



**Child Mind
Institute**

Collaborative programming with GitHub

Jon Clucas

MoBI Retreat | Monday 26 February 2024

General advice

- Commit often.
- Keep the content of each commit as single-purpose as possible.
- Write clear commit messages describing the content of the commits.
- Develop separate features in separate branches. These branches can be cascaded or from the same commit-in-common.
- Review as many PRs as makes sense for you.
- Review the project's community files (e.g., CONTRIBUTING, README).
- Different projects have different needs at different times.
- If you have a suggestion, share your suggestion.
- If you have a question, ask.

Collaborative programming with GitHub

- **Why?**
- What?
- How?
- Danger?
- Help?



Code readability decreases with time and distance

The moment of **peak readability** is **the moment just after you write a line of code**.
Your code will be far less readable to you one day, one week, and one month after you've written it.

— Hunner, Trey. “Craft Your Python Like Poetry.” *Trey Hunner*, July 23, 2017. <https://treyhunner.com/2017/07/craft-your-python-like-poetry/>.

Scope is hard to manage

Scope grows like grass

Scope grows naturally. Scope creep isn't the fault of bad clients, bad managers, or bad programmers. Projects are opaque at the macro scale. You can't see all the little micro-details of a project until you get down into the work. Then you discover not only complexities you didn't anticipate, but all kinds of things that could be fixed or made better than they are.

Every project is full of scope we don't need. Every part of a product doesn't need to be equally prominent, equally fast, and equally polished. Every use case isn't equally common, equally critical, or equally aligned with the market we're trying to sell to.

This is how it is. Rather than trying to stop scope from growing, give teams the tools, authority, and responsibility to constantly cut it down.

— Singer, Ryan. “Scope Grows like Grass.” In *Shape Up: Stop Running in Circles and Ship Work That Matters*, 2024. <https://basecamp.com/shapeup/3.5-chapter-14#scope-grows-like-grass>.

Scope needs to be managed

Cutting scope isn't lowering quality

Picking and choosing which things to execute and how far to execute on them doesn't leave holes in the product. Making choices makes the product better. It makes the product better *at some things* instead of others. Being picky about scope *differentiates* the product. Differentiating what is core from what is peripheral moves us in competitive space, making us more alike or more different than other products that made different choices.

Variable scope is not about sacrificing quality. We are extremely picky about the quality of our code, our visual design, the copy in our interfaces, and the performance of our interactions. The trick is asking ourselves which things actually matter, which things move the needle, and which things make a difference for the core use cases we're trying to solve.

— Singer, Ryan. “Cutting scope isn't lowering quality.” In *Shape Up: Stop Running in Circles and Ship Work That Matters*, 2024. <https://basecamp.com/shapeup/3.5-chapter-14#cutting-scope-isnt-lowering-quality>.

Scope needs to be managed

Doug McIlroy, the inventor of Unix pipes and one of the founders of the Unix tradition, had this to say at the time [\[McIlroy78\]](#):

- (i) Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
- (ii) Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
- (iii) Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
- (iv) Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

He later summarized it this way (quoted in *A Quarter Century of Unix* [\[Salus\]](#)):

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

— Raymond, Eric Steven. “Basics of the Unix Philosophy.” In *The Art of Unix Programming*, Revision 1.0., 2003.

<http://www.catb.org/~esr/writings/taoup/html/ch01s06.html>.

Contributions need to be trustworthy

Trust through transparency

- The easiest way to have people trust your work is by transparently sharing it without request.
 - Have everything accessible where people would look for it
 - Don't make them ask for it, because most won't

Klinger, Andreas. “Managing People 🤖.” *Andreas Klinger*, February 5, 2022. <https://klinger.io/posts/managing-people-🤖>.

Why?



- Code readability decreases with time and distance.
- Scope is hard to manage.
- Scope needs to be managed.
- Contributions need to be trustworthy.

Collaborative programming with GitHub

- Why?
- **What?**
- How?
- Danger?
- Help?

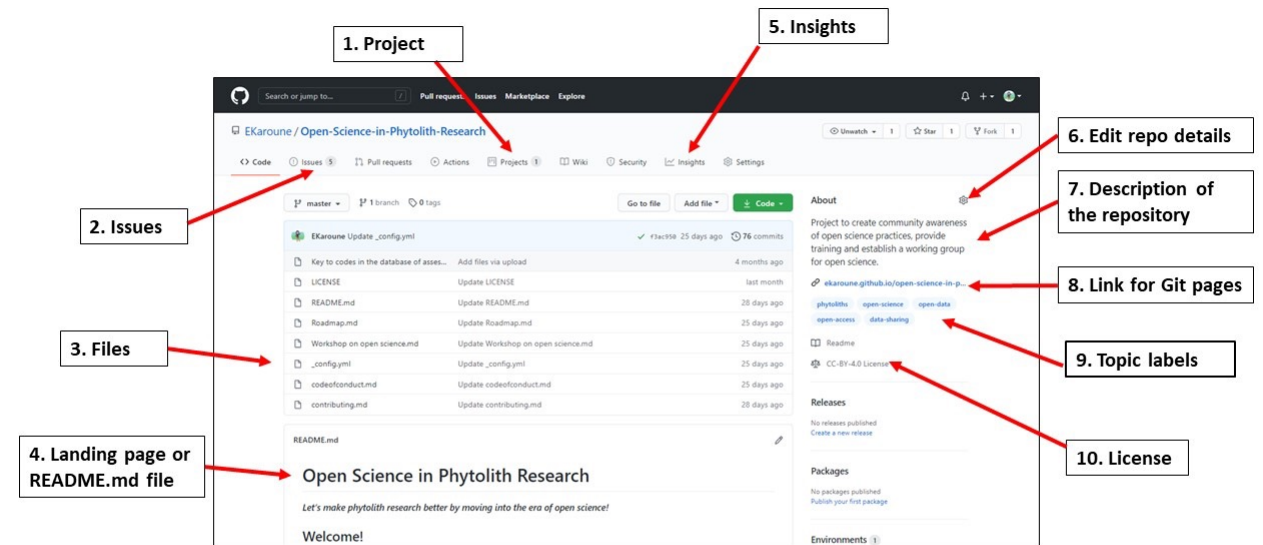


What's the difference between git and GitHub?



