

p5.js Technical Writing Guide

Introduction

Hello! Thanks for writing documentation for p5.js. We're excited to learn from you and look forward to collaborating as you write. We've developed the following guidelines to help you write the most effective and accessible documentation possible.

There are five sections in this guide. The first section describes the [Accessible Writing Style](#) we use when writing documentation. The other four sections each explain how to write one type of documentation:

- [Tutorials](#) are lessons that take the reader by the hand through a series of steps to complete a project of some kind.
- [How-to guides](#) are directions that take the reader through the steps required to solve a specific problem.
- [References](#) are technical descriptions of the API and how to use it.
- [Explanations](#) are discussions that clarify and illuminate a particular topic.

Please read the [Accessible Writing Style](#) section along with the guide for the type of documentation you're creating. Follow the [Code Style Guide](#) in your code snippets.

Naming Conventions

Always refer to the p5.js library as “**p5.js**”, not “p5”, “P5.js”, “p5.JS”, “p5*js”, or any other variation. You may have a favorite nickname, but that's between you and p5.js. Consistency helps to avoid confusion.

Similarly, always refer to the p5.js Web Editor as the “p5.js Web Editor”, not the “p5 Editor”, “p5.js Editor”, or any other variation.

In general, spell names correctly: p5.js, CSS, HTML, JavaScript, WebGL. When in doubt, refer to an authoritative source like their official documentation.

Accessible Writing Style

The style for p5.js documentation reflects our purpose in publishing it: to provide quality learning resources for artists, designers, educators, beginners, and anyone else. We ensure that all documentation is:

- Beginner-friendly
- Accessible for screen reader
- Inclusive

- Accurate and up-to-date
- Practical, useful, and self-contained

Readability

Writing in plain language increases the accessibility of our documentation. This is especially true for young people, people with cognitive or intellectual disabilities, and people whose native language isn't English. Everyone benefits when we keep it simple. We emphasize a few guidelines from MDN's overview of [cognitive accessibility](#):

- provide easily-understood content, such as text written using plain-language standards
- focus attention on important content
- minimize distractions, such as unnecessary content
- divide processes into logical, essential steps with progress indicators

We also consider [Web Accessibility](#) guidelines with every piece of writing.

We recommend using a free tool called [Hemingway App](#) to help you gauge the lowest education level needed to understand your writing. Aim for a Readability score of Grade 8 or lower.

Here's an example of using Hemingway to simplify the p5.js Reference documentation:

The image shows a side-by-side comparison of the Hemingway App interface. The left panel shows the original text from the p5.js documentation, which has a readability score of Grade 4. The right panel shows the same text after being simplified by the app, resulting in a readability score of Grade 8. The simplified text uses shorter sentences, simpler vocabulary, and more direct phrasing. The app's interface includes a toolbar at the top with options like Bold, Italic, H1, H2, H3, Quote, Bullets, Numbers, Link, and a 'r' icon. The main text area is on the left, and the readability score and various writing tips are on the right.

Original Text (Left)	Simplified Text (Right)
Draws text to the canvas.	Draws text to the screen. Displays the information specified in the first parameter on the screen in the position specified by the additional parameters. A default font will be used unless a font is set with the <code>textFont()</code> function and a default size will be used unless a font is set with <code>textSize()</code> . Change the color of the text with the <code>fill()</code> function. Change the outline of the text with the <code>stroke()</code> and <code>strokeWeight()</code> functions.
The first parameter, <code>str</code> , is the text to <code>be drawn</code> . The second and third parameters, <code>x</code> and <code>y</code> , set the coordinates of the text's bottom-left corner. See <code>textAlign()</code> for other ways to align text.	The text displays in relation to the <code>textAlign()</code> function, which gives the option to draw to the left, right, and center of the coordinates.
The fourth and fifth parameters, <code>maxWidth</code> and <code>maxHeight</code> , are optional. They set the dimensions of the invisible rectangle containing the text. By default, they set its <code>maximum</code> width and height. See <code>rectMode()</code> for other ways to define the rectangular text box. Text will wrap to fit within the text box. Text outside of the box won't <code>be drawn</code> .	The <code>x2</code> and <code>y2</code> parameters define a rectangular area to display within and may only <code>be used</code> with string data. When these parameters <code>are specified</code> , they <code>are interpreted</code> based on the current <code>rectMode()</code> setting. Text that does not fit completely within the rectangle specified will not <code>be drawn</code> to the screen. If <code>x2</code> and <code>y2</code> are not specified, the baseline alignment is the default, which means that the text will <code>be drawn</code> upwards from <code>x</code> and <code>y</code> .
Text can <code>be styled</code> a few ways. Call the <code>fill()</code> function to set the text's fill color. Call <code>stroke()</code> and <code>strokeWeight()</code> to set the text's outline. Call <code>textSize()</code> and <code>textFont()</code> to set the text's size and font, <code>respectively</code> .	WEBGL: Only opentype/truetype fonts <code>are supported</code> . You must load a font using the <code>loadFont()</code> method (see the example above). <code>stroke()</code> currently has no effect in webgl mode. Learn more about working with text in webgl mode on the wiki .
Note: WEBGL mode only supports fonts loaded with <code>loadFont()</code> . Calling <code>stroke()</code> has no effect in WEBGL mode.	
Readability Grade 4 Good Words: 152	Readability Grade 8 Good Words: 216
<ul style="list-style-type: none"> 1 adverb, meeting the goal of 2 or fewer. 3 uses of passive voice, meeting the goal of 3 or fewer. 1 phrase has a simpler alternative. 0 of 17 sentences are hard to read. 0 of 17 sentences are very hard to read. 	<ul style="list-style-type: none"> 0 adverbs. Well done. 0 uses of passive voice. Cut to 3 or fewer. 1 phrase has a simpler alternative. 0 of 14 sentences are hard to read. 0 of 14 sentences is very hard to read.

Alt text: “A screenshot of two text editor windows side-by-side. The editor window on the left has less text and very few highlights. Its readability score is Grade 4. The editor window on the right has more text and is highlighted heavily. Its readability score is Grade 8.”

The original documentation on the right requires an eighth-grade education to read. The highlighted sections indicate the potential for improvement. The revision on the left requires a fourth-grade education to read.

Web Accessibility

You can make your writing more accessible by using a clear HTML structure, writing descriptive alternative text for images, and making thoughtful color choices. Consider the [accessibility principles](#) and standards developed by the [Web Accessibility Initiative](#); this section provides a few guidelines for each of these documentation features.

HTML gives web pages their structure. Use headings to give your writing a clear outline and lists to organize items. Keep the following guidelines in mind as you write.

- Your title should be the only Level 1 heading on the page. Use Level 2 headings for the main sections in your document. Only use Level 3 headings if they’re essential to keep a section organized.
- Don’t list items within a paragraph. Use lists to make it easy for readers to find and understand groups of information.
- Use the template for the type of documentation you’re writing as a starting point.

