SWPP Practice Session #2
Git

2022 Sep 14

Reminder

- Team formation due: 9/13 6pm → Final announcement: 9/14
  - Team formation spreadsheet (link)
  - For those who does not form team until team formation due, TAs will form arbitrary teams among them
- Team project proposal due: 9/28 6pm
- HW2 due: 9/19 6pm
  - Add `swpp-tas` as collaborator
  - Push your codes

Please watch

https://github.com/swsnu/swppfall2022

- Watch the swppfall2022 repo!!
  - We frequently update HWs, announcements
  - Active & fruitful discussions on peer issues will grant you bonus (How to use Github issue board)

Today’s Objective

1. Learn the history of Version Control System (VCS)
2. Hacking Git – the inner workings
3. Typical Git cooperation workflow
4. Resolving merge conflicts
5. Useful Git commands
6. Tips about Github before diving into team project

Git

- Version Control System (VCS)
- 파일 변화를 시간에 따라 기록했다가, 나중에 특정 시점의 버전을 다시 꺼내올 수 있는 시스템

- Actually, Git is a DVCS (D for Distributed)
  - Differentiates from Local VCS(LVCS) & Centralized VCS(CVCS)
  - Are you interested in the details? *(Please say YES)*

- Functionality
  - Rollback/forth between each file versions
  - Rollback/forth between project versions
  - Compare diffs between versions
  - Recover on errors
  - and much more…

Fun fact.

Linus Torvalds and Linux community developed Git as an alternative VCS for Linux Kernel, which had to move away from its original VCS 'BitKeeper'

DVCS

Please note that this comparison highlights how the VCS evolved from time to time. It is true that there are variations among the applications in each category, and the latest ones have evolved a lot from there initial figure.

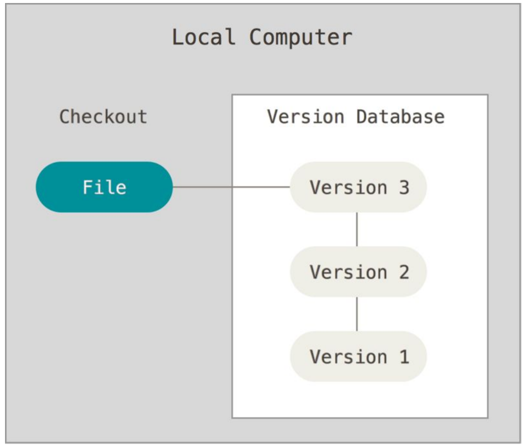
- LVCS
  - Simple local DB stores versions **for each file**
  - Switch between file versions by applying/detaching the patches (**single file diffs**)
  - Deficit
    - No simultaneous work among developers
    - No project level version control

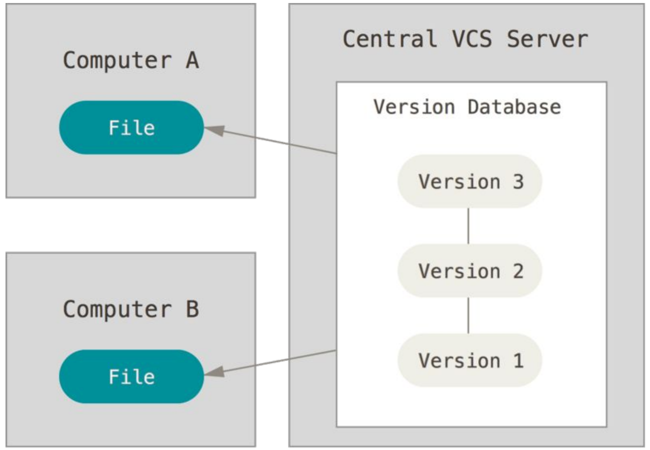
- CVCS
  - Central copy of the project on a server
  - Pull & Push mechanism
    - Client checkouts last snapshot of the file & Commit their changes to this central copy
    - **Changesets**: Cohesive group of changes to files
    - Commit == Push, since there is no concept of ‘local’ & ‘remote’

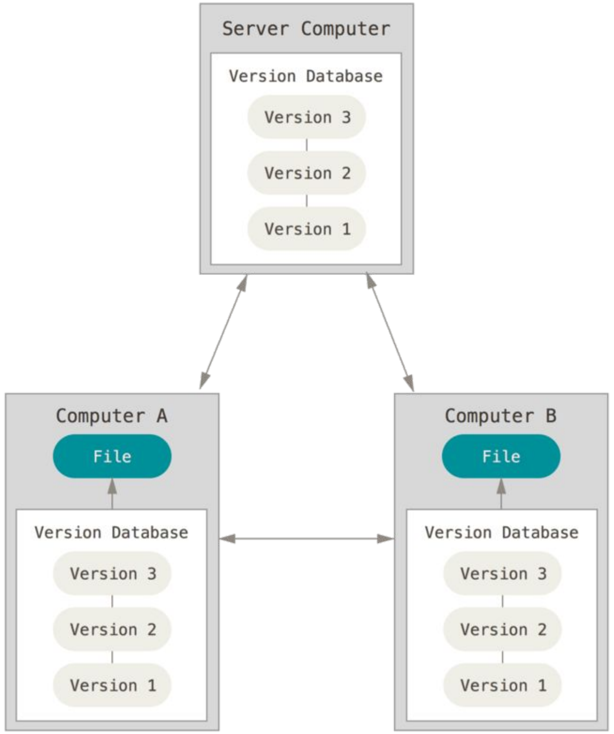
- DVCS
  - Client “clones” a copy of a repository ⇒ Has the full history of the project on HDD
    - Not a big deficit now, as modern HDDs are cheap
  - The concept of ‘local’ and ‘remote’. Local = repository in local; Remote = repository in server;
    - Now, offline commits (checking-in changes to local repo; or the “changeset”) is possible
    - Online push (checking-in changes to remote repo) does not need to be done on every commit