— Git. “Git.” <https://git-scm.com/>.

GitHub



— The Turing Way Community. “Using GitHub Features to Foster Collaboration.” In *The Turing Way*, 2024.

<https://the-turing-way.netlify.app/collaboration/github-novice/github-novice-features#using-github-features-to-foster-collaboration>.

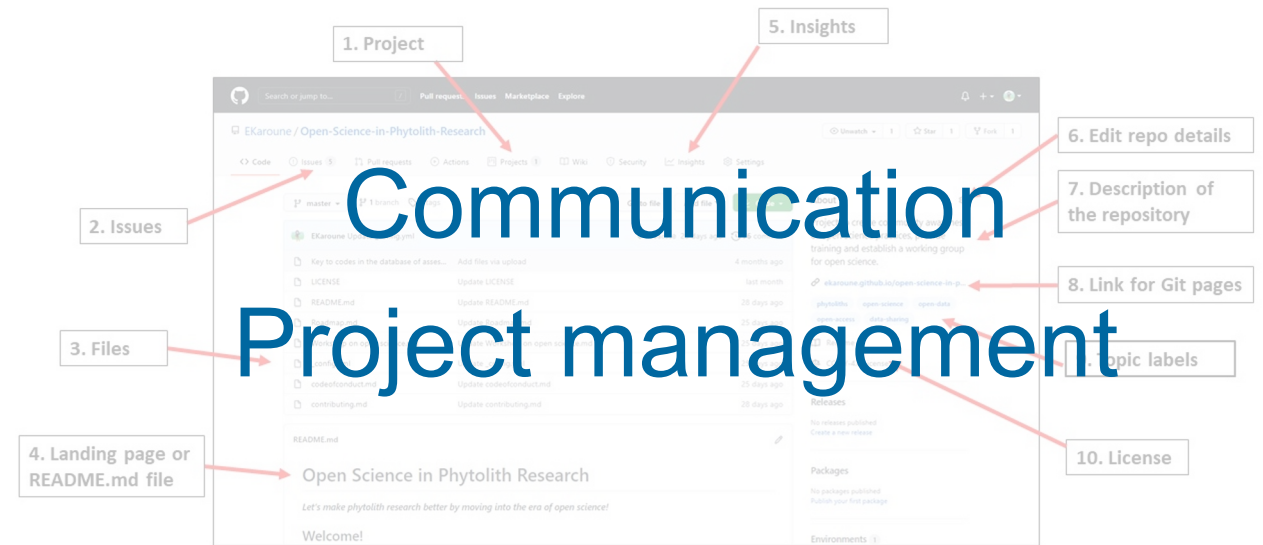
What's the difference between git and GitHub?



Version control

— Git. “Git.” <https://git-scm.com/>.

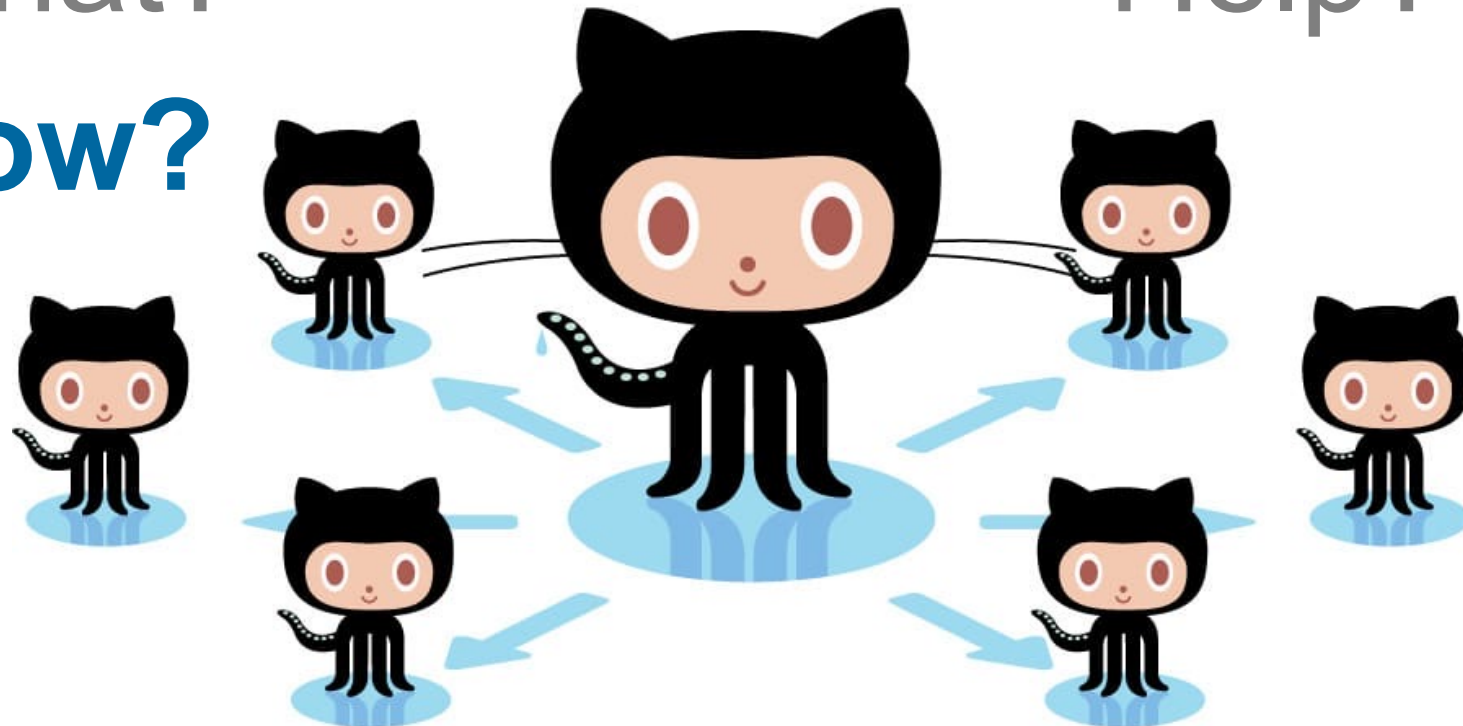
GitHub



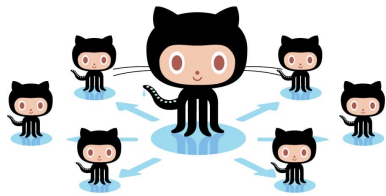
— The Turing Way Community. “Using GitHub Features to Foster Collaboration.” In *The Turing Way*, 2024.
<https://the-turing-way.netlify.app/collaboration/github-novice/github-novice-features#using-github-features-to-foster-collaboration>.