Alternative text (alt text)

Alt text makes images accessible to screen readers through concise (1–3 sentences) written descriptions. They convey the purpose of an image, including pictures, illustrations, charts, etc. Text alternatives are used by people who do not see the image.

When you write alt text, consider the image’s purpose. Most images you’ll use are either **informative** or **complex**. See [this resource for more](#) guidance.

- **Informative images** convey simple concepts or information. For example, an image of the canvas’ state or a button to press. Your alt text should focus on the image’s meaning, rather than its literal content.
- **Complex images** convey a large amount of information. For example, a diagram or illustration used to explain a concept. You should always explain such images in the surrounding text. Use alt text to summarize the image’s meaning. Add a caption to provide longer descriptions.
- Read W3C’s [alt Decision Tree](#) for more detailed suggestions based on your use case, & [W3C’s Images Tutorial](#) for even more guidance. Check out [Be My AI](#) to help you generate ideas for descriptions.

- GIFs should additionally include descriptions of any significant motion or animation. For example, when a user moves the mouse to click a button.

Color is perceived differently by everyone. People with various forms of color blindness, visual impairments, or cognitive difficulties may struggle to process color in various ways.

- **Contrast** is a balancing act, especially between text and its background. People with visual impairments can find it hard to read low-contrast text. People with certain types of dyslexia can find it hard to read high-contrast text.
- **Information** that's presented through color can be invisible to people with color blindness. For example, red error messages may blend in with the surrounding text. Don't rely on color alone to convey meaning.
- Download [Color Oracle](#) to test your visuals in grayscale.

English

Please use American English (color, center, modularize, and so on). See [a list of American and British English spelling differences here](#).

Oxford Comma

Please use the [Oxford comma](#) ("red, white, and blue", instead of "red, white and blue").

Grammar

Write simple sentences. Shorter is better: get to the point.

Imperative sentences provide instructions. "Write simple sentences." is an imperative sentence. You'll use them frequently in tutorials and how-to guides.

Declarative sentences make statements. They have a **subject** and a **predicate**. "Imperative sentences provide instructions." is a declarative sentence. Its subject is "imperative sentences" and its predicate is "provide instructions".

Write in the **present tense**: "Returns an object that...", rather than "Returned an object that...", or "Will return an object that...".

Write in the **active voice**: "The function draws a circle.", rather than "A circle is drawn by the function." Using the active voice helps with keeping things brief.


Start comments in upper case. Follow regular punctuation rules:


```
// Draws a fractal from a Julia set.
function drawFractal(c, radius, maxIter) {
  // ...
```

}

Tone

Our documentation aims for a friendly but formal tone. This means we don't include jargon, memes, excessive slang, emoji, or jokes. We're writing for a global audience, so we aim for a tone that works across language and cultural boundaries. Consider the following example:

"The statement `while (true) {}` creates an infinite loop, just like in the hit 1993 film *Groundhog Day* .

The reference doesn't translate easily across generations or cultures. The emoji is also a beaver, not a groundhog .

Tutorials

TLDR:

- [Tutorials Template](#)
- **Example:** [Get Started](#)
- [Writing for Accessibility](#) resource
- [Readability resource](#)
- [Alt text](#) resource
- [Thumbnail](#) resource
- [Thumbnails Folder](#)

Tutorials are lessons that take the reader by the hand through a series of steps to complete a project of some kind. Tutorials are learning-oriented. They help readers acquire specific skills.

Our tutorials are as clear and detailed as possible. **We explicitly include every step a reader needs to go from a blank canvas to the final, working sketch.** We also provide readers with all of the explanations and background information they need to understand the tutorial.

Tutorials are written as close to [6th grade reading level](#) as possible.

All tutorials should help the reader build their creative confidence. When appropriate, model ways to problem solve. Teach the reader to debug, read documentation, search the web, and so on.

Check out the following resource & example tutorial for more guidance:

- [Web accessibility tips](#) and standards resource.
- [Writing for readability](#) resource
- **Example Tutorial:** [Get Started](#)

Structure

Tutorials have a consistent structure, which includes a title, an introduction, a conclusion, and any prerequisites necessary for a reader to get started:

- Thumbnail & Description
- Title (Level 1 heading)
 - Introduction (Paragraph):
 - 2- 5 sentences that describes what the reader will be creating
 - link to p5.js project example
 - main image with [alt text](#) (gif or screenshot of potential project outcome)
- Prerequisites (Level 2 heading):
 - Bulleted list of prerequisite skills and/or accompanying links for previous tutorials that cover them
- Step 1 — Doing the First Thing (Level 2 heading)
 - Provide code lines and completed code after each step
 - Code should contain comments with helpful descriptions about chunks of code that are relevant
 - Explanation and motivation for code presented should follow an example of the users finished code after the step
 - For example: After giving instructions, you can include “Your code should look like this:” followed by example code from the code editor
 - After the example, explain the motivation and technical/coding concepts used in that step
 - (Optional) Include a “Try It!” section where users can customize their projects further before the next step - be sure to include the solution in a link.
- ...
- Step n — Doing the Last Thing (Level 2 heading)
- Conclusion (Level 2 heading): Try to spark interest for next steps by providing suggestions, challenges, examples to explore, etc
- Next Steps (Level 2 heading): link to next tutorial
- Resources (Level 2 heading): bullet list of all resources referenced in tutorial. If not from the [p5.js reference](#) try to use references from:
 - [MDN Docs](#)
 - [Coding Train](#)
 - [Code Academy](#)

Thumbnail & Description

Thumbnail: Add links for two versions of a thumbnail with alt text after saving them in the [Thumbnails folder](#).

- A thumbnail is easy to read, and represents the overall content idea of the tutorial. By looking at it, viewers can get a quick idea of what your tutorial is about and decide whether to click it or just move on (visit this resource for [examples of thumbnail & description](#)).

- Save thumbnails as **jpg or png** in the following formats:
 - **1500x1000 pixels at 144dpi**
- Use the following naming conventions: (replace “titleWord” with a word that helps identify the associated tutorial)
 - titleWordA
 - titleWordB
- You can use [Preview on mac](#), [MS Paint on PC](#), [Image Sizer on Adobe Express](#) or any other image processing app you feel comfortable with.
 - Read [How to find a perfect thumbnail](#) for more suggestions

Description: Include a description of your tutorial in 1 - 2 sentences. It should give the reader an idea beyond the title of what they will be doing, or what they will get out of each tutorial (visit this resource for [examples of thumbnail & description](#)).

Template

Our [tutorial template](#) has this structure already written for you. We encourage you to use the template as a starting point for your own tutorials.