DVCS

- LVCS
- CVCS
- DVCS



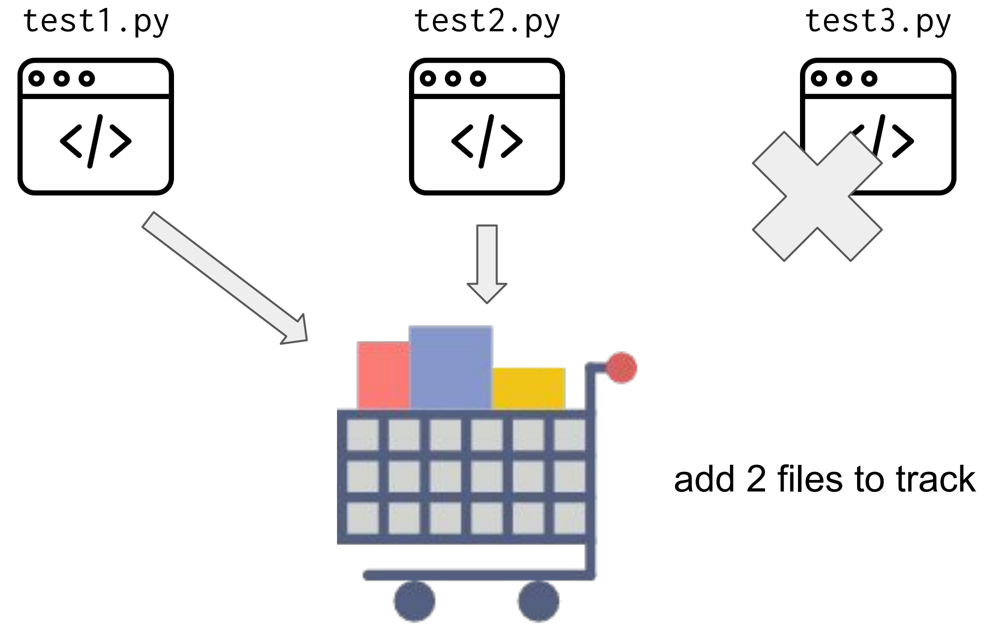




Before we start, I will assume you are familiar with git add, commit and push through HW1!

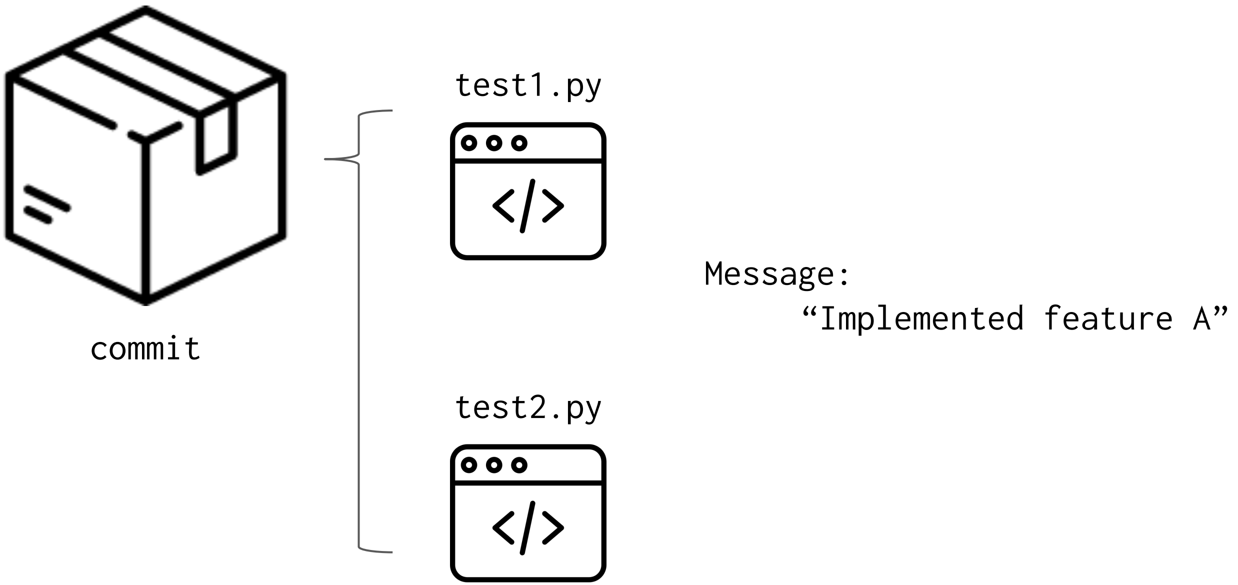
Quick Review (`git add`)

- add file to track (**`$ git add ${TARGET_FILE}`**)
  - ex) **`$ git add test1.py test2.py`**
- Track is like a shopping cart, where you put stuff that you want to buy



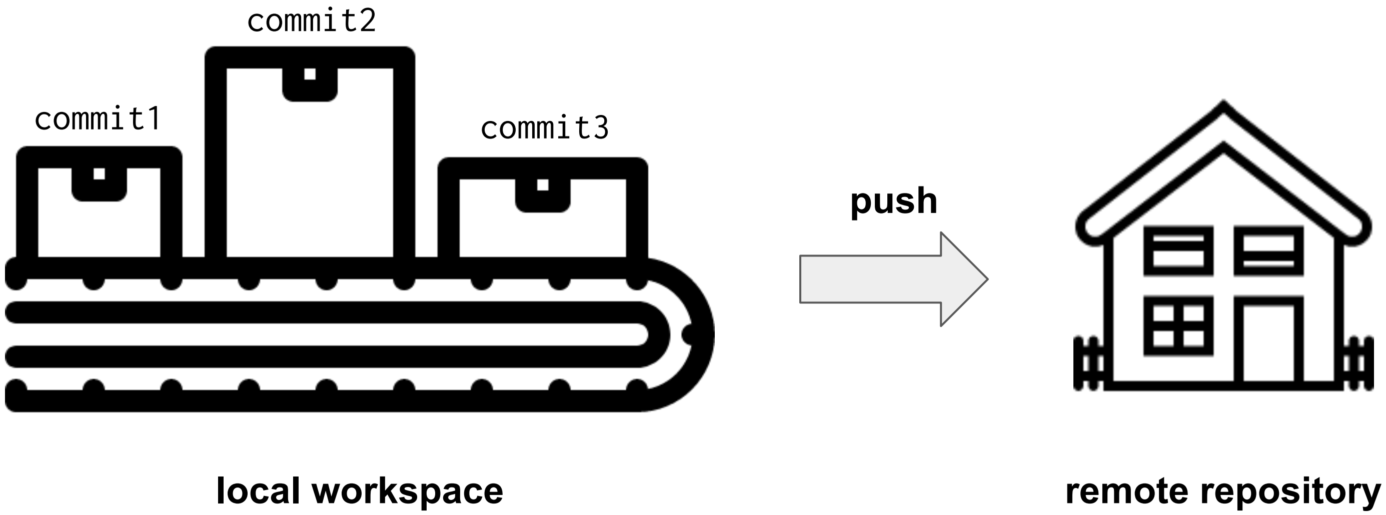
Quick Review (`git commit`)

