weight;

var roll = Math.random() * total;
var running = 0;
for (var j = 0; j < emotions.length; j++) {
    running += emotions[j].weight;
    if (roll < running) {
        context.character.personality += ", feels " + emotions[j].name;
        break;
    }
}
```

This creates natural distribution—more common emotions appear more often.

🟡 Key Takeaways

- Reaction engines centralize logic.
- Scoring and weighting allow nuance.
- Great for mood tracking and adaptive behavior.

Chapter 16 – Adaptive Engines & Hybrid States

This is where systems start combining.
An **adaptive engine** mixes multiple signals — lore, memory, emotion, and probability — to produce layered reactions.

🟡 Example: Adaptive Emotion Engine

```
var score = 0;

if (padded.indexOf(" insult ") !== -1) score += 2;
if (padded.indexOf(" compliment ") !== -1) score -= 1;
if (padded.indexOf(" thank ") !== -1) score -= 1;

if (Math.random() < 0.2) score += 1; // occasional temper

if (score >= 2) {
    context.character.personality += ", defensive and tense.";
} else if (score <= -1) {
    context.character.personality += ", appreciative and calm.";
}
```

You've now combined keyword detection, scoring, and randomness — that's an adaptive system.

🟡 Hybrid Lore & Emotion Example

```
if (padded.indexOf(" forest ") !== -1) {
    if (context.character.personality.indexOf("tired") !== -1) {
        context.character.scenario += " The forest feels heavy and still.";
    } else {
        context.character.scenario += " The forest hums with gentle life.";
    }
}
```

The same scene feels different depending on the character's mood.

🟡 Key Takeaways

- Adaptive engines combine multiple layers of logic.
- Hybrid states tie personality and scenario together.
- This is the bridge to full-world frameworks.

Chapter 17 – The "Everything Lorebook" Framework

At the top of the skill tree sits the **Everything Lorebook** — a single modular system that can power entire characters, worlds, and storylines.

This is where all prior lessons come together.

🟡 Core Idea

Create structured categories for people, places, moods, and events, then loop through them systematically.

```
var everything = {
    people: [
        { key: "friend", text: " They think fondly of their friend." },
        { key: "enemy", text: " Tension sharpens their tone." }
    ],
    places: [
        { key: "forest", text: " The trees whisper with hidden life." },
        { key: "city", text: " The streets buzz with distant noise." }
```

```
    ],
    moods: [
        { key: "happy", text: " Their steps feel light." },
        { key: "sad", text: " Each word feels slower." }
    ]
};

for (var group in everything) {
    var entries = everything[group];
    for (var i = 0; i < entries.length; i++) {
        if (padded.indexOf(" " + entries[i].key + " ") !== -1) {
            context.character.scenario += entries[i].text;
            break;
        }
    }
}
```

This framework lets you plug in whole databases of lore and handle them dynamically.

🟡 Scaling Up

Add categories for:

- **Weather**
- **Events**
- **Objects**
- **Memories**

Each one can build on what you already have.

🟡 Performance Note

For large Lorebooks (100+ entries), keep loops efficient — break early and avoid unnecessary nesting.

🟡 **Key Takeaways**

- The Everything Lorebook unifies all your systems.
- Organize by theme, not by script file.
- With this, your world runs itself — you just add entries.

Transition to Part V

You've built a living system. Now it's time to polish it — improving performance, cleaning code, and mastering best practices so your scripts stay fast, safe, and shareable.

Part V – Optimization & Craft

Refining performance, efficiency, and creative polish.

Chapter 18 – Performance, Efficiency & Debugging

Once your scripts start growing, performance becomes just as important as creativity.
You don't need to optimize every line — but you *should* make sure your code runs smoothly and safely in long chats.

🟡 **Avoid Unnecessary Loops**

Loops are powerful but can slow things down if used carelessly.

// Inefficient: checks the same condition multiple times

```
for (var i = 0; i < 10; i++) {
    if (padded.indexOf("hello") !== -1) {
        context.character.personality += "Friendly.";
    }
}
```

✅ **Better:**

```
if (padded.indexOf("hello") !== -1) {
    context.character.personality += "Friendly.";
}
```

Always ask: *does this loop need to exist?*

🟡 Break Early

When using loops, stop as soon as you find what you need.

```
for (var i = 0; i < list.length; i++) {
    if (padded.indexOf(" " + list[i] + " ") !== -1) {
        context.character.scenario += " Reaction triggered.";
        break; // Stops scanning early
    }
}
```

This single break saves unnecessary checks.

🟡 Keep Personality & Scenario Short

Each addition to `context.character.personality` and .scenario increases total prompt length. Over time, that can cause bloat or unwanted behavior drift.

- Use short phrases instead of long sentences.
- Occasionally clear or overwrite if needed:

```
context.character.personality = "Focused and alert.";
```

🟡 Debugging with console.log()

You can print test output to see what's happening.