Writing an introduction

The first section of every tutorial is the **Introduction**, which is usually 2 - 5 sentences long. The purpose of the introduction is to motivate the reader, set expectations, and summarize what the reader will do in the tutorial. Your introduction should answer the following questions:

- **What is the tutorial about?** What functions are involved and what concepts will be covered?
- **Why should the reader learn this topic?** What are the benefits of using these particular tools? What are some practical reasons why the reader should follow this tutorial?
- **What will the reader do or create in this tutorial?** Are they learning about color? Are they designing their first classes? Be specific, as this gets readers excited about the topic.
- **What will the reader have accomplished when they're done?** What new skills will they have? What will they be able to do that they couldn't do before?

Answering these questions in your introduction will also help you design a clear and reader-focused tutorial. **Don't forget to include a main image with alt text and link of the finished project.**

Specifying prerequisites

The **Prerequisites** section of a tutorial spells out exactly what the reader should know before they follow the current tutorial. The format is a list that the reader can use as a checklist. Each item must link to an existing tutorial that covers the necessary content.

For example, a tutorial that introduces loops may have the following prerequisites:

To complete this tutorial, you will need:

- An understanding of variables. Check out the [Variables and Change Tutorial](#) to strengthen these skills.
- An understanding of conditionals. Check out the [Conditionals and Interactivity](#) tutorial to strengthen these skills.

By reading the prerequisites, your reader knows exactly what they need to know before they start. There are no surprises.

You can also link to the p5.js Reference to provide additional information in the body of the tutorial. You should only send readers offsite to trusted resources such as MDN if the information can't be added to the tutorial directly in a short summary.

List of trusted resources:

- [MDN Docs](#)
- [Coding Train](#)
- [Code Academy](#)
- [Introduction to Computational Media](#) by cs4all
- [Geeks for Geeks: Introduction to p5.js](#)

Designing content

Once a reader has finished a tutorial, they will have created something from start to finish. We recommend starting at the finish when you plan your tutorial. Here's our remix of the [backward design](#) approach for designing tutorials:

- **Identify the learning goals.** What new skills should the reader take away from the tutorial?
- **Design the assessments.** How will the reader know they're making progress? Short coding prompts and self-check questions are most common.
- **Write the tutorial.** Seek feedback early and often. The writing team is here to help you.

You should test your tutorial to ensure it works. Follow it exactly as written and revise it as needed. Our stewards also test tutorials as part of the review process.

Every block of code should be followed by an explanation that describes what it does and why it works that way. When you ask the reader to modify a sketch, first explain what it does and why you're asking the reader to make those changes. These details give readers the information they need to grow their skills. Introduce key concepts and how they are applied to the code example. Link to the p5.js reference and trusted resources to learn more

Notes on style

We encourage the use of the second person (e.g., “You will simulate …”) to keep the focus on the reader and what they’ll accomplish. We also encourage motivational language focused on outcomes. For example, instead of “You will learn how to simulate motion,” try “In this tutorial, you will simulate motion.” This approach motivates the reader and focuses on the goal they need to accomplish.

How-To Guides

How-to guides are directions that take the reader through the steps required to solve a specific problem. How-to guides are goal-oriented. They are recipes for accomplishing common tasks.

Our how-to guides offer practical solutions to common problems. We assume the reader already has the knowledge and skills needed to follow each step. Guides should be general enough that readers can adapt them to their specific use cases.

Example: [How to label your p5.js code](#)

Structure

How-to guides have a consistent structure, which includes a title, an introduction, a conclusion, and any prerequisites necessary for a reader to get started:

- Title (Level 1 heading)
 - Introduction (Paragraph)
- Prerequisites (Level 2 heading)
- Step 1 — Doing the First Thing (Level 2 heading)
- Step 2 — Doing the Next Thing (Level 2 heading)
- ...
- Step n — Doing the Last Thing (Level 2 heading)
- Conclusion (Level 2 heading)

Our [how-to guide template](#) has this structure already written for you. We encourage you to use the template as a starting point for your own guides.

Writing an introduction

The first section of every how-to guide is the **Introduction**, which is usually one paragraph long. The purpose of the introduction is to motivate the reader, set expectations, and summarize what the reader will do in the guide. Your introduction should answer the following questions:

- **What will the reader have accomplished when they’re done?** What task will they have completed? What problem will they now be able to solve?

- **Why should the reader learn this topic?** What are the benefits of using these particular tools? What are some practical reasons why the reader should follow this guide?
- **What will the reader do or create in this guide?** What case study will be used for demonstration? Be specific, as this gets readers excited about the topic.

Answering these questions in your introduction will also help you design a clear and reader-focused how-to guide.

Specifying prerequisites

The **Prerequisites** section of a tutorial spells out exactly what the reader should know before they follow the current tutorial. The format is a list that the reader can use as a checklist. Each item must link to an existing tutorial that covers the necessary content.

For example, a tutorial that introduces loops may have the following prerequisites:

To complete this tutorial, you will need:

- An understanding of variables. Check out the [Variables and Change Tutorial](#) to strengthen these skills.
- An understanding of conditionals. Check out the [Conditionals and Interactivity](#) tutorial to strengthen these skills.

For example, a how-to guide that demonstrates using custom fonts on the p5 Editor may have the following prerequisites:

To complete this guide, you will need:

- An account on the p5 Editor. Check out [Setting Up Your Environment](#) to get set up.

You can also link to the p5.js Reference to provide additional information in the body of the tutorial. You should only send readers offsite to trusted resources such as MDN if the information can't be added to the tutorial directly in a short summary.

List of trusted resources:

- [MDN Docs](#)
- [Coding Train](#)
- [Code Academy](#)
- [Introduction to Computational Media](#) by cs4all
- [Geeks for Geeks: Introduction to p5.js](#)

By reading the prerequisites, your reader knows exactly what they need to do before they start. There are no surprises.

Designing content

Once a reader has finished a how-to guide, they will have solved a common problem. You should test your how-to guide to ensure it works for a couple of likely scenarios. Determine whether you need to add a fork or branch to cover another scenario. For example, is the reader getting started with the p5.js Editor or with VSCode? Our stewards also test how-to guides as part of the review process.

Every block of code should be followed by an explanation that describes what it does and why it works that way. When you ask the reader to modify a sketch, first explain what it does and why you're asking the reader to make those changes. These details give readers the information they need to grow their skills. Introduce key concepts and how they are applied to the code example. Link to the p5.js reference and trusted resources to learn more

Notes on style

We encourage the use of the second person (e.g., “You will upload ...”) to keep the focus on the reader and what they'll accomplish. We also encourage motivational language focused on outcomes. For example, instead of “You will learn how to upload custom fonts,” try “In this guide, you will upload custom fonts.” This approach motivates the reader and focuses on the goal they need to accomplish.