- Commit is a ‘wrapping box’, in which there are stuffs that you’ve added to track
- **`$ git commit -m “commit message (wrapping box description)”`**



Quick Review (`git push`)

- After wrapping, we should bring them from market (local workspace) to our home (remote repository)
- **`$ git push ${REMOTE} ${BRANCH} (usually git push origin main)`**



Useful git command (`git status`)

- Shows you state of track
  - which changes have been tracked
  - which changes have not been tracked
  - which files are not being tracked by git

- **`$ git status`**

Inner workings of Git

Let's try some hands-on! *(Please say YAY)*

Beforehand, pull the docker image & run

```
docker pull snuspl/swpp:session2
docker run --rm -it --ipc=host --name session2 snuspl/swpp:session2 /bin/bash
```

Init an empty git repository & Make a file with some contents in it. Commit it.

```
mkdir temp && cd temp && git init
git config user.email "$your_email" && git config user.name "$github_id"
echo hello > hello.file && echo world > world.file
git add . & git commit -m "1st commit"
```

Everything is a hash

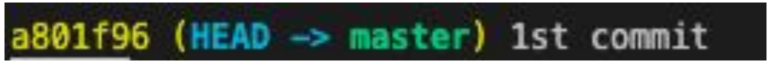
- Git refers to all commits by their SHA-1 hashes
- Git creates references to references in a tree-like structure to store and retrieve your data, and its metadata, as quickly and efficiently as possible
- .git directory would have been created after you initialized. Let's dissect the secret files underneath .git.

Dissecting the commit

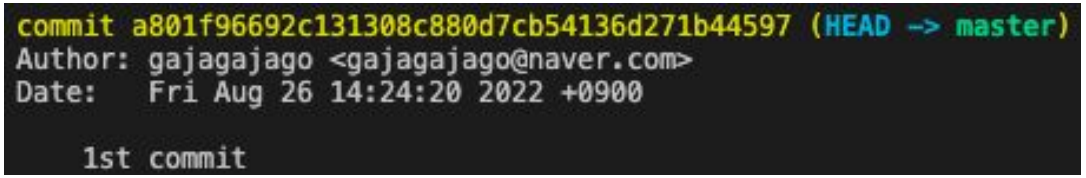
`git log –oneline` will show you the commit history like the following.

`git log` will show you the long hash.

Let's select the short hash.



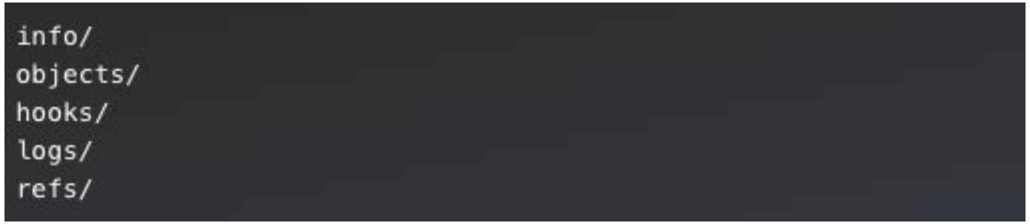
```
a801f96 (HEAD -> master) 1st commit
```



```
commit a801f96692c131308c880d7cb54136d271b44597 (HEAD -> master)
Author: gajagajago <gajagajago@naver.com>
Date:   Fri Aug 26 14:24:20 2022 +0900

    1st commit
```

Inner workings of Git (cont)

Navigate into .git directory of your repo (`cd .git`). You should, at minimum, see the following directories.

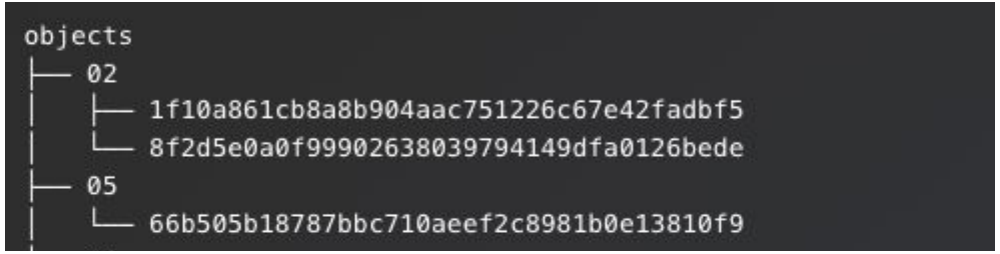


```
info/
objects/
hooks/
logs/
refs/
```

The directory you're interested in is the objects directory. In Git, the most common objects are:

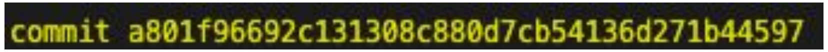
- **Commits**: Structures that hold metadata about your commit, as well as the pointers to the parent commit and the files underneath.
- **Trees**: Tree structures of all the files contained in a commit.
- **Blobs**: Compressed collections of files in the tree.