Collaborative programming with GitHub

- Why?
- What?
- **How?**
- Danger?
- Help?



How?



Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

— Peters, Tim. “PEP 20 – The Zen of Python.” Python Enhancement Proposals, August 22, 2004.

[https://peps.python.org/pep-0020/#the-zen-of-python.](https://peps.python.org/pep-0020/#the-zen-of-python)

To remove any confusion, here's a simple rule to get it right every time.

A properly formed Git commit subject line should always be able to complete the following sentence:

- If applied, this commit will *your subject line here*

For example:

- If applied, this commit will *refactor subsystem X for readability*
- If applied, this commit will *update getting started documentation*
- If applied, this commit will *remove deprecated methods*
- If applied, this commit will *release version 1.0.0*
- If applied, this commit will *merge pull request #123 from user/branch*

— Beams, Chris. “How to Write a Git Commit Message.” *cbeams*, August 31, 2014. <https://cbea.ms/git-commit/>.

Follow the project's existing style

Gitmoji is an emoji guide for GitHub commit messages. Aims to be a standarization cheatsheet - guide for using **emojis** on GitHub's commit messages.

Using emojis on commit messages provides an **easy way** of **identifying the purpose or intention of a commit** with only looking at the emojis used. As there are a lot of different emojis I found the need of creating a guide that can help to use emojis easier.

— Cuesta, Carlos. “About.”
gitmoji. <https://gitmoji.dev/about>.

The Conventional Commits specification is a lightweight convention on top of commit messages. It provides an easy set of rules for creating an explicit commit history; which makes it easier to write automated tools on top of. This convention dovetails with **SemVer**, by describing the features, fixes, and breaking changes made in commit messages.

The commit message should be structured as follows:

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

— Conventional Commits. “Conventional Commits 1.0.0 Summary.”
Conventional Commits.

<https://www.conventionalcommits.org/en/v1.0.0/#summary>.

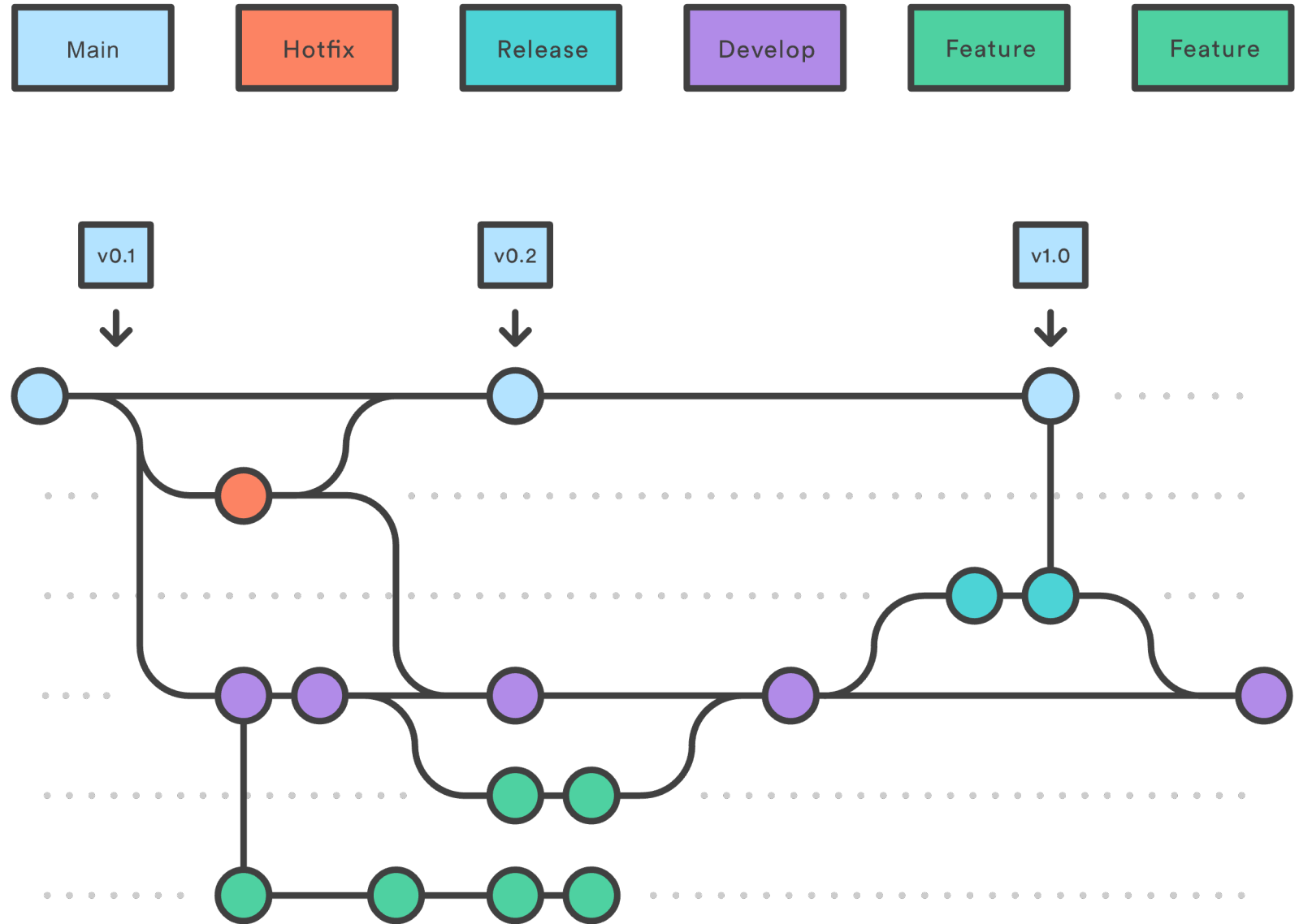
Follow the project's existing style

If your team is doing continuous delivery of software, I would suggest to adopt a much simpler workflow (like [GitHub flow](#)) instead of trying to shoehorn git-flow into your team.