References

References are technical descriptions of the API and how to use it. Reference material is information-oriented. It's an authoritative resource that readers consult as needed.

Our references are straightforward and to the point. We cover all of the essential details in prose that's technically accurate and precise. We also provide readers with focused code snippets that demonstrate common use cases.

Examples: [fill\(\)](#) and [Flocking](#)

Structure

References have a consistent structure, which includes a title, a description, and one or more code snippets:

- Title (Level 1 heading)
- Description (Level 2 heading)
- Examples (Level 2 heading)

Our API reference is generated automatically from p5.js' inline documentation. Here's our guide for [adding inline documentation](#).

Notes on style

Prefer wordings that avoid "you"s and "your"s. For example, instead of:

If you need to declare a variable, it is recommended that you use `let`.

use this style:

Always use `let` to declare variables.

You may need to write in the **passive voice** to avoid “you” in references.

Code snippets should follow the example code guidelines. Choose meaningful code snippets that cover the basics as well as gotchas. Only use advanced syntax if it’s necessary to explain how a feature works. Don’t draw five circles to explain something when one circle will convey the idea.

Explanations

Explanations are discussions that clarify and illuminate a particular topic. Explanations are understanding-oriented. They weave webs of understanding by connecting things.

Our explanations help readers understand why things are the way they are. We provide context on the history, design choices, technical tradeoffs, and other considerations that brought p5.js to its current form. We tell readers a well-researched story that connects them to the makers who came before them.

Example: [Looking inside p5](#)

Structure

Explanations have a consistent structure, which includes a title, an introduction, and a conclusion:

- Title (Level 1 heading)
 - Introduction (Paragraph)
- Subtopic 1 (Level 2 heading)
- Subtopic 2 (Level 2 heading)
- ...
- Subtopic n (Level 2 heading)
- Conclusion (Level 2 heading)

Our [explanation template](#) has this structure already written for you. We encourage you to use the template as a starting point for your own explanations.

Writing an introduction

The first section of every explanation is the **Introduction**, which is usually one paragraph long. The purpose of the introduction is to motivate the reader, set expectations, and summarize the topic the reader will explore. Your introduction should answer the following questions:

- **Why should the reader learn about this topic?** What are the benefits of using these particular tools? What is unique, interesting, or unexpected about their design?
- **How does this topic relate to others?** How does the tool fit into the bigger picture of coding with p5.js? How will the reader's perspective change from a deeper understanding?

Answering these questions in your introduction will also help you design a clear and reader-focused explanation.

Designing content

Once a reader has finished an explanation, they will have a clearer view of how the pieces fit together. We recommend starting at the finish when you plan your explanation. Here's our remix of the [backward design](#) approach for designing explanations:

- **Identify the learning goals.** What new knowledge or big ideas should the reader take away from the explanation?
- **Design the assessments.** How will the reader know they're making progress? Short writing prompts and self-check questions are most common.
- **Write the explanation.** Seek feedback early and often. The community is here to help you.

You should thoroughly research your explanation to ensure it's accurate and comprehensive. Our stewards are happy to share their experience during the review process.

Notes on style

We encourage the use of the second person (e.g., "You may notice ...") to keep the focus on the reader and where they fit in the story. Feel free to offer opinions, but make sure you consider alternative points of view.

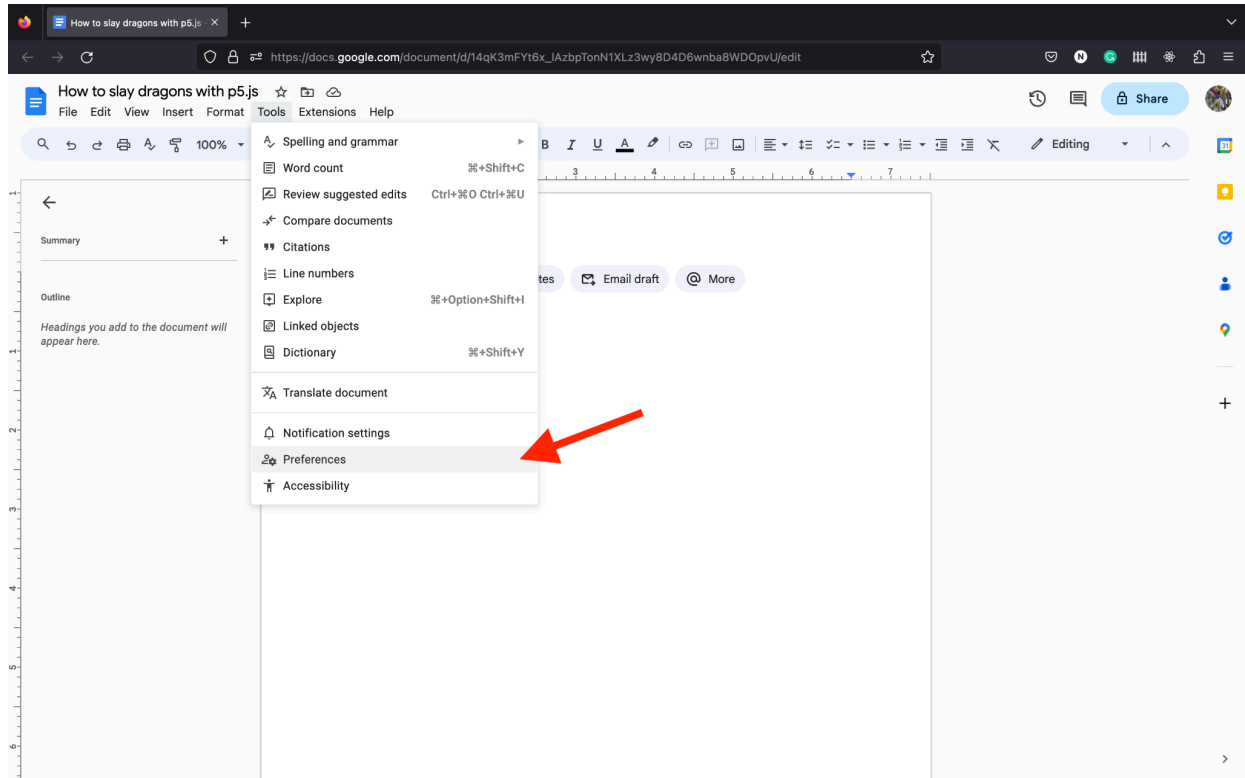
Getting Started

When you create a new document in Google Drive, ensure that you enable three helpful features: Markdown support and code formatting. The screenshots below detail how to enable them.

