Version Control using GitHub and Python.

Juan F. Imbet

Paris Dauphine University

juan.imbet@dauphine.psl.eu
jfimbett.github.io

GitHub

Github is a cloud-based platform to store and share code.

- Showcase or share your work.
- Track and manage changes to your code over time.
- Let others review your code, and make suggestions to improve it.
- Collaborate on a shared project, without worrying that your changes will impact the work of your collaborators before you're ready to integrate them

Git

Git is a version control system. It is useful when you and other people are working on the same project. It was created by Linus Torvalds in 2005 (the creator of Linux). Typically, to do this in a Git-based workflow, you would:

- Create a branch off from the main copy of files that you (and your collaborators) are working on.
- Make edits to the files independently and safely on your own personal branch.
- Let Git intelligently merge your specific changes back into the main copy of files, so that your changes don't impact other people's updates.
- Let Git keep track of your and other people's changes, so you all stay working on the most up-to-date version of the project

How do Git and GitHub work together?

- When you upload files to GitHub, you'll store them in a "Git repository." This means that when you make changes (or "commits") to your files in GitHub, Git will automatically start to track and manage your changes.

Once you start to collaborate with others and all need to work on the same repository at the same time, you’ll continually:

- Pull all the latest changes made by your collaborators from the remote repository on GitHub.
- Push back your own changes to the same remote repository on GitHub.

Requirements

- Create an account on GitHub
- Install Git on your computer.
- Connect to GitHub
- Install the GitHub CLI

Communicating on GitHub

GitHub provides built-in collaborative communication tools allowing you to interact closely with your community. E.g. you can follow the last updates of your favorite library, or even your favorite programming language.

GitHub Issues

- Are useful for discussing specific details of a project such as bug reports, planned improvements and feedback.
- Are specific to a repository, and usually have a clear owner.
- Are often referred to as GitHub's bug-tracking system.

Pull Requests

- Allow you to propose specific changes
- Allow you to comment directly on proposed changes suggested by others
- Are specific to a repository
- Provide a collaborative experience outside the codebase, allowing the brainstorming of ideas, and the creation of a community knowledge base

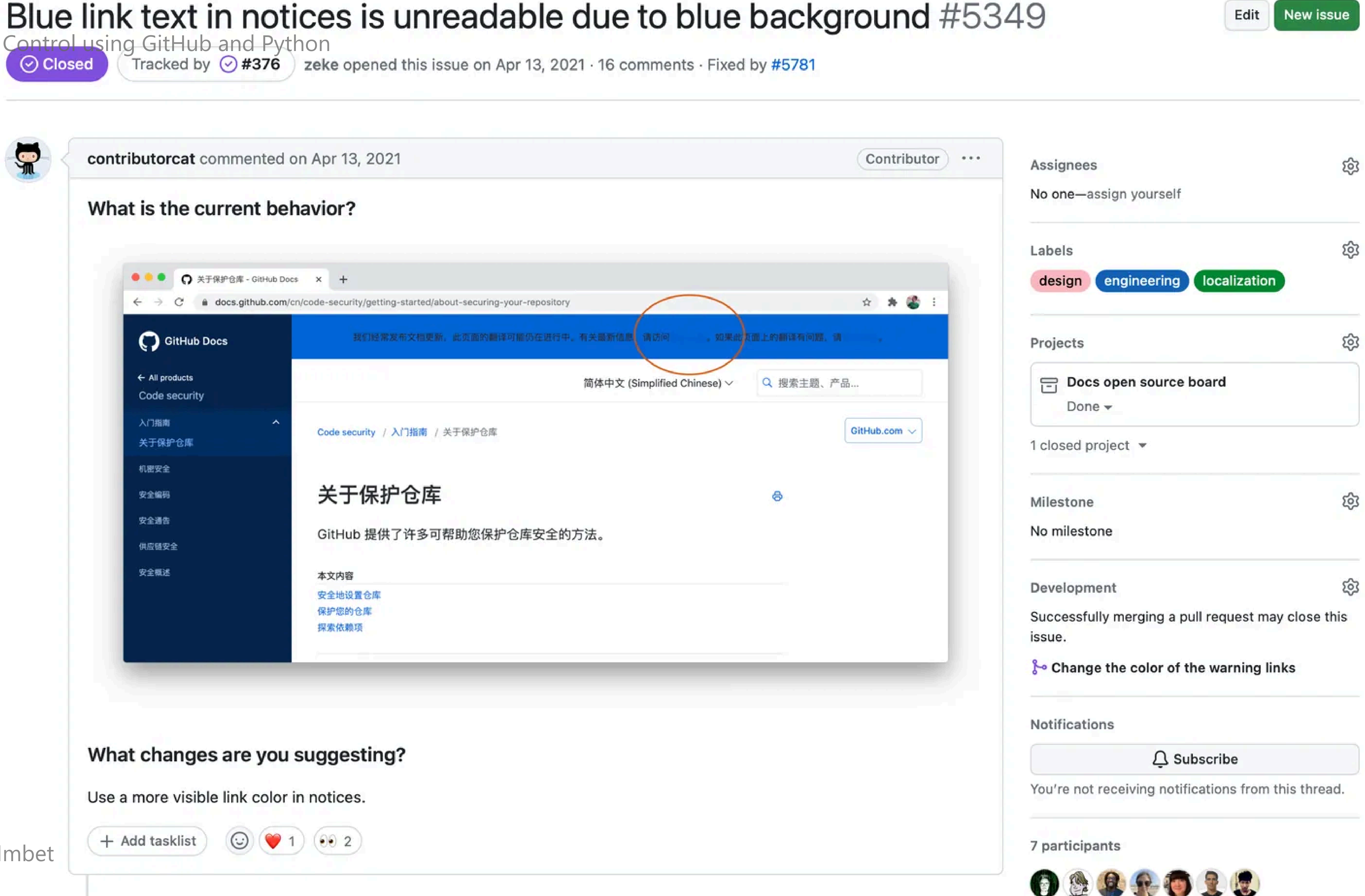
GitHub Discussions

- Are like a forum, and are best used for open-form ideas and discussions where collaboration is important.
- May span many repositories

What should I use?

Issues

- I want to keep track of tasks, enhancements and bugs.
- I want to file a bug report.
- I want to share feedback about a specific feature.
- I want to ask a question about files in the repository.