If, however, you are building software that is explicitly versioned, or if you need to support multiple versions of your software in the wild, then git-flow may still be as good of a fit to your team as it has been to people in the last 10 years.

— Driessen, Vincent. “A Successful Git Branching Model: Note of Reflection.” *nvie.com*, March 5, 2020. <http://nvie.com/posts/a-successful-git-branching-model/>.

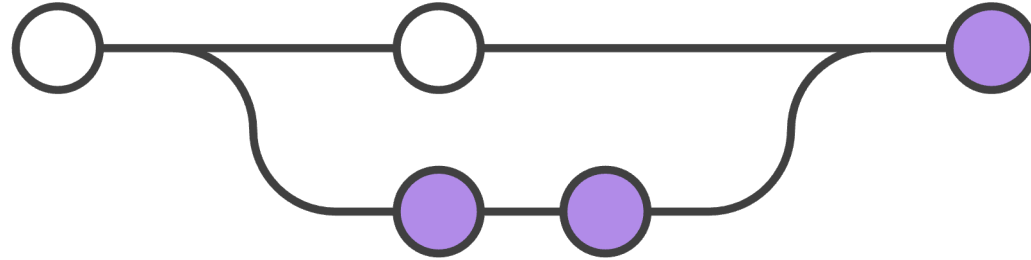
Gitflow



— Atlassian. “Gitflow Workflow.” Software Development, 2024.

<https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>.

GitHub Flow



— Atlassian. “Git Feature Branch Workflow.” Software Development, 2024.

<https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>.

Follow the project's existing style

A project also has documentation. These files are usually listed in the top level of a repository.

- **LICENSE:** By definition, every open source project must have an [open source license](#). If the project does not have a license, it is not open source.
- **README:** The README is the instruction manual that welcomes new community members to the project. It explains why the project is useful and how to get started.
- **CONTRIBUTING:** Whereas READMEs help people *use* the project, contributing docs help people *contribute* to the project. It explains what types of contributions are needed and how the process works. While not every project has a CONTRIBUTING file, its presence signals that this is a welcoming project to contribute to. A good example of an effective Contributing Guide would be the one from [Codecademy's Docs repository](#).
- **CODE_OF_CONDUCT:** The code of conduct sets ground rules for participants' behavior associated and helps to facilitate a friendly, welcoming environment. While not every project has a CODE_OF_CONDUCT file, its presence signals that this is a welcoming project to contribute to.
- **Other documentation:** There might be additional documentation, such as tutorials, walkthroughs, or governance policies, especially on bigger projects like [Astro Docs](#).

— Open Source Guides. “Anatomy of an Open Source Project.” How to Contribute to Open Source, February 8, 2024. <https://opensource.guide/how-to-contribute/#anatomy-of-an-open-source-project>.