Step 0 – Go Pageless

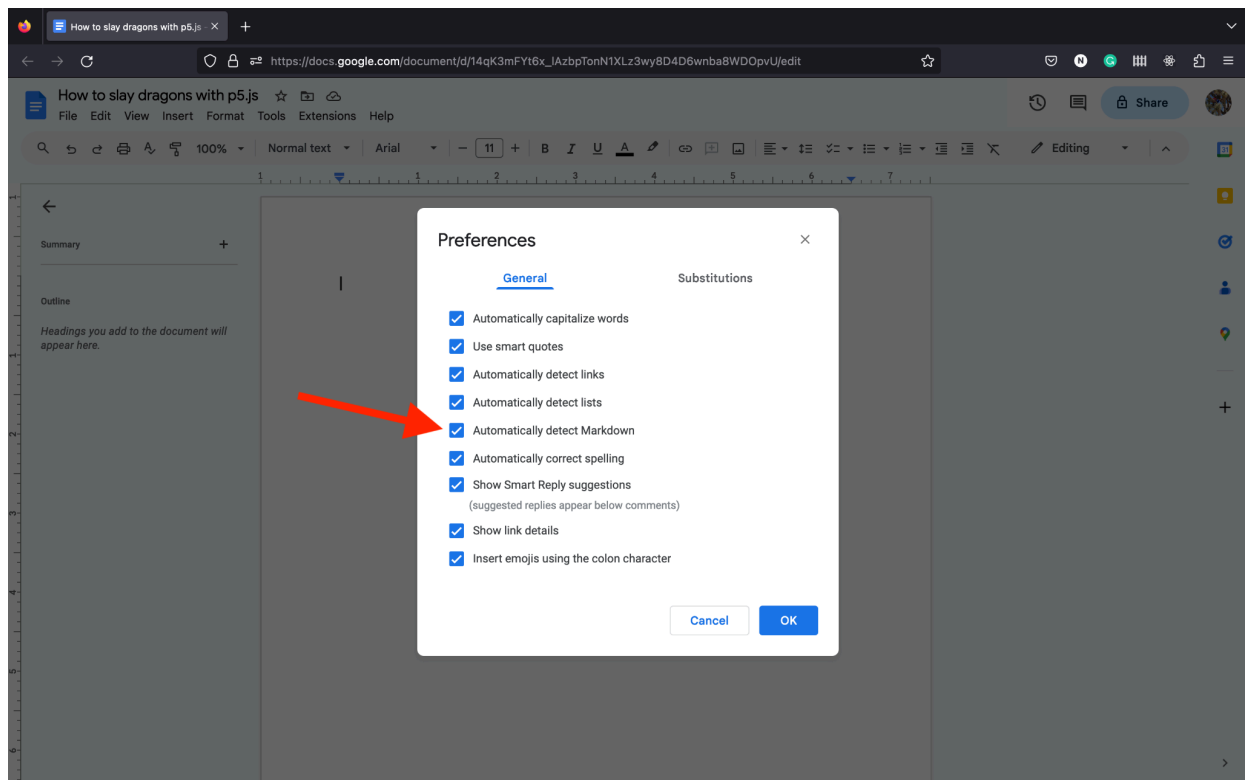
[Change a document's page setup: pages or pageless - Computer - Google Docs Editors Help](https://support.google.com/docs/answer/9299702?hl=en)

Step 1 – Open “Tools” > “Preferences”



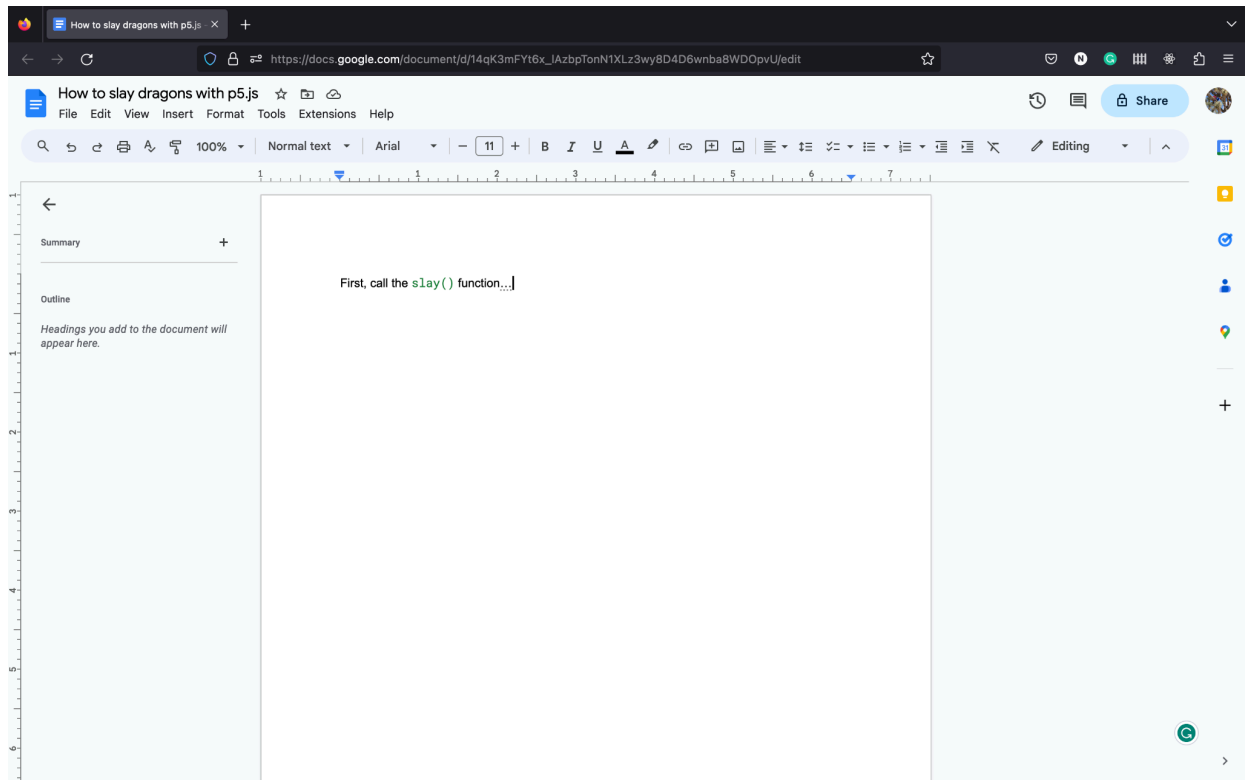
Alt text: A word processor with a dropdown menu open. A red arrow points to the “Preferences” menu option.