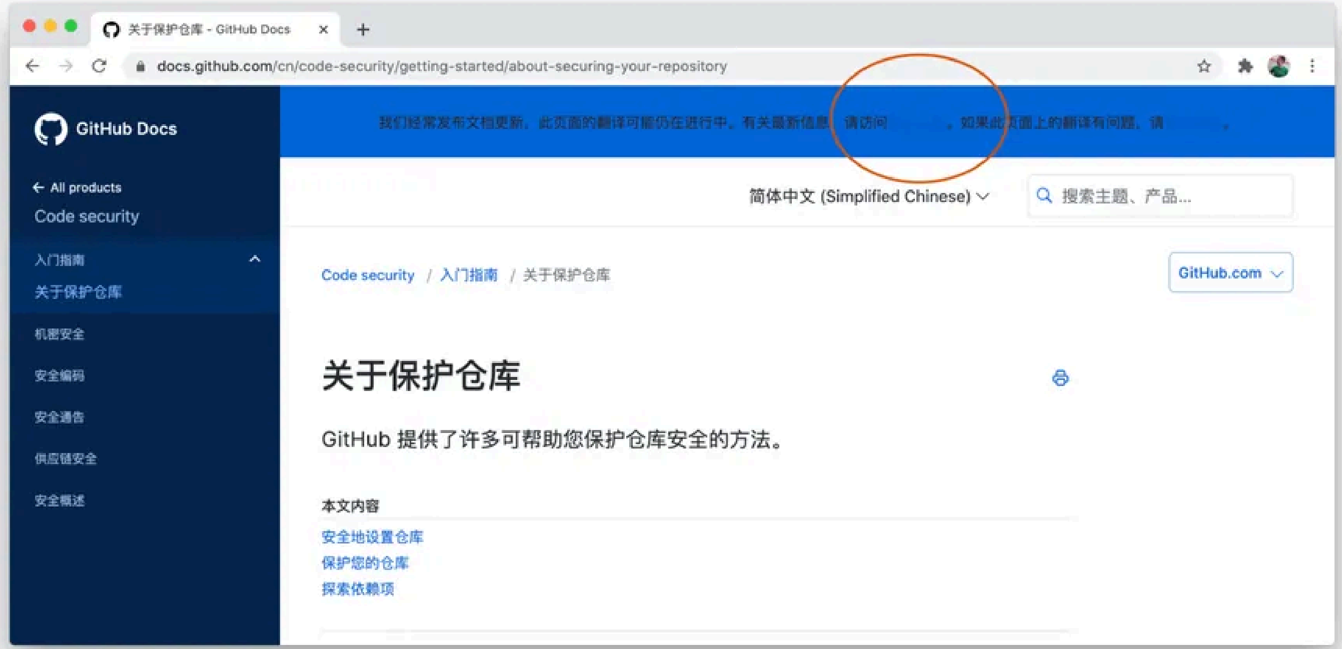
Blue link text in notices is unreadable due to blue background #5349

Edit | New issue

Closed | Tracked by #376 | **zeke** opened this issue on Apr 13, 2021 · 16 comments · Fixed by #5781

contributorcat commented on Apr 13, 2021 · Contributor

What is the current behavior?



关于保护仓库 - GitHub Docs

docs.github.com/cn/code-security/getting-started/about-securing-your-repository

GitHub Docs

我们经常发布文档更新，此页面的翻译可能仍在进行中。有关最新信息，请访问 。如果此页面上的翻译有问题，请 。

← All products
Code security
入门指南
关于保护仓库
机密安全
安全编码
安全通告
供应链安全
安全概述

简体中文 (Simplified Chinese) | 搜索主题、产品...

Code security / 入门指南 / 关于保护仓库 | GitHub.com

关于保护仓库

GitHub 提供了许多可帮助您保护仓库安全的方法。

本文内容
安全地设置仓库
保护您的仓库
探索依赖项

What changes are you suggesting?

Use a more visible link color in notices.

+ Add tasklist | ❤️ 1 | 👀 2

Assignees
No one—assign yourself

Labels
design | engineering | localization

Projects
Docs open source board
Done
1 closed project

Milestone
No milestone

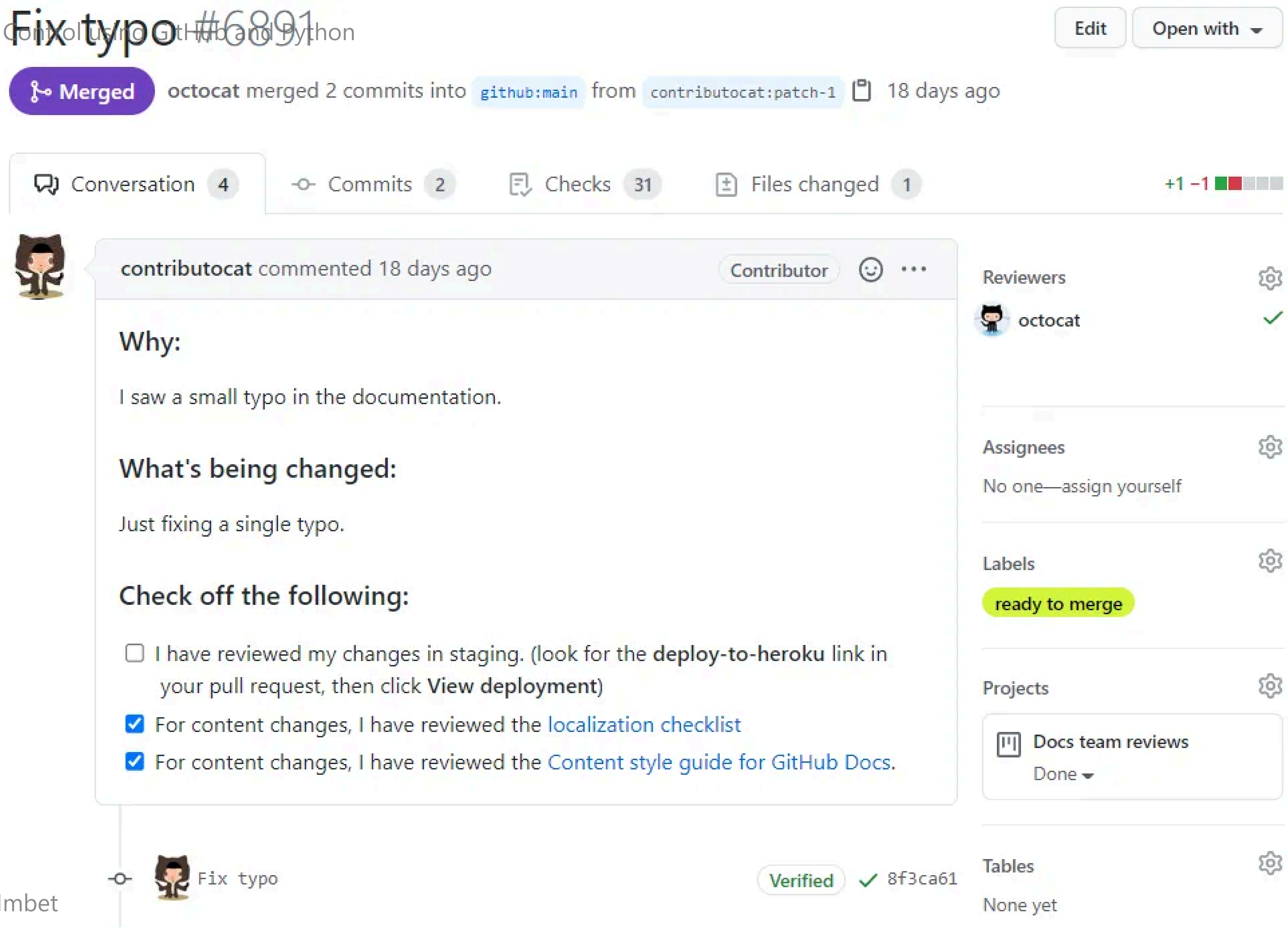
Development
Successfully merging a pull request may close this issue.
Change the color of the warning links