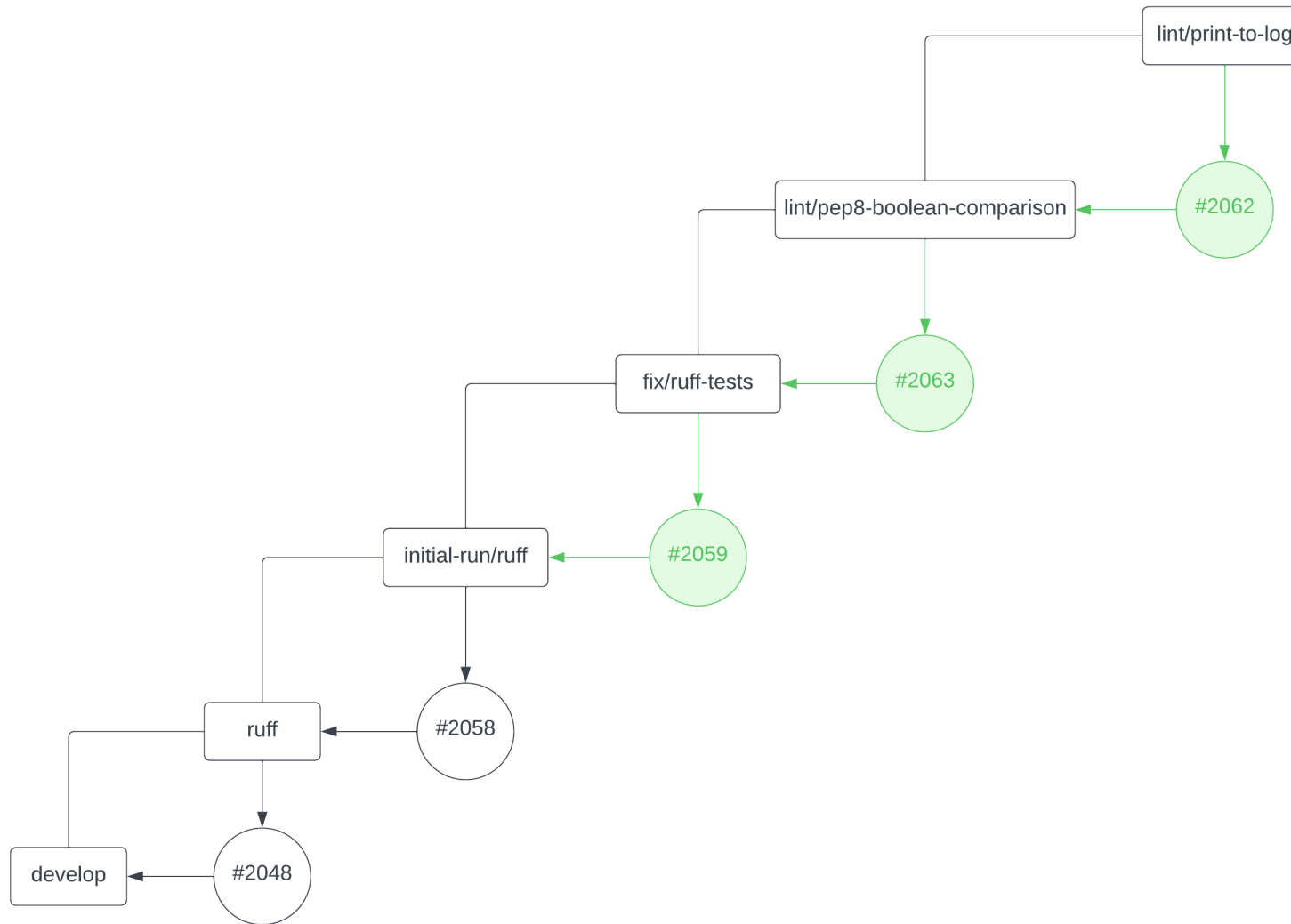
Follow the project's existing style

Finally, open source projects use the following tools to organize discussion. Reading through the archives will give you a good picture of how the community thinks and works.

- **Issue tracker:** Where people discuss issues related to the project.
- **Pull requests:** Where people discuss and review changes that are in progress whether its to improve a contributor's line of code, grammar usage, use of images, etc. Some projects, such as [MDN Web Docs](#), use certain GitHub Action flows to automate and quicken their code reviews.
- **Discussion forums or mailing lists:** Some projects may use these channels for conversational topics (for example, *"How do I..."* or *"What do you think about..."* instead of bug reports or feature requests). Others use the issue tracker for all conversations. A good example of this would be [CHAOSS' weekly Newsletter](#)
- **Synchronous chat channel:** Some projects use chat channels (such as Slack or IRC) for casual conversation, collaboration, and quick exchanges. A good example of this would be [EddieHub's Discord community](#).

— Open Source Guides. "Anatomy of an Open Source Project." How to Contribute to Open Source, February 8, 2024. <https://opensource.guide/how-to-contribute/#anatomy-of-an-open-source-project>.

example cascade of pull requests for related feature branches



Collaborative programming with GitHub

- Why?
- What?
- How?
- **Danger?**
- Help?



What if I mess something up?

- As long as what was working was committed,



- Git makes permanent irreversible changes hard to do by design.
- GitHub has features like "branch protection rules" to add further friction to publishing unintentional changes.

What if I *need to* permanently delete a mistake?

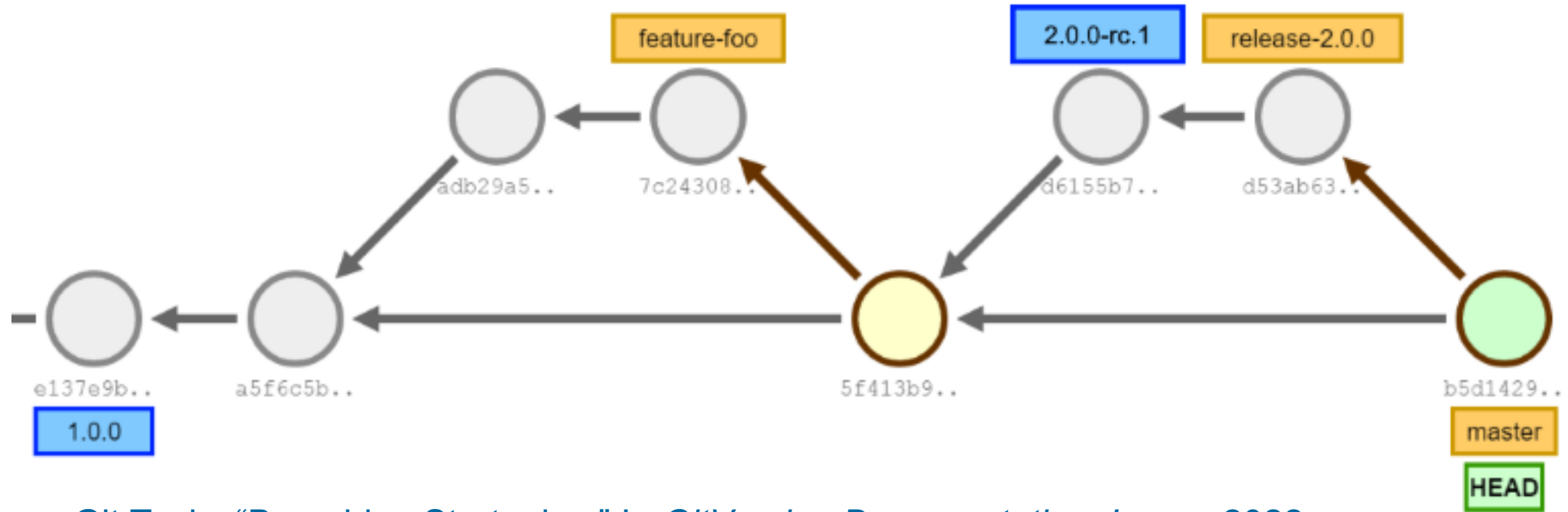
Removing sensitive data from a repository

If you commit sensitive data, such as a password or SSH key into a Git repository, you can remove it from the history. To entirely remove unwanted files from a repository's history you can use either the `git filter-repo` tool or the BFG Repo-Cleaner open source tool.

— GitHub, Inc. “Removing Sensitive Data from a Repository.” GitHub Docs.

<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/removing-sensitive-data-from-a-repository>.

SHAs (grey) and branches (yellow) and tags (blue) ...