Step 2 – Make sure “Automatically detect Markdown” is checked



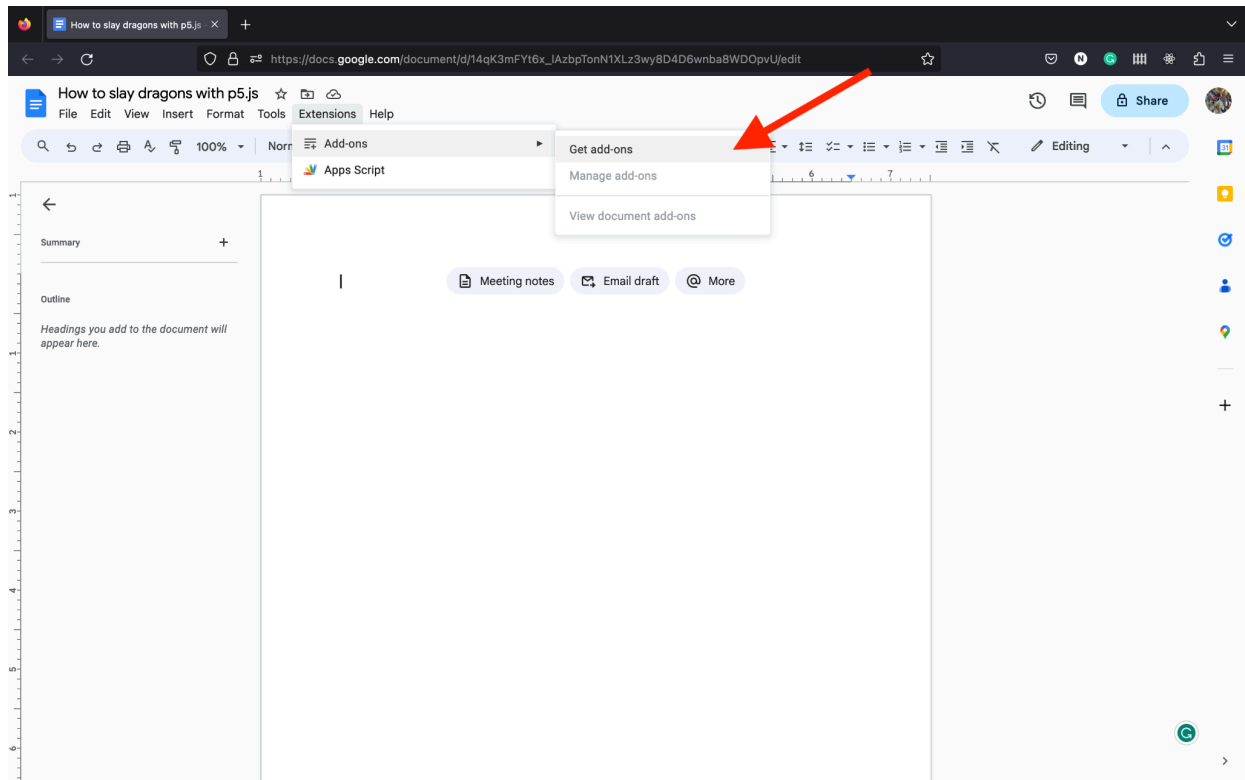
Alt text: A word processor with a modal open. The modal lists several preferences with checkboxes next to them. A red arrow points to the “Automatically detect Markdown” preference.

Step 3 – Use a pair of backticks `` to enclose inline code.



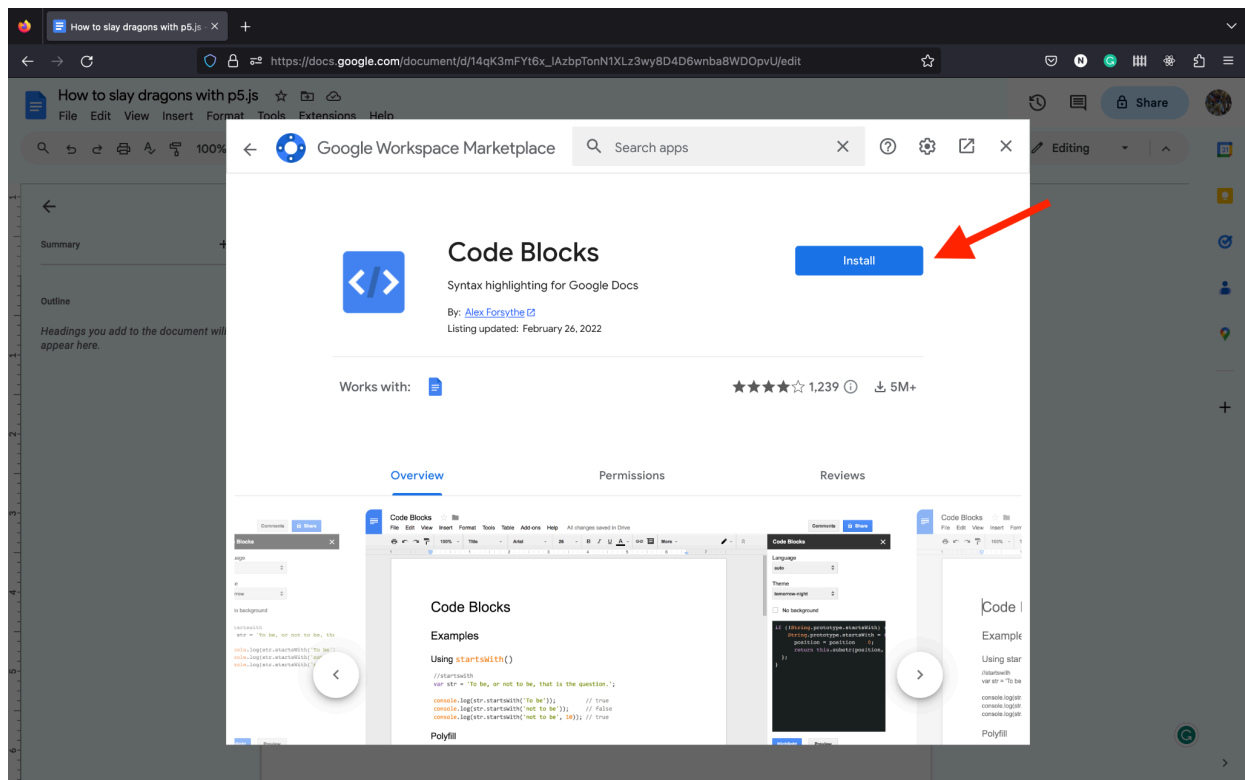
Alt text: A word processor with a line of text. The JavaScript function `slay()` is written in a green, monospace font.