Start by navigating into the **objects** directory:



```
objects
├── 02
│   ├── 1f10a861cb8a8b904aac751226c67e42fadbf5
│   └── 8f2d5e0a0f99026380397941 49dfa0126bede
├── 05
│   └── 66b505b18787bbc710aeef2c8981b0e13810f9
```

Decompose that into a directory name and an object identifier: (for me,)

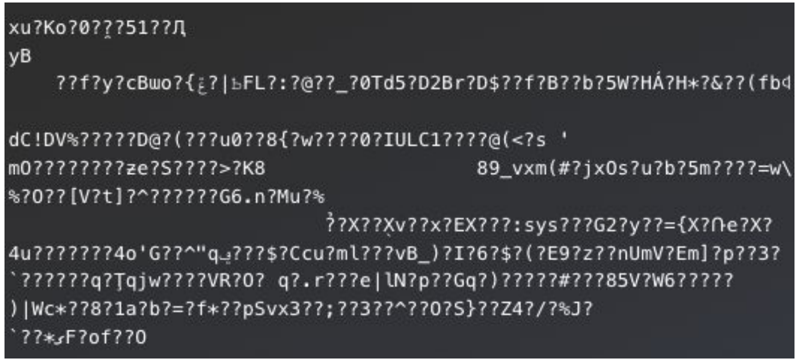


```
commit a801f96692c131308c880d7cb54136d271b44597
```

- **Directory**: a8
- **Object identifier**: 01f96692c131308c880d7cb54136d271b44597

Inner workings of Git (cont)

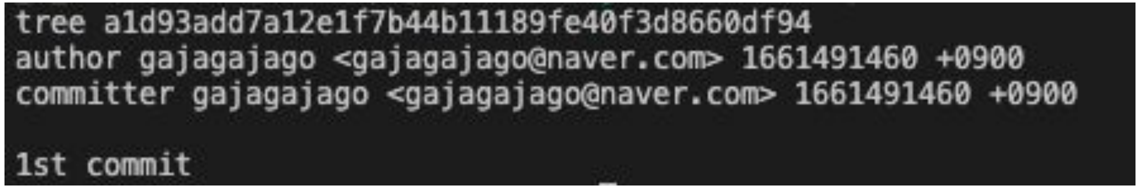
You won’t be able to look at this object directly, as it is compressed. Try `cat $object_file_name` and you’ll see the gibberish.



```
xu?Ko?0?̰??51??Л
yB
    ??f?y?cBшo?{¿?|ьFL?:?@??_?0Td5?D2Br?D$??f?B??b?5W?HÁ?H*?&??(fb◁

dC!DV%?????D@?(???u0??8{?w????0?IULC1????@(<?s '
mO?????????ƶe?S????>?K8                        89_vxm(#?jxOs?u?b?5m????=w\
%?O??[V?t]?^??????G6.n?Mu?%
                                ??X??X̧v??x?EX???:sys???G2?y??={X?Ռe?X?
4u???????4o'G??^"qڍ???$?Ccu?ml???vB_)?I?6?$?(?E9?z??nUmV?Em]?p??3?
`??????q?Ţqjw????VR?O? q?.r???e|lN?p??Gq?)?????#???85V?W6?????
)|Wc*??8?1a?b?=?f*??pSvx3??;??3??^??O?S}??Z4?/?%J?
`??*ىF?of??O
```

Viewing Git objects

`git cat-file -p $short_hash`



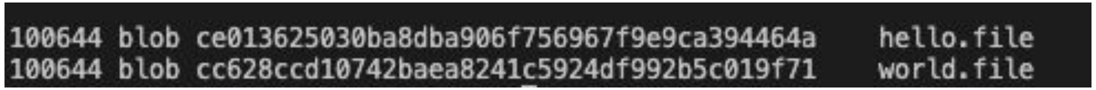
```
tree a1d93add7a12e1f7b44b11189fe40f3d8660df94
author gajagajago <gajagajago@naver.com> 1661491460 +0900
committer gajagajago <gajagajago@naver.com> 1661491460 +0900

1st commit
```

What you are interested in is the **tree** hash

Inner workings of Git (cont)

The tree object

It is a pointer to another object that holds the collection of files for this commit.
So execute `git cat-file -p $tree_hash` to see what's inside that object.



```
100644 blob ce013625030ba8dba906f756967f9e9ca394464a    hello.file
100644 blob cc628ccd10742baea8241c5924df992b5c019f71    world.file
```

Looks a lot like the working tree of your project, doesn't it? That's because that's precisely what this is: a compressed representation of your file structure inside the repository. Now, again, this object is simply a pointer to other objects.

Let's keep unwrapping objects as you go.
`git cat-file -p $world.file_hash`
You will see the content of the world.file. Please copy this command to somewhere else, since we will use it later again.

Now, let's make a commit to the world.file. `cd` back to your repo.
`echo swpp > world.file`
`git add . && git commit -m "second commit"`
Now let's follow the step again. Check the short hash of this commit, and keep unwrapping objects until you reach world.file

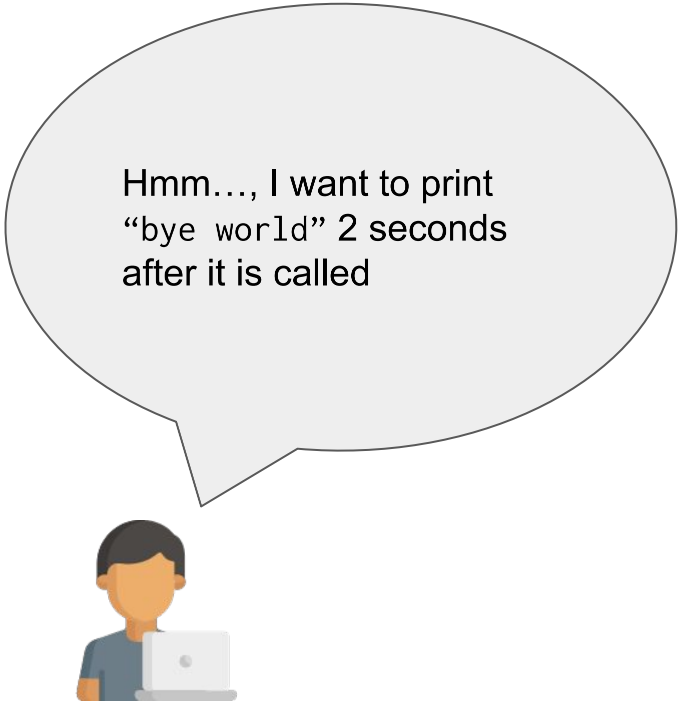
You will see the content of the swpp.file changed. Then what if you follow the first unwrapping step of the 1st commit?
The file content is not changed! This is how Git manages versions. Now, you can imagine that you can just follow different pointers to move between commit versions.