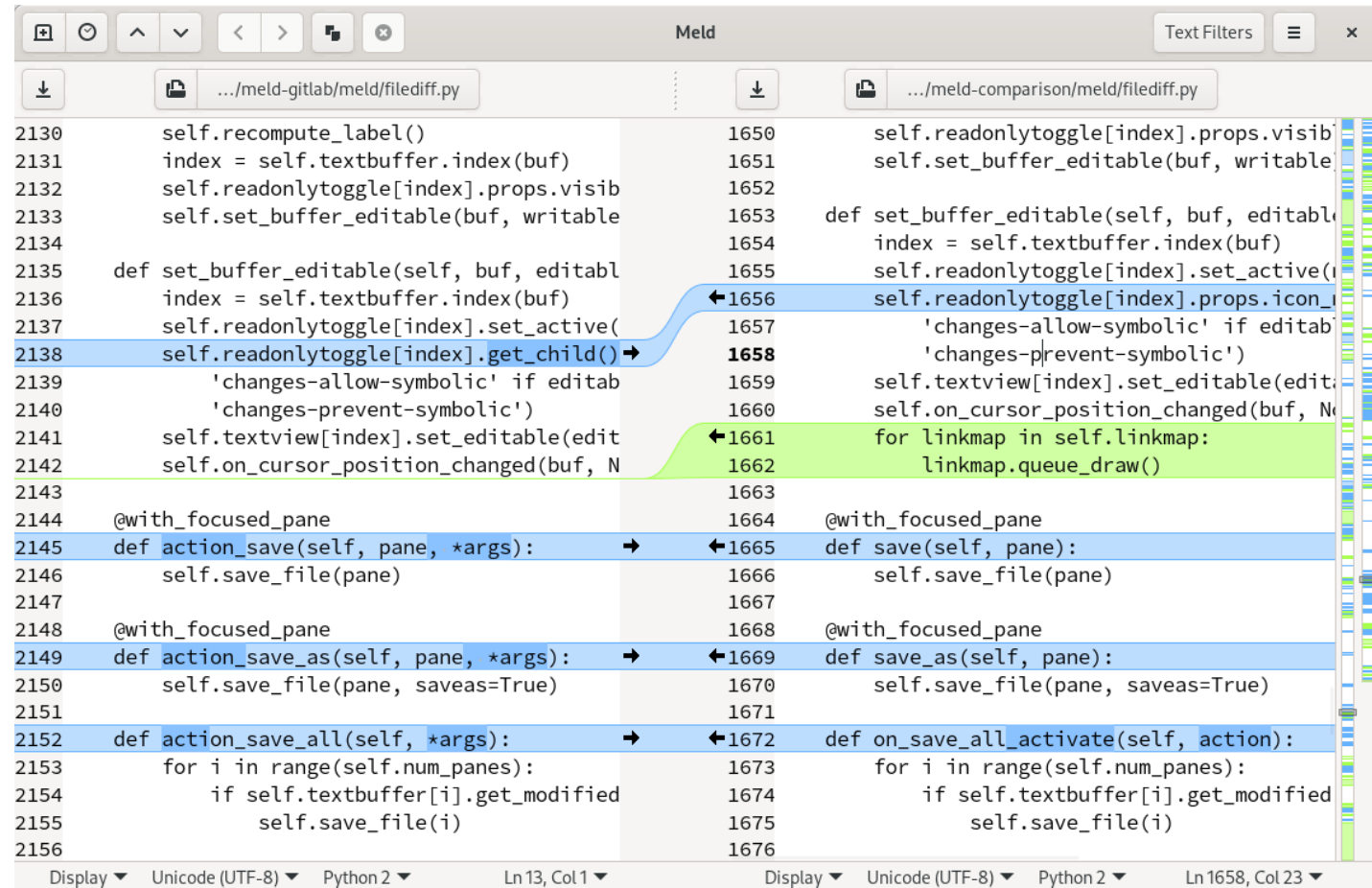


— Git Tools. “Branching Strategies.” In *GitVersion Documentation: Learn*, 2022.
<https://gitversion.net/docs/learn/branching-strategies/>.

Note: the arrows here and in the Git Pro book point to sources, reverse-chronologically.

This merge conflict resolution is a nightmare.

- Merge conflict resolutions are absolutely necessary.
- Conflicts arise when parallel changes are made to the same code and git can't tell how those changes should interact or affect each other.
- Find a tool that works for you.
- I adore Meld (<https://meldmerge.org>)



The screenshot shows the Meld merge conflict resolution interface. It displays two versions of the file `filediff.py` side-by-side. The left pane shows the version from `.../meld-gitlab/meld/filediff.py` and the right pane shows the version from `.../meld-comparison/meld/filediff.py`. The code is Python 2, UTF-8 encoded. The interface highlights several conflict points with blue and green background colors and arrows indicating the conflicting changes. The conflicts are as follows:

- Line 2138 (left) vs Line 1656 (right): `self.readonlytoggle[index].get_child()` vs `self.readonlytoggle[index].props.icon`
- Line 2139 (left) vs Line 1657 (right): `'changes-allow-symbolic' if editab` vs `'changes-allow-symbolic' if editab`
- Line 2140 (left) vs Line 1658 (right): `'changes-prevent-symbolic')` vs `'changes-prevent-symbolic')`
- Line 2141 (left) vs Line 1659 (right): `self.textview[index].set_editable(edit` vs `self.textview[index].set_editable(edit`
- Line 2142 (left) vs Line 1660 (right): `self.on_cursor_position_changed(buf, N` vs `self.on_cursor_position_changed(buf, N`
- Line 2143 (left) vs Line 1661 (right): `for linkmap in self.linkmap:` vs `for linkmap in self.linkmap:`
- Line 2144 (left) vs Line 1662 (right): `linkmap.queue_draw()` vs `linkmap.queue_draw()`
- Line 2145 (left) vs Line 1663 (right): `@with_focused_pane` vs `@with_focused_pane`
- Line 2146 (left) vs Line 1664 (right): `def action_save(self, pane, *args):` vs `def save(self, pane):`
- Line 2147 (left) vs Line 1665 (right): `self.save_file(pane)` vs `self.save_file(pane)`
- Line 2148 (left) vs Line 1666 (right): `@with_focused_pane` vs `@with_focused_pane`
- Line 2149 (left) vs Line 1667 (right): `def action_save_as(self, pane, *args):` vs `def save_as(self, pane):`
- Line 2150 (left) vs Line 1668 (right): `self.save_file(pane, saveas=True)` vs `self.save_file(pane, saveas=True)`
- Line 2151 (left) vs Line 1669 (right): `def action_save_all(self, *args):` vs `def on_save_all_activate(self, action):`
- Line 2152 (left) vs Line 1670 (right): `for i in range(self.num_panes):` vs `for i in range(self.num_panes):`
- Line 2153 (left) vs Line 1671 (right): `if self.textbuffer[i].get_modified` vs `if self.textbuffer[i].get_modified`
- Line 2154 (left) vs Line 1672 (right): `self.save_file(i)` vs `self.save_file(i)`
- Line 2155 (left) vs Line 1673 (right): `self.save_file(i)` vs `self.save_file(i)`
- Line 2156 (left) vs Line 1674 (right): `self.save_file(i)` vs `self.save_file(i)`