Step 4 – Open “Extensions” > “Add-ons” > “Get add-ons”



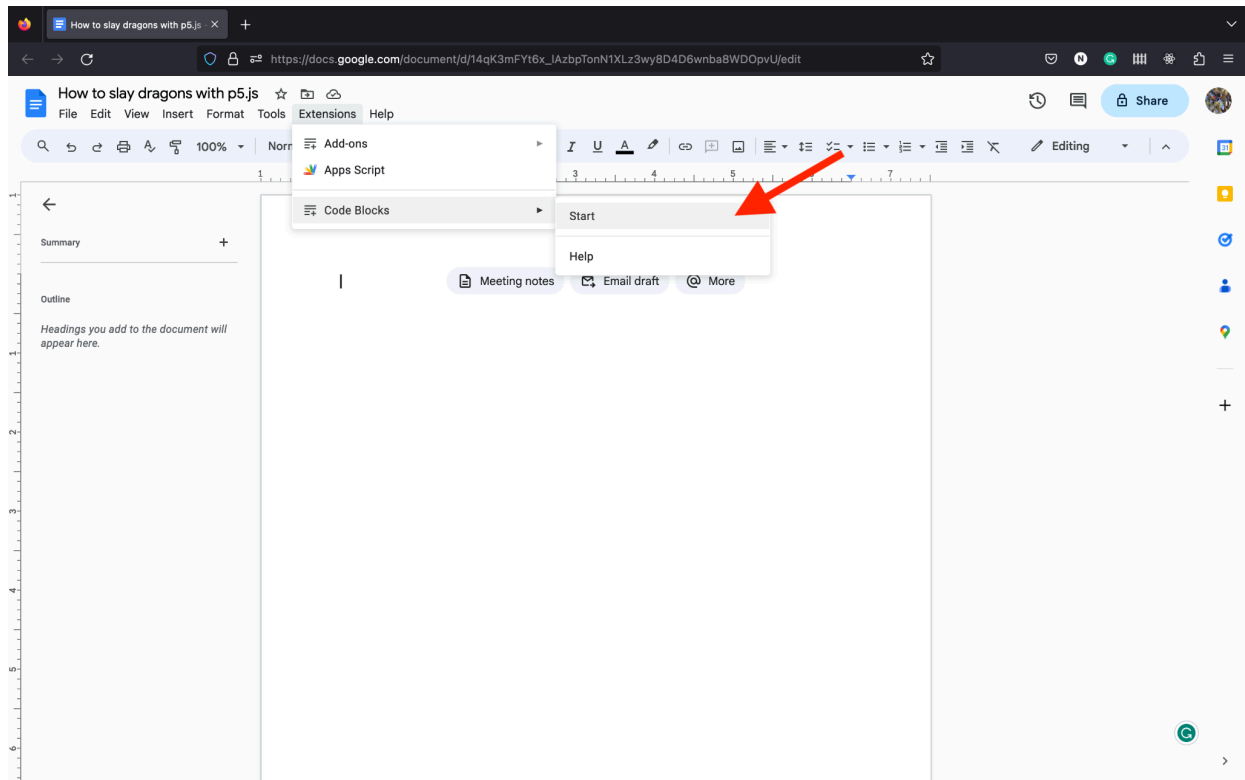
Alt text: A word processor with a dropdown menu open. A red arrow points to the “Get add-ons” menu option.

Step 5 – Search for “Code Blocks” and install it.

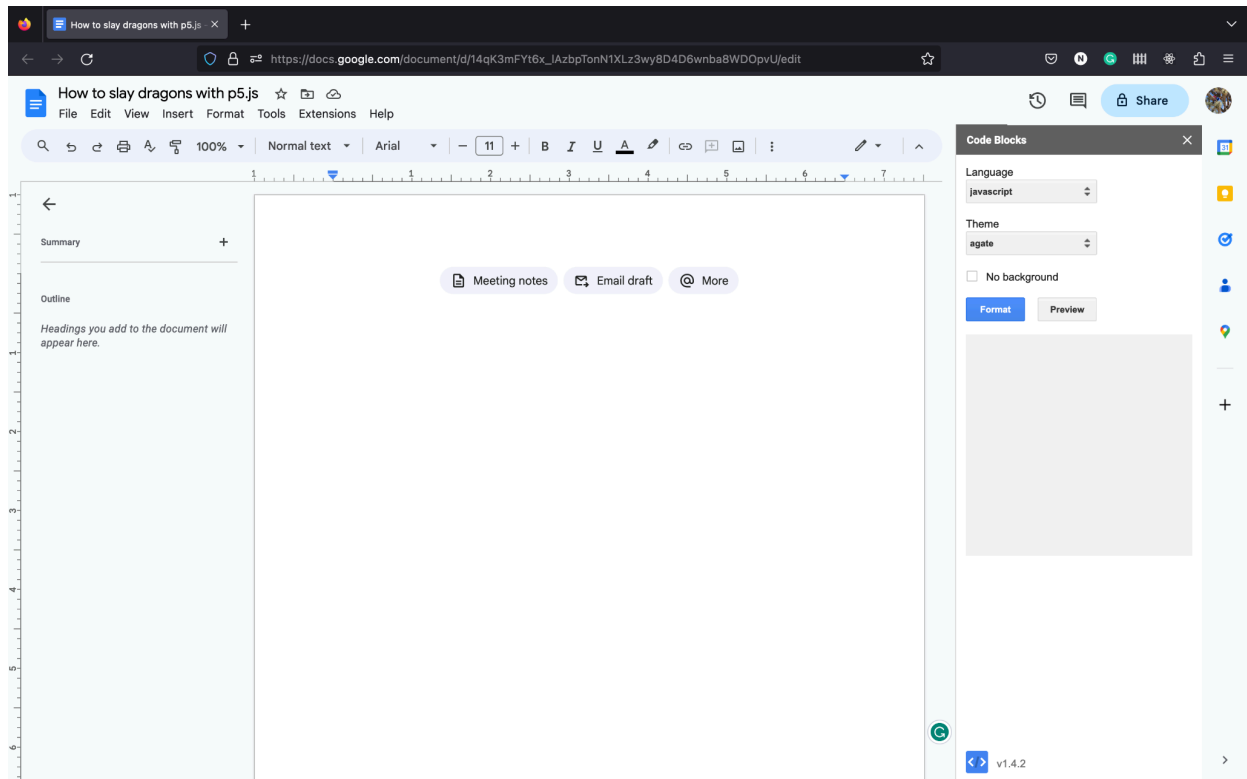


Alt text: A word processor with a modal open. The modal displays information about the Code Blocks extension. A red arrow points to the “Install” button.

Step 6 – Open “Extensions” > “Code Blocks” > “Start” to use the extension.