Typical Workflow

Typical Workflow

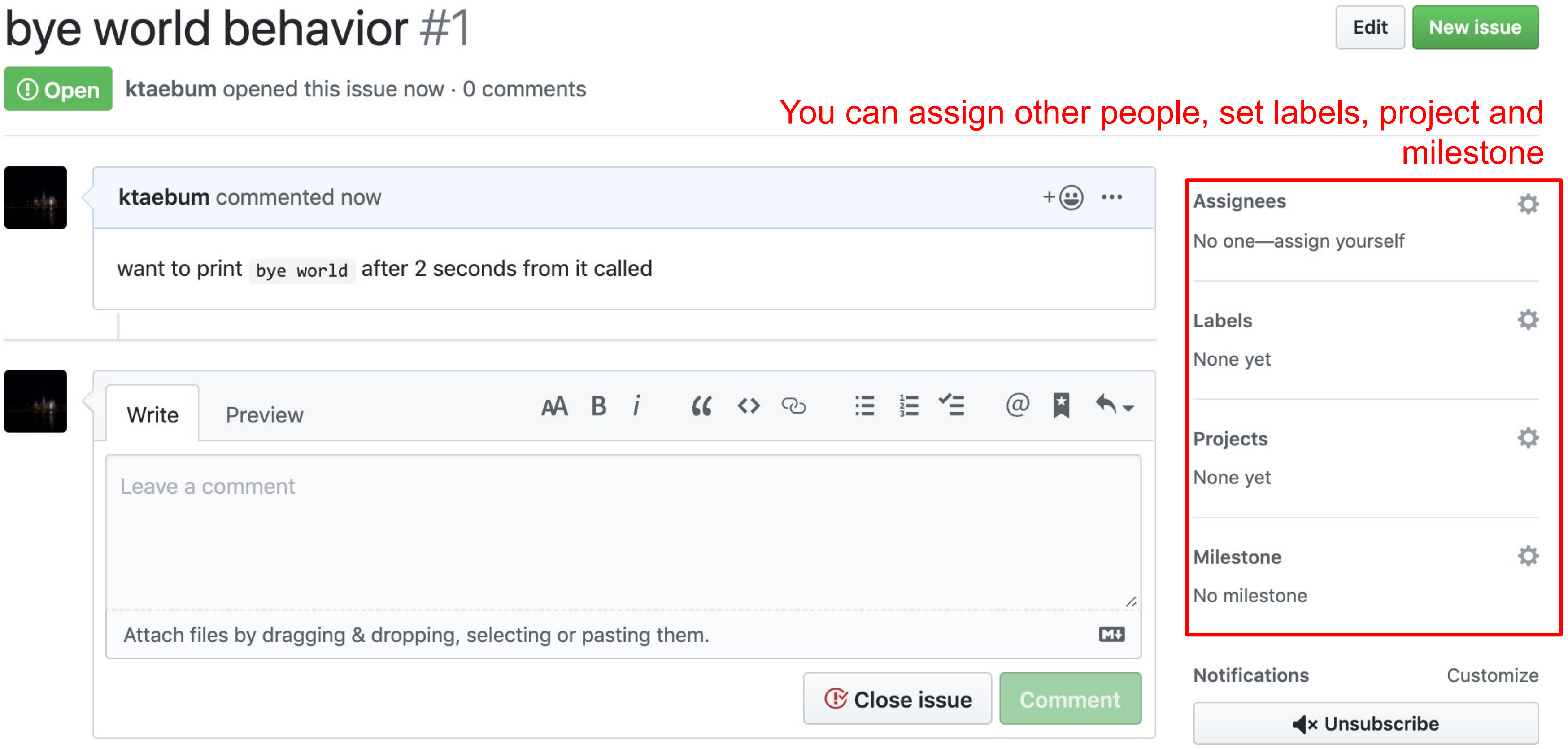
- Assume a simple Python script exists:

```
# world.py file

def hello_world():
  print(“hello world”)

def bye_world():
  print(“bye world”)

if __name__ == “__main__”:
  hello_world()
  bye_world()
```