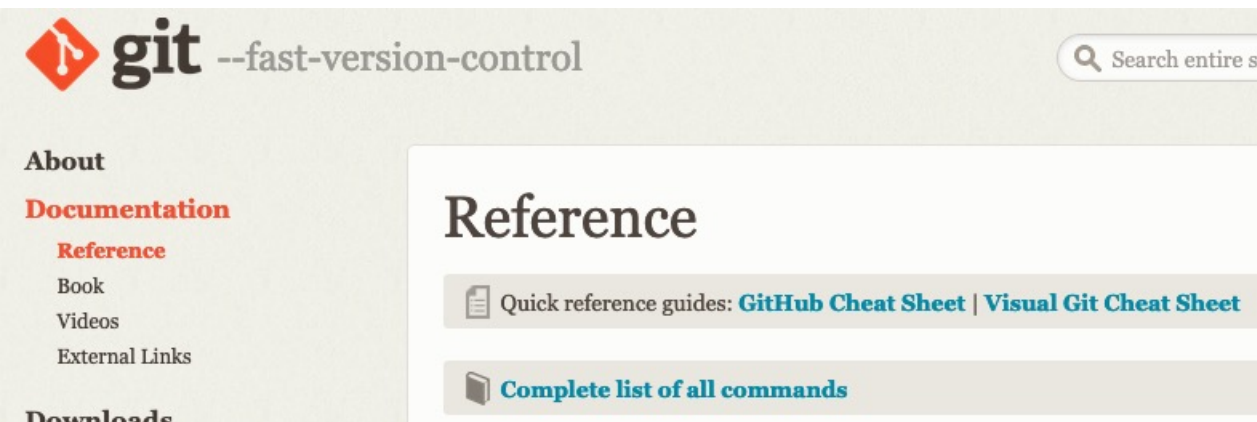
Collaborative programming with GitHub

- Why?
- What?
- How?

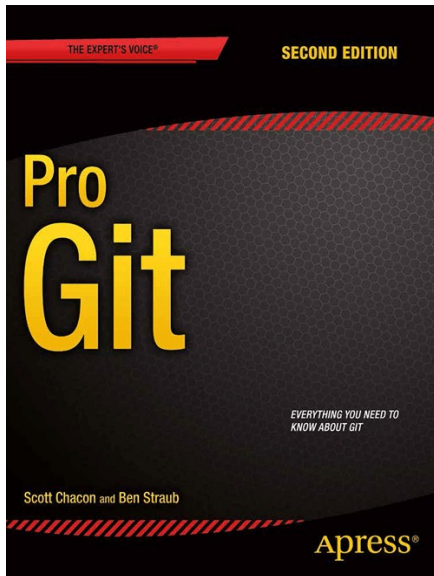
- Danger?
- **Help?**



official git documentation



— Git. “Reference.” Git. <https://git-scm.com/docs>.



— Chacon, Scott, and Ben Straub. *Pro Git*. 2nd ed. Apress, 2014. <https://git-scm.com/book/en/v2>.