Notifications
Subscribe
You're not receiving notifications from this thread.

7 participants

Scenarios for Pull Requests

- I want to fix a typo in a repository.
- I want to make changes to a repository.
- I want to make changes to fix an issue.
- I want to comment on changes suggested by others.



Fix typo #6891

Merged octocat merged 2 commits into github:main from contributocat:patch-1 18 days ago

contributocat commented 18 days ago Contributor

Why:

I saw a small typo in the documentation.

What's being changed:

Just fixing a single typo.

Check off the following:

- [ ] I have reviewed my changes in staging. (look for the **deploy-to-heroku** link in your pull request, then click **View deployment**)
- [x] For content changes, I have reviewed the localization checklist
- [x] For content changes, I have reviewed the Content style guide for GitHub Docs.

Fix typo Verified ✓ 8f3ca61

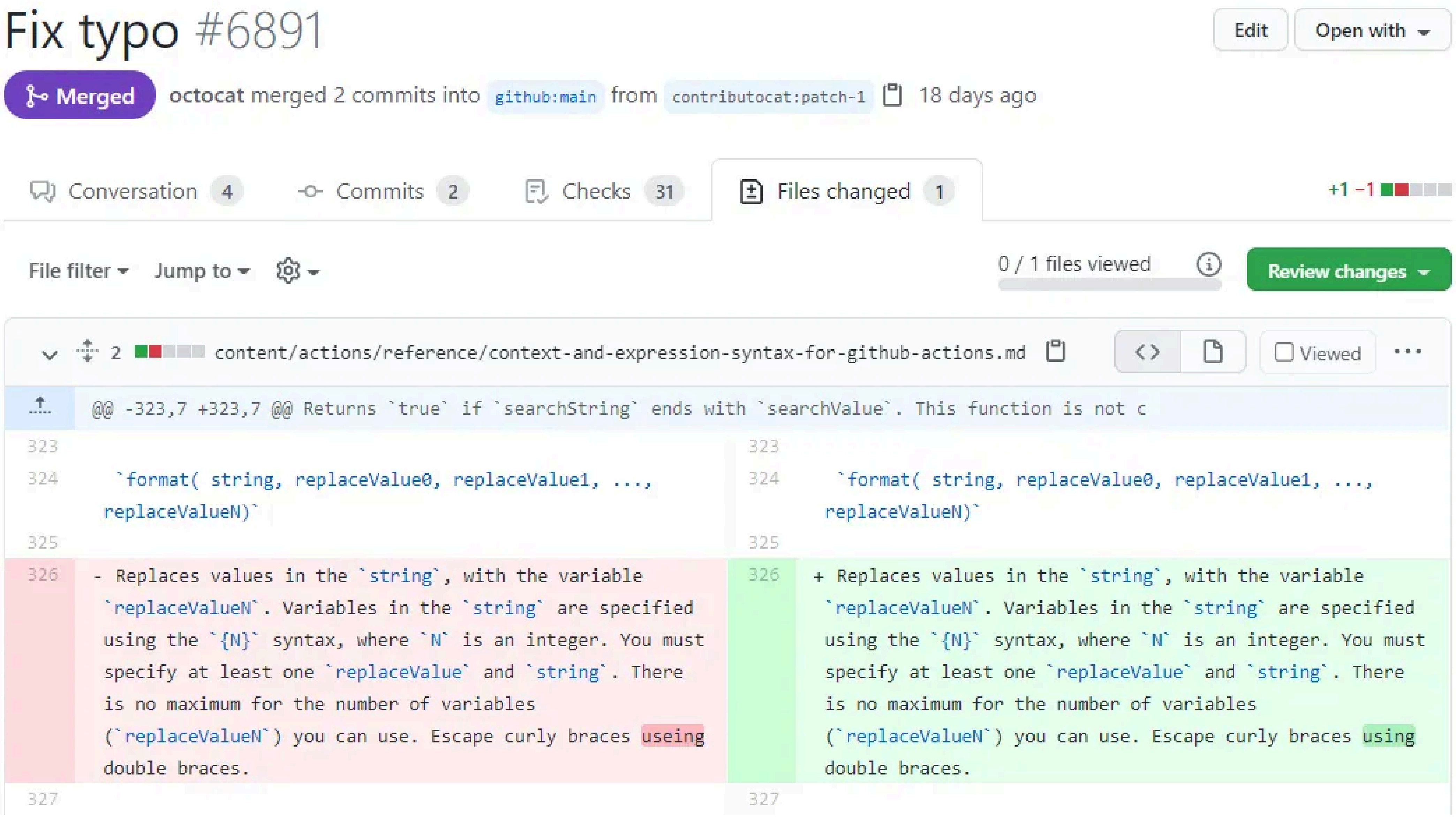
Reviewers: octocat ✓

Assignees: No one—assign yourself

Labels: ready to merge

Projects: Docs team reviews — Done

Tables: None yet



Fix typo #6891

Merged octocat merged 2 commits into `github:main` from `contributocat:patch-1` 18 days ago