Hmm…, I want to print “bye world” 2 seconds after it is called

Typical Workflow

- Create issue in github repo



bye world behavior #1

Edit
New issue

Open ktaebum opened this issue now · 0 comments

You can assign other people, set labels, project and milestone

ktaebum commented now

want to print `bye world` after 2 seconds from it called

Write Preview

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Close issue Comment

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

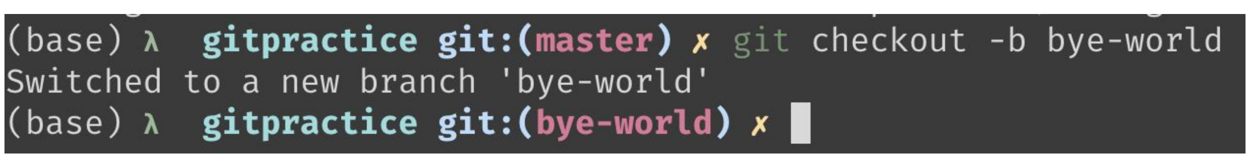
Milestone
No milestone

Notifications Customize

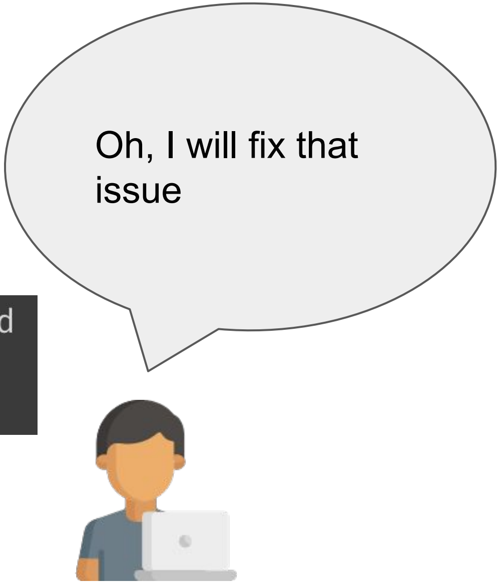
Unsubscribe

Typical Workflow

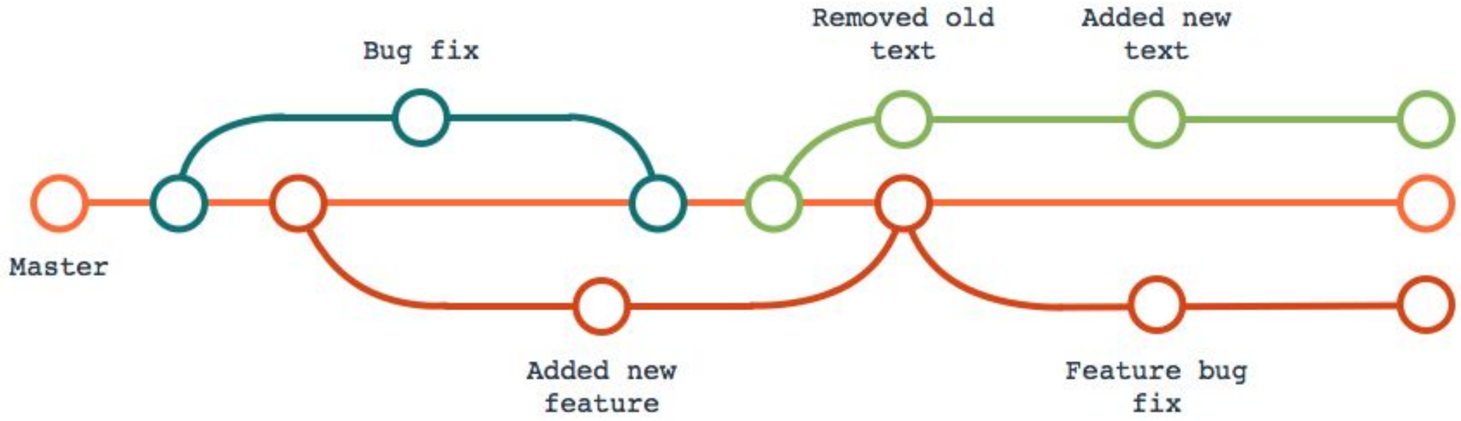
- Assignee (or someone else) sees that issue
- Branch out from the current branch (assume current == main branch)
  - `$ git branch bye-world && git checkout bye-world`
- Or you can do it at once as
  - `$ git checkout -b bye-world (-b == branch)`



```
(base) λ  gitpractice git:(master) ✗ git checkout -b bye-world
Switched to a new branch 'bye-world'
(base) λ  gitpractice git:(bye-world) ✗ █
```



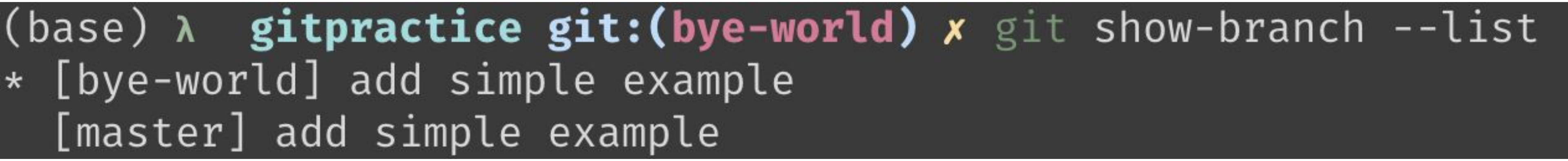
What is branch exactly? (1/2)



- A single branch represents an independent line of development.
- Circles represent commits.
- Main branch is the default branch, usually the mainstream of the project.
- You can branch out from the current branch, where you can add commits while not affecting the main branch.
    - Suppose you want to add codes to add a new feature, but you don’t want to add it to the project’s mainstream until you are sure it is working

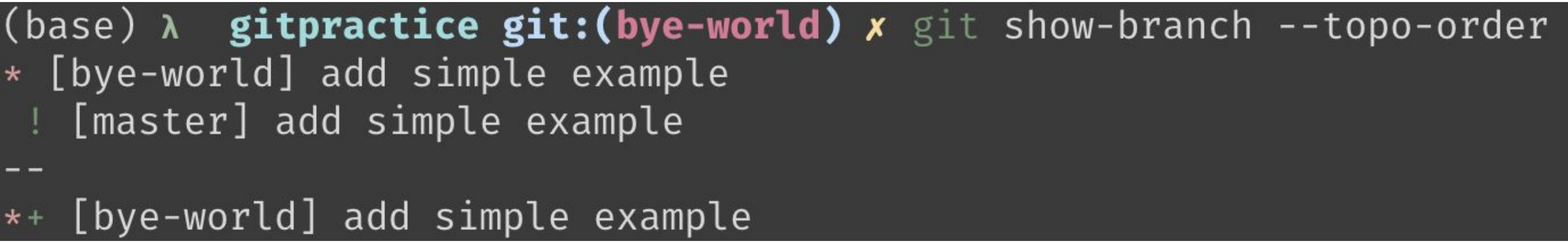
What is branch exactly? (2/2)

- See branch list using **`git show-branch --list`**



```
(base) λ  gitpractice git:(bye-world) ✗ git show-branch --list
* [bye-world] add simple example
  [master] add simple example
```

- See branch in topological tree order using **`git show-branch --topo-order`**



```
(base) λ  gitpractice git:(bye-world) ✗ git show-branch --topo-order
* [bye-world] add simple example
 ! [master] add simple example
--
*+ [bye-world] add simple example
```