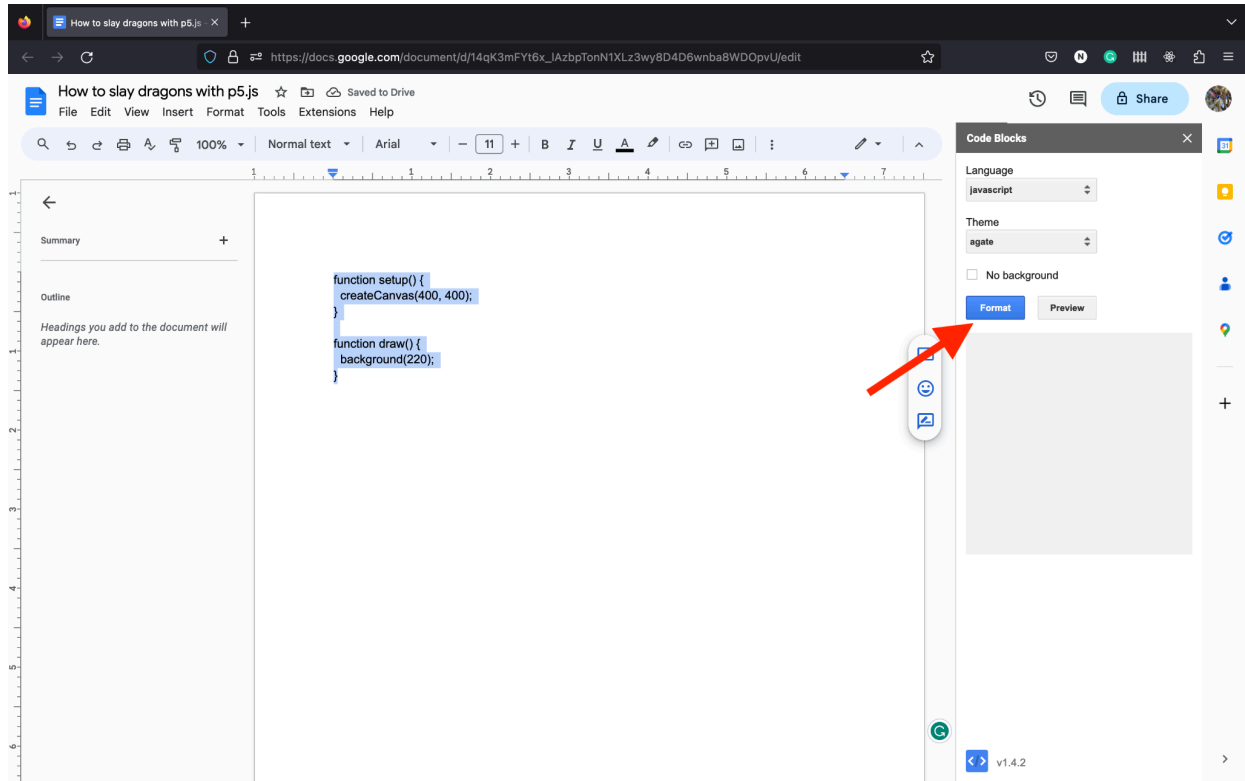


Alt text: A word processor with a dropdown menu open. A red arrow points to the “Start” menu option.

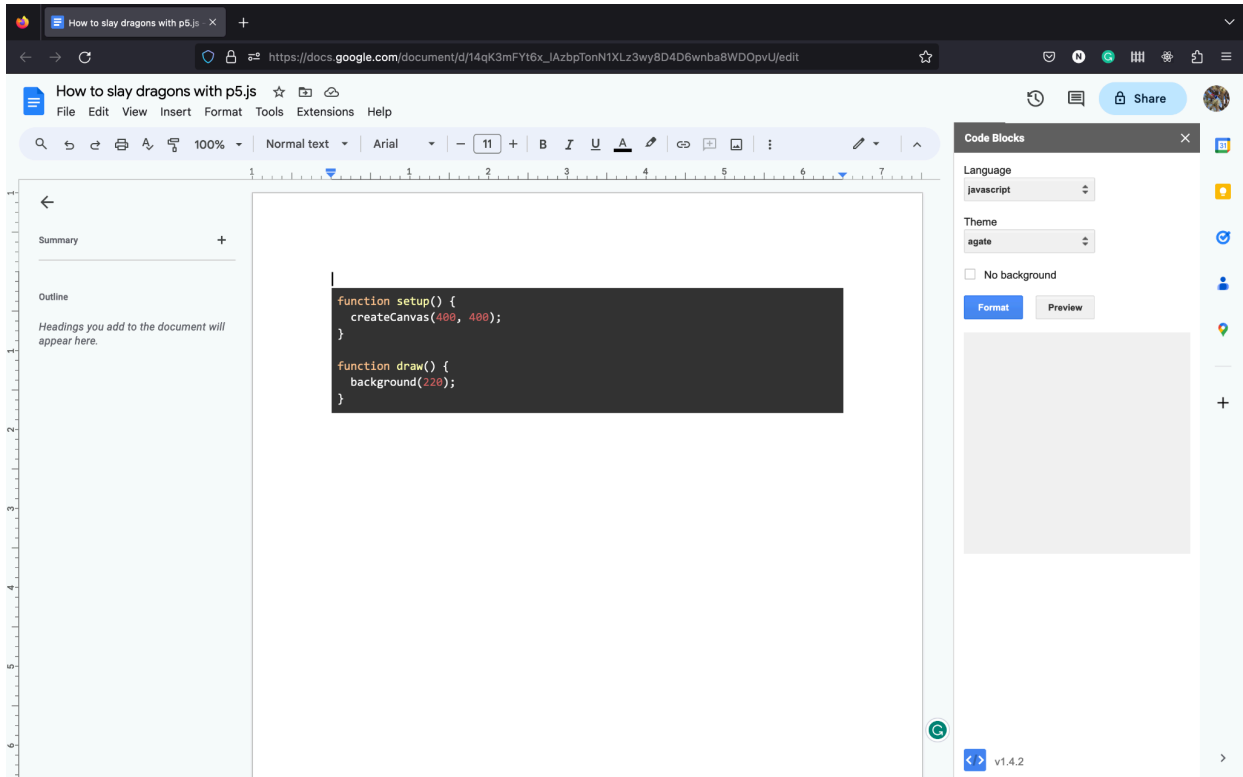


Alt text: A word processor with a side menu open.

Step 7 –Select the “javascript” language option, highlight your code snippet, and click the “Format” button.



Alt text: A word processor with unformatted code. A side menu is open with an arrow pointed at the “Format” button.



Alt text: A word processor with formatted code and an open side menu.

Sources

- [Digital Ocean](#) (CC BY-NC-SA 4.0)
- [Vue.js](#) (MIT)
- [Diátaxis](#) (CC BY-SA 4.0)
- [Ruby on Rails](#) (CC BY-SA 4.0)
- WordPress [Accessibility](#) and [Inclusivity](#) (CC0)
- [Write accessible documentation](#) (Google developer documentation style guide)
- [Creating Accessible Documents](#) (abilitynet.org.uk)
 - Writing with accessibility in mind means that you are trying to ensure that your content can be read and understood by as wide an audience as possible.
- [p5.js Community Statement](#) (CC BY-SA 4.0)
- [p5.js Access Statement](#) (?)
- [MDN cognitive accessibility reference](#) (CC BY-SA 2.5)