```
console.log("Current message:", context.chat.last_message);
console.log("Message count:", context.chat.message_count);
```

This doesn't affect the chat — it's just a dev tool.

🟡 Safe Guarding Conditions

Always check that a variable exists before using it.

```
if (context.chat.last_message && context.chat.last_message.toLowerCase) {
    // safe to process
}
```

This prevents silent errors.

🟡 Key Takeaways

- Keep loops short and efficient.
- Break early when you find a match.
- Use console.log() for testing.
- Clean up old or repetitive personality/scenario lines.

Chapter 19 – Best Practices & Style Conventions

Your scripts don't just need to *work* — they should also be readable and easy to share.
Let's standardize a few stylistic habits that make life easier for everyone in the JanitorAI community.

🟡 Consistent Indentation

Use two or four spaces per level, and stay consistent.
This makes nested conditions much clearer.

```
if (condition) {
    if (subcondition) {
        // do something
    }
}
```

🟡 Append vs Overwrite

Use **append (+=)** when building up personality and scenario, and **overwrite (=)** when resetting or changing direction.

✅ *Append Example*:

```
context.character.personality += ", cautious but friendly.";
```

✅ *Overwrite Example*:

```
context.character.scenario = "The city has fallen silent after the storm.";
```

Use overwriting sparingly — only when a scene truly shifts.

🟡 Comment Liberally

Explain why code exists, not just what it does.

```
// Adds gentle ambient world events at random
if (Math.random() < 0.1) {
    context.character.scenario += " A soft breeze drifts through.";
}
```

Future you (or someone else) will thank you.

🟡 **Keep Functions Small**

If you find yourself scrolling more than a few screens, break your logic into sections or helper snippets.

This makes it easier to debug and reuse.

🟡 **Share Modular Snippets**

When posting in the community, share smaller modules people can plug into their own work. That way everyone learns and improves together.

🟡 **Key Takeaways**

- Clean code is community-friendly code.
- Append for growth; overwrite for change.
- Comment intent, not mechanics.
- Modular scripts help others learn faster.

Chapter 20 – Bringing It All Together (Capstone)

It's time to combine everything you've learned — context, safety, pacing, Lorebooks, and adaptation — into a single, cohesive script example.

This isn't meant to be "perfect code."
It's a demonstration of how all these systems fit together.

Capstone Example: The Living Character Script

```
// CAPSTONE DEMO SCRIPT
var last = context.chat.last_message.toLowerCase();
var padded = " " + last + " ";
var count = context.chat.message_count;
var hour = new Date().getHours();

// --- Weighted emotion reaction ---
var mood = "neutral";
if (padded.indexOf(" happy ") !== -1) mood = "happy";
if (padded.indexOf(" sad ") !== -1) mood = "sad";
if (Math.random() < 0.1) mood = "tired"; // small random variance

// --- Basic pacing ---
if (count < 10) {
    context.character.personality += ", polite and measured.";
} else if (count < 25) {
    context.character.personality += ", more relaxed now.";
} else {
    context.character.personality += ", speaks freely and openly.";
}

// --- Time-based tone ---
if (hour < 6 || hour > 22) {
    context.character.scenario += " It's quiet and dark outside.";
} else {
    context.character.scenario += " Sunlight glows across the room.";
}

// --- Simple lorebook system ---
var lore = [
    { key: "forest", text: " The trees hum softly in the breeze." },
    { key: "river", text: " A soft current ripples nearby." },
    { key: "storm", text: " Thunder murmurs far in the distance." }
];

for (var i = 0; i < lore.length; i++) {
    if (padded.indexOf(" " + lore[i].key + " ") !== -1) {
        context.character.scenario += lore[i].text;
        break;
    }
}

// --- Adaptive twist based on mood ---
if (mood === "happy") {
    context.character.scenario += " Everything feels alive and bright.";
} else if (mood === "sad") {
    context.character.scenario += " The world feels muted and slow.";
} else if (mood === "tired") {
    context.character.scenario += " Each sound echoes a bit too long.";
}