Conversation 4 Commits 2 Checks 31 Files changed 1 +1 −1

File filter Jump to 0 / 1 files viewed Review changes

2 content/actions/reference/context-and-expression-syntax-for-github-actions.md Viewed

```
@@ -323,7 +323,7 @@ Returns `true` if `searchString` ends with `searchValue`. This function is not c
323                                                   323
324  `format( string, replaceValue0, replaceValue1, ..., replaceValueN)`      324  `format( string, replaceValue0, replaceValue1, ..., replaceValueN)`
325                                                   325
326 - Replaces values in the `string`, with the variable `replaceValueN`. Variables in the `string` are specified using the `{N}` syntax, where `N` is an integer. You must specify at least one `replaceValue` and `string`. There is no maximum for the number of variables (`replaceValueN`) you can use. Escape curly braces useing double braces.
326 + Replaces values in the `string`, with the variable `replaceValueN`. Variables in the `string` are specified using the `{N}` syntax, where `N` is an integer. You must specify at least one `replaceValue` and `string`. There is no maximum for the number of variables (`replaceValueN`) you can use. Escape curly braces using double braces.
327                                                   327
```

Scenarios for Discussions

- I have a question that's not necessarily related to specific files in the repository.
- I want to share news with my collaborators, or my team.
- I want to start or participate in an open-ended conversation.
- I want to make an announcement to my community.

Feature preview

GitHub's release cycle

GitHub's products and features can go through multiple release phases.

- Alpha: The product or feature is under heavy development and often has changing requirements and scope
- Beta: The product or feature is in a more stable state and is available to a limited number of users
- General Availability (GA): The product or feature is fully released and available to all users

Github has a feature preview that allows you to try out new features of a repository before they are released to the general public.

Interacting with GitHub

- Browser (only use one of the following)
  - Apple Safari
  - Google Chrome
  - Microsoft Edge
  - Mozilla Firefox
- GitHub Mobile
- Command Line Interface (CLI)

I recommend to do (almost) everything from the command line.

Writing on GitHub: Markdown

- Markdown is an easy-to-read, easy-to-write language for formatting plain text. You can use Markdown syntax, along with some additional HTML tags, to format your writing on GitHub, in places like repository READMEs and comments on pull requests and issues.
- **These slides are written in Marp, a language based on Markdown for slides.**.
- Markdown files are saved with the `.md` extension. The most common file in any repository is the `README.md` file.
- You can use HTML tags in Markdown files for more complex formatting.
- Click here for a guide on how to write in Markdown.

Basic Markdown Syntax

```
# Heading 1
## Heading 2
### Heading 3

*italic*
**bold**
***bold italic***

* List item 1
1. List item 2

Plain text
```

Heading 1

Heading 2

Heading 3

italic

bold

bold italic

- List item 1

1. List item 2

Plain text

Finding ways to contribute to open source on GitHub

- Discovering relevant projects `github.com/topics/<topic>`
- Finding good first issues `github.com/<owner>/<repository>/contribute`.
- Opening an issue.

Let's get started

Main commands, `git` and `gh`

- `git` is the command line tool for Git, this is the command we will use more often.
- `gh` is the command line tool for GitHub, it is useful to create repositories and manage issues.

Command Line Interface

```
git --version
git config --global user.name "Juan Imbett"
git config user.name
git config --global user.email "jfimbett@gmail.com"
git config --global user.email
```

```
git version 2.44.0.windows.1
Juan Imbett
jfimbett@gmail.com
```

Make sure you are connected to GitHub

```
gh auth login
```

HTTPS and SSH

- HTTPS is the default way to connect to GitHub. It stands for HyperText Transfer Protocol Secure. It is a secure way to transfer data between your computer and GitHub.
- SSH stands for Secure Shell. It is a secure way to connect to GitHub. It is more secure than HTTPS, but it requires more configuration. Since we do not have nuclear codes in our repositories, we will use HTTPS.

Step 3: Create a Local Repository

```
cd
mkdir hello-world
cd hello-world
git init # creates .git folder
```

```
Initialized empty Git repository in C:/Users/jfimb/Documents/hello-world/.git/
```

Step 4: Add Files to the Repository

Windows

```
echo # My First Commit >> README.md
more README.md
git add .
git commit -m "First commit"
```

iOS

```
echo "# My First Commit" >> README.md
cat README.md
git add .
git commit -m "First commit"
```

```
[master (root-commit) eaef7df] First commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

Step 5: Create a Remote Repository

```
gh repo create hello-world --public --source=.
```

```
✓ Created repository jfimbett/hello-world on GitHub
  https://github.com/jfimbett/hello-world
✓ Added remote https://github.com/jfimbett/hello-world.git
```

Step 6: Push the Local Repository to GitHub

```
git push -u origin master
```

Where `origin` is the name of the remote repository and `master` is the name of the branch (more on this later)

Step 7: Check the Repository on GitHub

Step 8: Pulling changes from GitHub

- Go directly to the GitHub website and make a change to the README.md file.
- Pull the changes to your local repository.

```
git pull origin master
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 923 bytes | 115.00 KiB/s, done.
From https://github.com/jfimbett/hello-world
 * branch            master     -> FETCH_HEAD
   8db2404..b1066dc  master     -> origin/master
Updating 8db2404..b1066dc
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
```

Step 9: Create a Branch

- A branch is a copy of the main repository. It is useful when you want to work on a feature without affecting the main repository.
- Let's create a branch called `feature1`, create a python file and push it to the remote repository.

```
git checkout -b feature1 # creates and switches to the branch
echo print('Hello World') >> hello.py # creates a python file
git add . # adds the file to the staging area
git commit -m "Add hello.py" # commits the file
git push origin feature1 # pushes the branch to the remote repository
```

```
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 36 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 302 bytes | 302.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
remote:
remote: Create a pull request for 'feature1' on GitHub by visiting:
remote:      https://github.com/jfimbett/hello-world/pull/new/feature1
remote:
To https://github.com/jfimbett/hello-world.git
 * [new branch]      feature1 -> feature1
```

Check the repository on GitHub

- You will see a new branch called `feature1` with the file `hello.py`.
- The branch will be 1 commit ahead of the `master` branch.
- You can create a pull request to merge the changes to the `master` branch.