Everyday Git in twenty commands or so

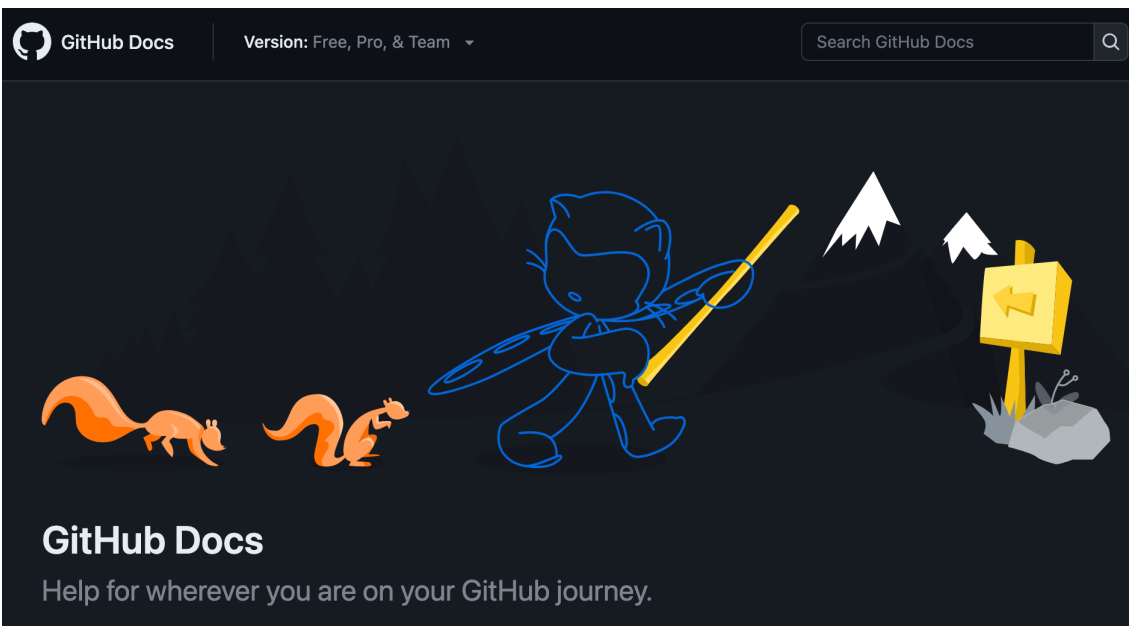
```
git help everyday
```

Show helpful guides that come with Git

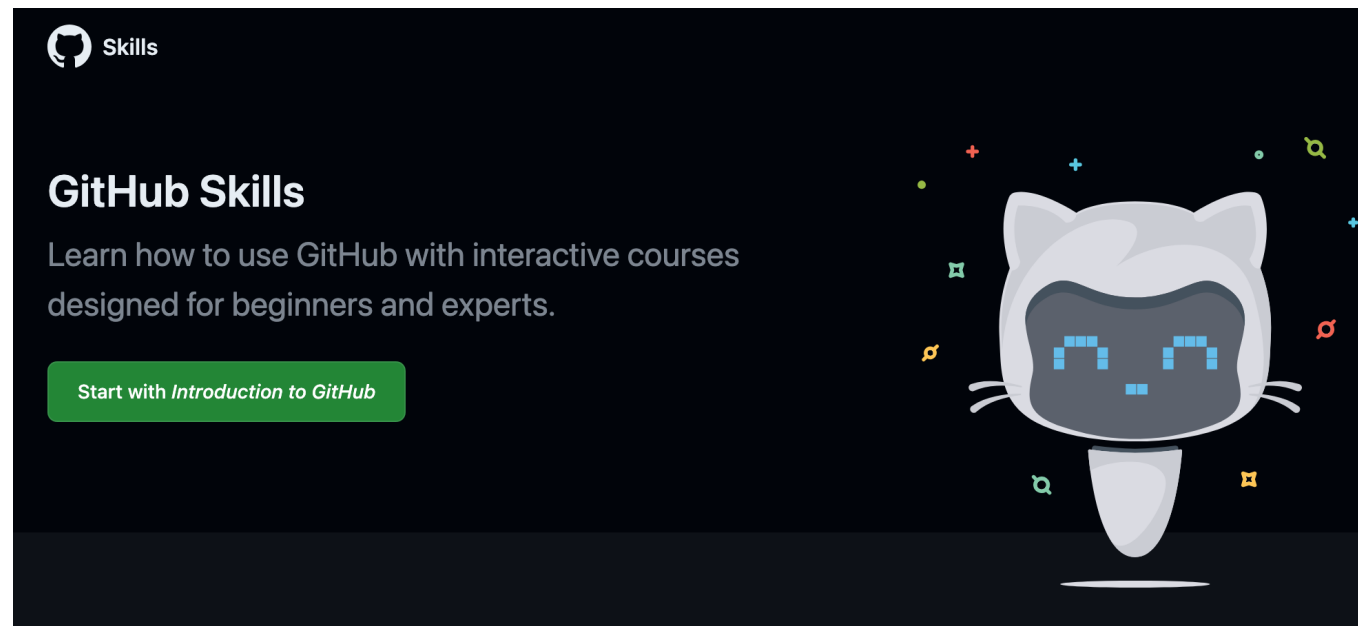
```
git help -g
```

— git-tips. “Git-Tips.” 2024. git.io/git-tips.

official GitHub documentation

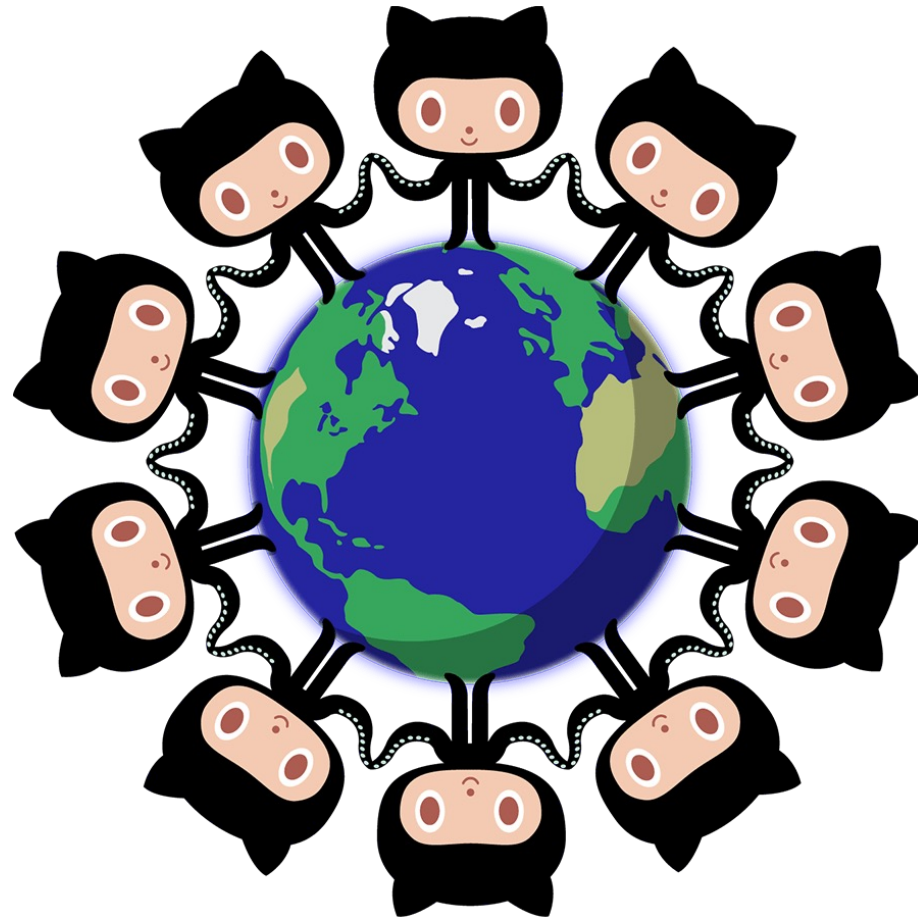


— GitHub, Inc. “GitHub Docs.” GitHub Docs, 2024.
<https://docs.github.com/>.



— GitHub, Inc. “GitHub Skills,” 2023.
<https://skills.github.com/>.

Thank you for your time and attention.



This and all other octocat illustrations in
this presentation from Octodex
(octodex.github.com)



**Child Mind
Institute**

Transforming Children's Lives

childmind.org