Typical Workflow

- Now let’s modify `world.py` file

```
# world.py file
import time

def hello_world():
  print(“hello world”)

def bye_world():
  print(“bye world” )

if __name__ == “__main__”:
  hello_world()
  time.sleep(2)
  bye_world()
```

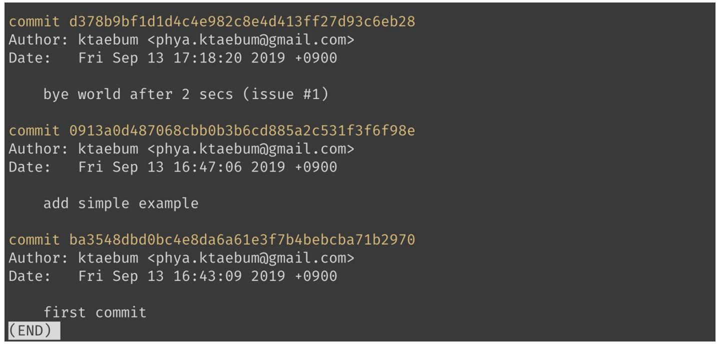
Typical Workflow

- Add file to track, and push it to the remote branch

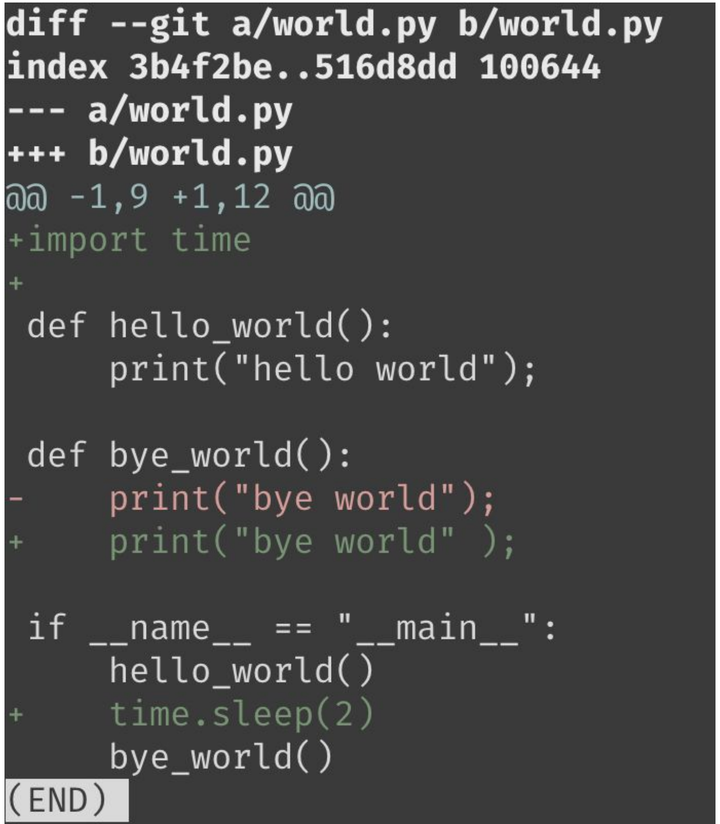
- `$ git status`
- **`$ git add world.py`**
- `$ git status`
- **`$ git commit -m ‘bye world after 2 secs (issue #1)’`**
  - Usually, we mark corresponding issue number in commit message
- `$ git status`
- `$ git log`
- **`$ git push origin bye-world`**

Typical Workflow

- See differences between the two branches with **`git diff`**
- **`$ git diff main`**
- `git diff` also works for specific commit log
- See commit logs via **`$ git log`**



```
commit d378b9bf1d1d4c4e982c8e4d413ff27d93c6eb28
Author: ktaebum <phya.ktaebum@gmail.com>
Date:   Fri Sep 13 17:18:20 2019 +0900

    bye world after 2 secs (issue #1)

commit 0913a0d487068cbb0b3b6cd885a2c531f3f6f98e
Author: ktaebum <phya.ktaebum@gmail.com>
Date:   Fri Sep 13 16:47:06 2019 +0900

    add simple example

commit ba3548dbd0bc4e8da6a61e3f7b4bebcba71b2970
Author: ktaebum <phya.ktaebum@gmail.com>
Date:   Fri Sep 13 16:43:09 2019 +0900

    first commit
(END)
```



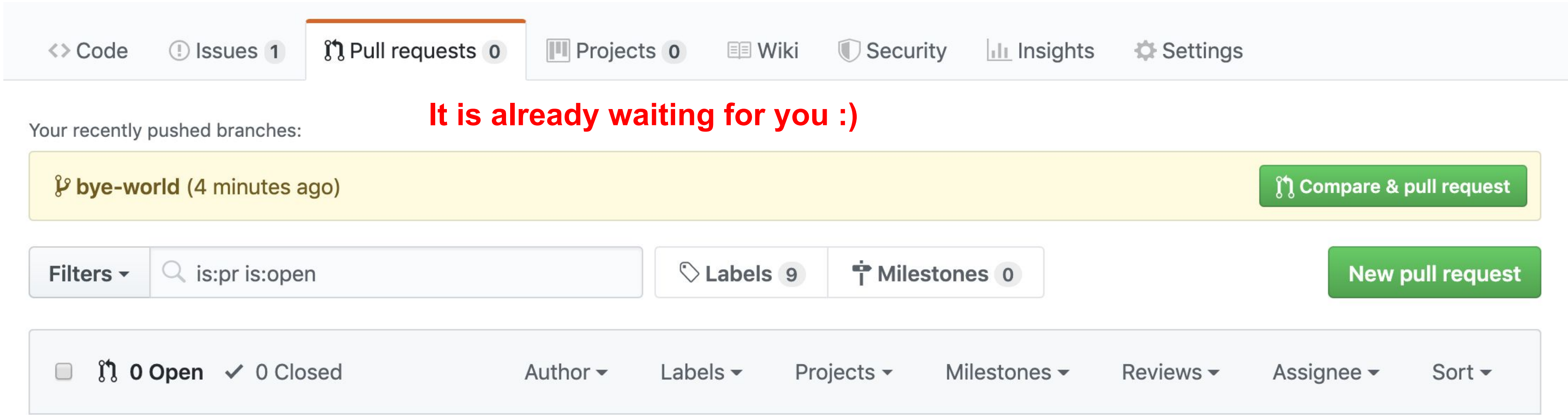
```
diff --git a/world.py b/world.py
index 3b4f2be..516d8dd 100644
--- a/world.py
+++ b/world.py
@@ -1,9 +1,12 @@
+import time
+
 def hello_world():
     print("hello world");

 def bye_world():
-    print("bye world");
+    print("bye world" );

 if __name__ == "__main__":
     hello_world()
+    time.sleep(2)
     bye_world()
(END)
```