```
gh pr create --base master --head feature1
```

Let's explore the command. `--base` is the branch where you want to merge the changes. `--head` is the branch where the changes are.

```
Creating pull request for feature1 into master in jfimbett/hello-world

? Title Add hello.py
? Body <Received>
? What's next? Submit
https://github.com/jfimbett/hello-world/pull/1
```

- You should see a pull request on the GitHub website.

A pull request is a proposal to merge a set of changes from one branch into another. In a pull request, collaborators can review and discuss the proposed set of changes before they integrate the changes into the main codebase.

Commenting on a Pull Request

- The repository administrator is the one that can accept or reject the pull request. As an admin you can make commens on the pull request.
- Let's imagine that the admin asks you to make a change to the file `hello.py`. You can do it in the browser, for example asking to have double quotes instead of single quotes.
- You can check the pull request comments witht he command `gh pr view 1`.

```
gh pr view 1
```

```
Add hello.py jfimbett/hello-world#1
Open • jfimbett wants to merge 1 commit into master from feature1 • about 10 minutes ago
+1 -0 • No checks


  No description provided


jfimbett commented (Owner) • 0m • Newest comment

  Good commit, but please do the changes.


View this pull request on GitHub: https://github.com/jfimbett/hello-world/pull/1
```

Making changes to the Pull Request

- You can make changes to the pull request by checking out the branch and making the changes.

```
git checkout feature1
echo print("Hello World") > hello.py ## two > are for appending, one is for overwriting
git add .
git commit -m "Change hello.py"
git push origin feature1
```

Step 10: Merge the Pull Request

- Once the changes are made, you can accept the pull request.

```
gh pr merge 1 --delete-branch # merges the pull request and deletes the branch,
gh pr merge 1 # merges the pull request and keeps the other branch
```

```
Merging pull request jfimbett/hello-world#1 (Add hello.py)
? What merge method would you like to use? Create a merge commit
? What's next? Submit
✓ Merged pull request jfimbett/hello-world#1 (Add hello.py)
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (1/1), 889 bytes | 222.00 KiB/s, done.
From https://github.com/jfimbett/hello-world
 * branch            master     -> FETCH_HEAD
   b1066dc..66a1170  master     -> origin/master
Updating b1066dc..66a1170
Fast-forward
 hello.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
✓ Deleted local branch feature1 and switched to branch master
✓ Deleted remote branch feature1
```

Forking and Cloning a Repository: Explore ideas before they are proposed.

- A fork is a new repository that shares code and visibility settings with the original “upstream” repository. It doesnt download the code to your local machine by itself.
- Propose changes to someone else's project.
- Use someone else's project as a starting point for your own idea.
- E.g. I like this repository on how to train a neural network to play the game snake. https://github.com/greerviau/SnakeAI
- Cloning is the process of downloading a repository to your local machine.

```
gh repo fork greerviau/SnakeAI --remote=true
```

```
? Would you like to clone the fork? Yes
Cloning into 'SnakeAI'... the fork? (y/N) y
remote: Enumerating objects: 211, done.
remote: Total 211 (delta 0), reused 0 (delta 0), pack-reused 211 (from 1)
Receiving objects: 100% (211/211), 85.75 KiB | 3.30 MiB/s, done.
Resolving deltas: 100% (103/103), done.
From https://github.com/greerviau/SnakeAI
 * [new branch]      master     -> upstream/master
✓ Cloned fork
! Repository greerviau/SnakeAI set as the default repository.
To learn more about the default repository, run: gh repo set-default --help
```

```
cd SnakeAI
```

The .gitignore file

- The `.gitignore` file is a text file that tells Git which files or folders to ignore in a project.

Example of a `.gitignore` file:

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so
```

You should include here large data files.

Step 12: Delete the Local Repository

- Deleting a repository is a dangerous operation, it is irreversible. To avoid any unwanted deletion, I recommend deleting directly on the github website.
- To stop tracking a repository, you can delete the `.git` folder.

iOS

```
rm -rf .git
```

Windows

```
rmdir .git /s /q
```

The Git Fork-Branch-Pull Workflow

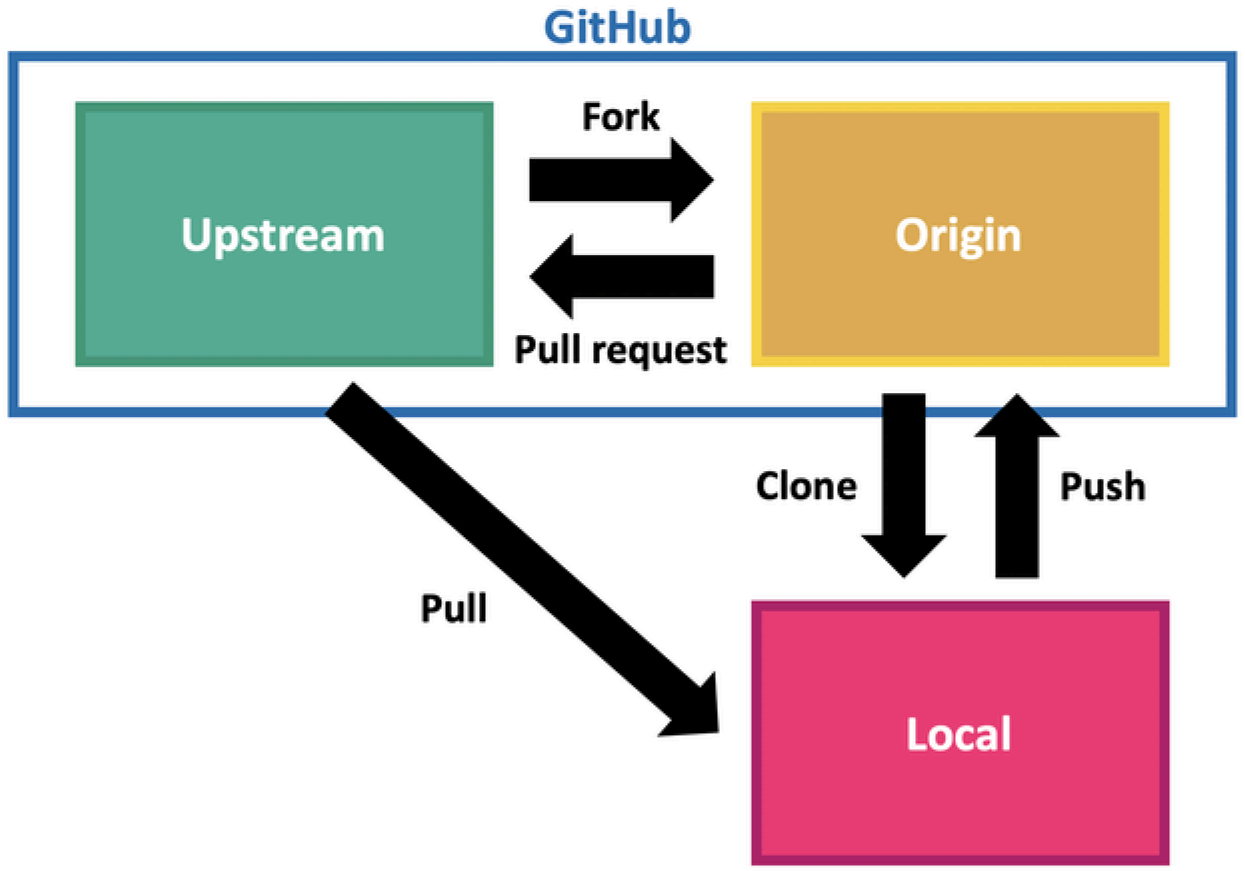
1. Fork and Clone a project.