// --- Random world events ---
if (Math.random() < 0.05) {
```

```
    var events = [
        " A bell rings somewhere unseen.",
        " The lights flicker for a moment.",
        " A shadow passes silently by."
    ];
    var e = events[Math.floor(Math.random() * events.length)];
    context.character.scenario += e;
}
```

🟡 Why It Works

- Uses safe string methods (indexOf, lowercase).
- Integrates pacing, time, and probability.
- Modular structure keeps it readable.
- One loop per system — efficient and clean.

🟡 Final Thoughts

If you've made it this far, you're no longer a beginner — you're a script crafter.
You can design adaptive personalities, dynamic environments, and immersive Lorebooks.

But most importantly, you can *teach others*.
This guide exists because of the community — every shared snippet, every test, every late-night debugging session helps the next scripter climb a little faster.

Keep experimenting, keep improving, and keep teaching.

You're part of the reason scripting keeps getting better.

— *Icehellionx*

Appendices

Cheat Sheet: Common Safe Tools

Purpose	Method	Example
Lowercasing	`.toLowerCase()`	msg.toLowerCase()

Purpose	Method	Example
Keyword Check	`.indexOf()`	if (msg.indexOf("hello") !== -1)
Random Roll	`Math.random()`	if (Math.random() < 0.3)
Array Loop	`for (var i = 0; i < arr.length; i++)`	iterate safely
Time	`new Date().getHours()`	determine hour of day
Debugging	`console.log()`	view script output

Glossary

- **Context Object** – The data your script works with (`context.character`, `context.chat`).
- **Personality** – Describes how your character feels and behaves.
- **Scenario** – Describes the current scene or environment.
- **Sandbox** – The limited, safe JavaScript environment scripts run in.
- **Lorebook** – A structured list of keywords and related text triggers.
- **Reaction Engine** – A system that assigns intensity or emotion values to reactions.
- **Adaptive Engine** – A combined system that reacts dynamically to multiple factors.

Troubleshooting & Testing Tips

- If a script seems silent, add console.log() lines to see what it's reading.
- If a keyword doesn't trigger, check casing and spacing (" hello " padding).
- If events repeat too often, lower the random chance.
- Test each system separately before combining them.

Template Library (Starter Scripts)

Blank Starter:

```
var last = context.chat.last_message.toLowerCase();
var padded = " " + last + " ";
Emotion Template:
if (padded.indexOf(" happy ") !== -1) {
    context.character.personality += ", cheerful.";
}
```

Lore Template:

```
var lore = [{ key: "forest", text: " Trees sway gently." }];
for (var i = 0; i < lore.length; i++) {
    if (padded.indexOf(" " + lore[i].key + " ") !== -1) {
        context.character.scenario += lore[i].text;
        break;
    }
}
```

Example Code: How to keep functional smaller, more readable, and more efficient.

Video on this topic for further reference: https://www.youtube.com/watch?v=-AzSRHiV9Cc

METHOD 1. INVERSION: Keep Ideal Case Last.

Reasoning: You want to focus on what is happening not "what is not allowed" allowing you to skip the checks and focus on relevant code.

Quick reminder: ! is NOT, || is OR. && is AND logical actions

BAD EXAMPLE

```
if (!isNight) {
  if (!isRaining) {
    if (!isWindy) {
      startSunnyDayActivities();
      // A function that adds information into context and personality but more
complex. For example.
```

```
      // context.character.scenario += "The sun is shining outside";
      // context.character.personality += "{{Char}} is having a wonderful time
and very eager to play on the beach!";
    }
  }
}
```

GOOD EXAMPLE

```
//Weather checker. (Remember: Comment what it does, not how it functions)
if (isNight || isRaining || isWindy) break; //Stop here. No use checking
further.
startSunnyDayActivities(); //Will activate if weather is good.
```

METHOD 2. AVOIDING DUPLICATION AND REDUNANCY

Reasoning: If two things are doing the exact same thing and it's easy to combine them: It's usually good practice to do so.

Note: If two functions are similar but you plan to greatly expand the other: Do not. The extra-connections but vastly different usage will create you problems

BAD EXAMPLE. We can see logging is same if statement.

```
//Adjust Char's mood and scene based on weather and log it to dev console
if (isRaining) {
  context.character.scenario = "The weather is rainy and bleak outside";
  context.character.personality = "{{Char}} is sad";
} else {
  context.character.scenario = "The weather is sunny and warm outside!";
  context.character.personality = "{{Char}} is happy";
}

if (isRaining) {
  console.log("Weather is: Rainy");
} else {
  console.log("Weather is: Sunny");
}
```

GOOD EXAMPLE

```
// Adjust Char's mood and scene based on weather and log it to dev console
if (isRaining) {
  context.character.scenario = "The weather is rainy and bleak outside";
  context.character.personality = "{{Char}} is sad";
  console.log("Weather is: Rainy");
```

```
} else {
  context.character.scenario = "The weather is sunny and warm outside!";
  context.character.personality = "{{Char}} is happy";
  console.log("Weather is: Sunny");
}
```

Credits

This guide represents the collective knowledge and collaboration of the **JanitorAI scripting community**, curated and expanded by *Icehellionx*.

Every line of code shared, tested, and refined together adds to this body of knowledge.

Thank you for being part of it.

Written by: Icehellionx
Edited by: @AnnabelleCaprine