- Fork: Create a copy of the repository on your GitHub account.
- Clone: Download the repository to your local machine.

```
gh repo fork jfimbett/pybacktestchain --remote=true
```

```
✓ Created fork jfimbett-test/pybacktestchain
? Would you like to clone the fork? Yes
Cloning into 'pybacktestchain'...
remote: Enumerating objects: 128, done.
remote: Counting objects: 100% (128/128), done.
remote: Compressing objects: 100% (69/69), done.
remote: Total 128 (delta 52), reused 110 (delta 39), pack-reused 0 (from 0)
Receiving objects: 100% (128/128), 158.67 KiB | 4.07 MiB/s, done.
Resolving deltas: 100% (52/52), done.
From https://github.com/jfimbett/pybacktestchain
 * [new branch]      branch1           -> upstream/branch1
 * [new branch]      branch2           -> upstream/branch2
 * [new branch]      branch_blockchain -> upstream/branch_blockchain
 * [new branch]      master            -> upstream/master
✓ Cloned fork
! Repository jfimbett/pybacktestchain set as the default repository. To learn more about the default repository, run: gh repo set-default --help
```

Origin vs Upstream vs Local



`origin`

What is it?

- `origin` is the default name given to the remote repository when you clone a repository. It refers to the repository from which your local copy was cloned.

Purpose:

- It's typically where you push your changes (e.g., git push origin main) and pull updates from.
- If you fork a repository on GitHub and then clone your fork locally, origin points to your forked repository (i.e., your own copy of the repository on GitHub).
- If you fork a repository using `gh repo fork` the `--remote=true` flag will automatically bring both the `origin` and `upstream` remotes.

`upstream`

What is it?

`upstream` is a common convention (but not a special keyword in Git) to refer to the original repository from which you forked a project. It points to the repository you forked from, typically maintained by the original author or organization.

- Purpose: It is used to pull changes from the original repository into your fork. This is especially useful when you want to keep your fork up-to-date with the latest changes from the original repository.
- It helps maintain a link to the "source of truth" repository.

What are the branches I have access to?

```
cd repository
git branch -a
```

```
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/branch1
  remotes/origin/branch2
  remotes/origin/branch_blockchain
  remotes/origin/master
  remotes/upstream/branch1
  remotes/upstream/branch2
  remotes/upstream/branch_blockchain
  remotes/upstream/master
```

What if I only have `origin` or `upstream` but not both?

- You can add a remote repository with the command `git remote add <name> <url>`.

```
git remote add upstream jfimbett/pybacktestchain.git
```

Why would I want both?

- `origin` points to `https://github.com/yourusername/repo.git`.
- `upstream` points to `https://github.com/originaluser/repo.git`

`fetch`

- `fetch` is a command that downloads changes from a remote repository to your local repository. It does not merge the changes, it only downloads them.

```
git fetch upstream
```

Pull from a branch

`git checkout upstream/branch_blockchain`

```
Switched to a new branch 'branch_blockchain'
```

```
git pull upstream branch_blockchain
```

```
From https://github.com/jfimbett/pybacktestchain
 * branch            branch_blockchain -> FETCH_HEAD
```

What if there is a new branch in the `upstream` repository?

Before

```
master
remotes/origin/HEAD -> origin/master
remotes/origin/branch1
remotes/origin/branch2
remotes/origin/branch_blockchain
remotes/origin/master
remotes/upstream/branch1
remotes/upstream/branch2
remotes/upstream/branch_blockchain
remotes/upstream/master
```

`fetch`

```
git fetch upstream
```

```
remotes/origin/HEAD -> origin/master
remotes/origin/branch1
remotes/origin/branch2
remotes/origin/branch_blockchain
remotes/origin/master
remotes/upstream/branch1
remotes/upstream/branch2
remotes/upstream/branch_blockchain
remotes/upstream/master
remotes/upstream/testing_branch
```

Version Control

- How to automate the process of version control?
- Python through Anaconda.
- Cookiecutter: A tool that helps you create a project template.
- Poetry: Dependency management and packaging in Python.
- Semantic Release: A tool that automates the versioning and release process based on the commit messages.
- For this example let's create an empty repository, that we will convert into a library, upload to GitHub and manage the version control.

Anaconda

- Install Python through Anaconda and create a virtual environment. This will allow you to install packages without affecting the base environment, plus you can easily share the environment with others. Different versions of the same libraries/packages can have different behavior. A good practice is to create a virtual environment for each project, you can use the same name as the project.

```
conda create --name mylibrary python=3.11
conda activate mylibrary
```

```
(mylibrary) C:\Users\jfimb\Documents> # Look at the (mylibrary) at the beginning of the line
```

Cookiecutter

- Cookiecutter is a command-line utility that creates projects from project templates.
- A professional project consists of a set of files and directories that are organized in a specific way. E.g. tests, documentation, source code, etc.
- You can look for your favorite template here
- Install cookiecutter using `conda` rather than `pip` to avoid conflicts with other packages.

```
conda install -c conda-forge cookiecutter
```

Treat your code as a package/library

Our goal is to creat something that looks like this, a package structure. Even if you dont use python, you can have a similar structure with another language.

```
mylibrary
├── CHANGELOG.md                ┐
├── CONDUCT.md                  │
├── CONTRIBUTING.md             │
├── docs                        │ Package documentation
│   └── ...                     │
├── LICENSE                     │
├── README.md                   ┘
├── pyproject.toml              ┐
├── src                         │
│   └── mylibrary         │ Package source code, metadata,
│       ├── __init__.py         │ and build instructions
│       ├── moduleA.py          │
│       └── moduleB.py          ┘
└── tests                       ┐
    └── ...                     ┘ Package tests
```

Notes:

- The `CHANGELOG.md` file contains a list of changes made to the package. It is useful to keep track of the changes made to the package.
- The `CONDUCT.md` file contains the code of conduct of the package. It is useful to keep track of the rules of the package.
- The `CONTRIBUTING.md` file contains the guidelines for contributing to the package. It is useful to keep track of the contributions to the package.
- The `docs` folder contains the documentation of the package. It is useful to keep track of the documentation of the package.
- The `LICENSE` file contains the license of the package. It is useful to keep track of the license of the package.

Important

- Do not worry about the structure of the package src with a directory called `mylibrary`. This is the way python packages are organized. The `__init__.py` file is necessary for python to recognize the directory as a package.

Retrieve a template

`py-pkgs` provides a template for Python packages.

```
cookiecutter https://github.com/py-pkgs/py-pkgs-cookiecutter.git
```

```
[1/7] author_name (Monty Python): Juan F. Imbet
[2/7] package_name (mypkg): mylibrary
[3/7] package_short_description (A package for doing great things!): Example of how to use templates in cookiecutter.
[4/7] package_version (0.1.0): 0.0.0
[5/7] python_version (3.9):
[6/7] Select open_source_license
  1 - MIT
  2 - Apache License 2.0
  3 - GNU General Public License v3.0
  4 - CC0 v1.0 Universal
  5 - BSD 3-Clause
  6 - Proprietary
  7 - None
  Choose from [1/2/3/4/5/6/7] (1):
[7/7] Select include_github_actions
  1 - no
  2 - ci
  3 - ci+cd
  Choose from [1/2/3] (1):
```

Important:

- Select 0.0.0 for the package version, this will allow us to use `semantic-release` to calculate the next version based on the commit message history.

```
cd mylibrary
dir # or ls in iOS
```

```
    Directory: C:\Users\jfimb\Documents\mylibrary


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        28/08/2024     15:51                docs
d-----        28/08/2024     15:51                src
d-----        28/08/2024     15:51                tests
-a----        28/08/2024     15:51           2066 .gitignore
-a----        28/08/2024     15:51            624 .readthedocs.yml
-a----        28/08/2024     15:51            107 CHANGELOG.md
-a----        28/08/2024     15:51           3090 CONDUCT.md
-a----        28/08/2024     15:51           2319 CONTRIBUTING.md
-a----        28/08/2024     15:51           1094 LICENSE
-a----        28/08/2024     15:51            365 pyproject.toml
-a----        28/08/2024     15:51            681 README.md
```

Poetry

- Poetry is a tool that helps you manage dependencies and packaging in Python. It is a good practice to use it when you are creating a package.

```
pip install poetry
```

Cookiecutter has already created a `pyproject.toml` file for you. This file contains the metadata of the package, the dependencies, and the build instructions.

`pyproject.toml`

- `toml` stands for Tom's Obvious, Minimal Language. It is a configuration file format that is easy to read due to its simplicity. It was created by Tom Preston-Werner, the co-founder of GitHub.

```toml
[tool.poetry]
name = "mylibrary"
version = "0.0.0"
description = "Example of how to use templates in cookiecutter."
authors = ["Juan F. Imbet"]
license = "MIT"
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.9"

[tool.poetry.dev-dependencies]

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Poetry, the basics

- `poetry install` installs the dependencies.
- `poetry add <package>` adds a package to the dependencies.
- `poetry remove <package>` removes a package from the dependencies.
- `poetry build` builds the package.
- `poetry publish` publishes the package to PyPI (requires an account).

Sync with GitHub

```
git init
git add .
git commit -m "First commit"
gh repo create mylibrary --public --source=.
git push -u origin master
```

Version Control `major`.`minor`.`patch`

- The most common way to define a version of a package is with three numbers separated by dots. The first number is the major version, the second number is the minor version, and the third number is the patch version.
- Patchs are for bug fixes, minors are for new features, and majors are for breaking changes.
- Patchs and minors are backward compatible, this means that if you have a package that depends on version 1.0.0, it will work with version 1.0.1 and 1.1.0, but not necessarily with version 2.0.0.

Semantic Release

- Semantic Release is a tool that automates the versioning and release process based on the commit messages. It is a good practice to use it when you are creating a package.

```
<type>(optional scope): short summary in present tense

(optional body: explains motivation for the change)

(optional footer: note BREAKING CHANGES here, and issues to be closed)
```

`<type>` refers to the kind of change made and is usually one of:

- `feat`: A new feature.
- `fix`: A bug fix.
- `docs`: Documentation changes.
- `style`: Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc).
- `refactor`: A code change that neither fixes a bug nor adds a feature.
- `perf`: A code change that improves performance.
- `test`: Changes to the test framework.
- `build`: Changes to the build process or tools.

Examples

A `type` of `fix` triggers a patch version bump.

```
git commit -m "fix(mod_plotting): fix confusing error message in \
                 plot_words"
```

A `type` of `feat` triggers a minor version bump.

```
git commit -m "feat(package): add example data and new module to \
                 package"
```

The text `BREAKING_CHANGE:` in the footer triggers a major version bump.

```
git commit -m "feat(mod_plotting): move code from plotting module \
              to pycounts module

BREAKING CHANGE: plotting module wont exist after this release."
```

Configuration

To configure PSR, we need to tell it where the version number of our package is stored. The package version is stored in the pyproject.toml file for a poetry-managed project. It exists as the variable version under the table `[tool.poetry]`. To tell PSR this, we need to add a new table to synchronize the version number.

```
...rest of file hidden...

[tool.semantic_release]
version_variable = "pyproject.toml:version"
```

We can do it with the following command:

```
poetry add --group dev python-semantic-release
echo [tool.semantic_release] >> pyproject.toml
echo version_variable = "pyproject.toml:version" >> pyproject.toml
echo version_toml = [ >> pyproject.toml
echo "pyproject.toml:tool.poetry.version", >> pyproject.toml
echo ] >> pyproject.toml
poetry install
```

Installing dependencies

```
Installing dependencies from lock file

Package operations: 0 installs, 2 updates, 0 removals

  - Updating certifi (2024.7.4 /home/conda/feedstock_root/build_artifacts/certifi_1720457958366/work/certifi -> 2024.7.4)
  - Updating urllib3 (2.2.2 /home/conda/feedstock_root/build_artifacts/urllib3_1719391292974/work -> 2.2.2)

Installing the current project: mylibrary (0.0.0)
```

Look at the `pyproject.toml` file, you will see the new table `[tool.semantic_release]`. Also note that the version 0.0.0 is the same one that appears in the `pyproject.toml` file.

```toml
[tool.poetry]
name = "mylibrary"
version = "0.0.0"
```

What if I need more libraries in the future?

- You can add them to the `pyproject.toml` file with the command `poetry add <package>`.
- You can remove them with the command `poetry remove <package>`.
- E.g. `poetry add numpy` or `poetry remove numpy`.

Semantic Release, the basics

- `semantic-release version` calculates the next version based on the commit message history.

```
semantic-release version
```

```
0.0.0
The next version is: 0.0.0! 🚀
No build command specified, skipping
```

Note: Failed to create release on Github!

Sometimes you will see the following message:

```
[09:46:57] ERROR    [semantic_release.cli.commands.version] ERROR version.version: 404 Client Error: Not  version.py:744
                    Found for url: https://api.github.com/repos/jfimbett/sec_yf/releases
                    NoneType: None
404 Client Error: Not Found for url: https://api.github.com/repos/jfimbett/sec_yf/releases
Failed to create release on Github!
```

This is because semantic-release requires a GitHub token to create a release.

Create a GitHub token

(Thanks to Samuli Salonen for finding this solution)

- Go to https://github.com/settings/tokens
- Generate new token with “repo” and “workflow” access
- Copy the token (only shown once, so save it somewhere)
- Add the token to your environment variables with the name `GH_TOKEN`. In Windows, you can do this with the command `setx GH_TOKEN <your token>`.
- Remember to restart your terminal after setting the environment variable.

Instructions for MacOS (zsh)

1. For `zsh` users, (default chell on macOS Catalina and later).
2. Open your terminal
3. Type `nano ~/.zshrc` this will open the `.zshrc` file in the nano text editor (a minimalistic text editor that comes by default in most Unix systems).
4. Add the following line to the end of the file

```
export GH_TOKEN=<your token>
```

5. Save the file and exit. (Look at the bottom of the terminal for the commands to save and exit).
6. Apply the changes with the command `source ~/.zshrc` .
7. For `bash` users do the same but with the file `~/.bash_profile` .

Patches

Add some code to src/mylibrary/mylibrary.py

```python
# src/mylibrary/mylibrary.py
def hello_world():
    return "Hello World"
```

```
git add src/mylibrary/mylibrary.py
git commit -m "fix(mylibrary): add hello_world function"
git push -u origin master
```

```
[master 07e4667] fix(mylibrary): add hello_world function
 1 file changed, 2 insertions(+)

Enumerating objects: 9, done.
...
branch 'master' set up to track 'origin/master'.
```

Patches with Semantic Release

```
semantic-release version
```

```
0.0.1
The next version is: 0.0.1! 🚀
No build command specified, skipping
```

Documentation

Add documentation to the `hello_world` function.

```python
# src/mylibrary/mylibrary.py
def hello_world():
    """Returns the string 'Hello World'."""
    return "Hello World"
```

```
git add src/mylibrary/mylibrary.py
git commit -m "docs(mylibrary): add documentation to hello_world function"
git push -u origin master
```

```
[master 654d724] docs(mylibrary): add documentation to hello_world function
 1 file changed, 1 insertion(+)

Enumerating objects: 9, done.
...
branch 'master' set up to track 'origin/master'.
```

Not every commit triggers a version bump

```
semantic-release version
```

```
0.0.1
No release will be made, 0.0.1 has already been released!
```

Minor change, backward compatible

Add a new function to the `mylibrary.py` file.

```python
# src/mylibrary/mylibrary.py
def hello_world():
    """Returns the string 'Hello World'."""
    return "Hello World"

def hello_world2():
    """Returns the string 'Hello World 2'."""
    return "Hello World 2"
```

```
git add src/mylibrary/mylibrary.py
git commit -m "feat(mylibrary): add hello_world2 function"
git push -u origin master
```

```
[master a5094a5] feat(mylibrary): add hello_world2 function
 1 file changed, 5 insertions(+), 1 deletion(-)
 ...
```

Minor change with Semantic Release

```
semantic-release version
```

```
0.1.0
The next version is: 0.1.0! 🚀
No build command specified, skipping
```

Major change, not backward compatible

Move the `hello_world` function to a new file called `hello.py` and remove the `hello_world2` function.

```python
# src/mylibrary/hello.py
def hello_world():
    """Returns the string 'Hello World'."""
    return "Hello World"
```

```python
# src/mylibrary/mylibrarypy

```

```
git add . # we use . when we modify multiple files
git commit -m "feat: update the code majorly" -m "BREAKING CHANGE: hello_world2 function removed it cannot be used"
git push -u origin master
```

Note how you can use more than once the `-m` flag to add more information to the commit message.

Major change with Semantic Release

```
semantic-release version
```

```
1.0.0
The next version is: 1.0.0! 🚀
No build command specified, skipping
```

Where are the versions stored?

- Releases tab on GitHub.
- `CHANGELOG.md` file.

Back to the package structure

```
poetry install
python
Python 3.11.9 | packaged by Anaconda, Inc. | (main, Apr 19 2024, 16:40:41)
[MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from mylibrary.hello import hello_world
>>> hello_world()
'Hello World'
```

Poetry installs locally the package in the virtual environment. You can import the package as if it was a library. You must be in the root directory of the package for python to recognize the package.

Adding a new library and updating the version

```python
# src/mylibrary/mylibrary.py
import pandas as pd
```

```
poetry add pandas
poetry install
python
Python 3.11.9 | packaged by Anaconda, Inc. | (main, Apr 19 2024, 16:40:41)
[MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from mylibrary.mylibrary import pd
>>> pd
<module 'pandas' from 'C:\\Users\\...\\__init__.py'>
>>>
```

Further reading

- Creating a python package
- GitHub cheat sheet
- GitHub documentation.