Typical Workflow

- Now, you want to add your changes to the main branch!
  - Option 1: Push directly to main branch
  - Option 2: Create a PR

Typical Workflow

- Let’s create a PR (pull request)



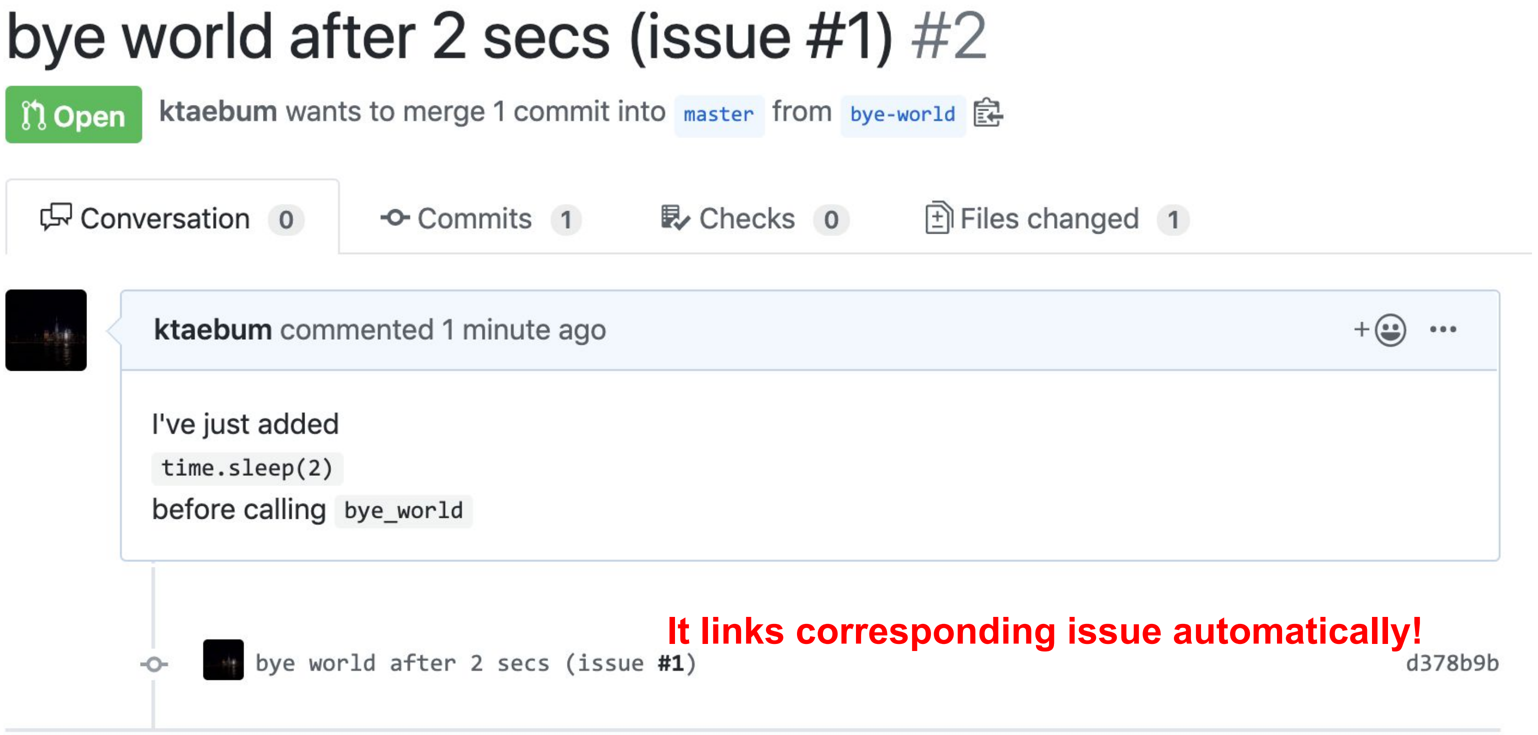
Your recently pushed branches:

bye-world (4 minutes ago)

is:pr is:open

It is already waiting for you :)

Typical Workflow


bye world after 2 secs (issue #1) #2

Open ktaebum wants to merge 1 commit into `master` from `bye-world`

ktaebum commented 1 minute ago

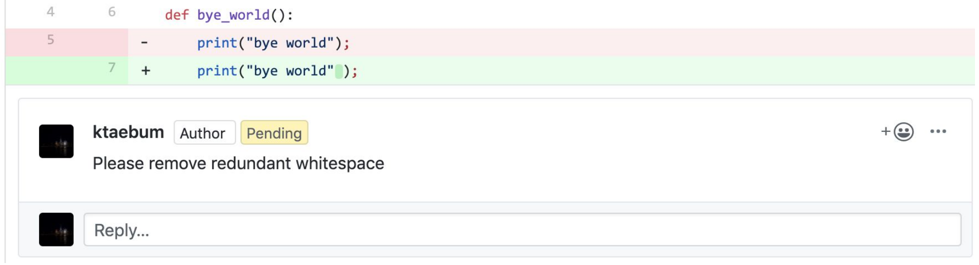
I've just added
`time.sleep(2)`
before calling `bye_world`

```
bye world after 2 secs (issue #1)    d378b9b
```

It links corresponding issue automatically!

Typical Workflow

- Now, reviewers (maybe your teammate) review your code
- Could be minor style issue (actually, not minor)



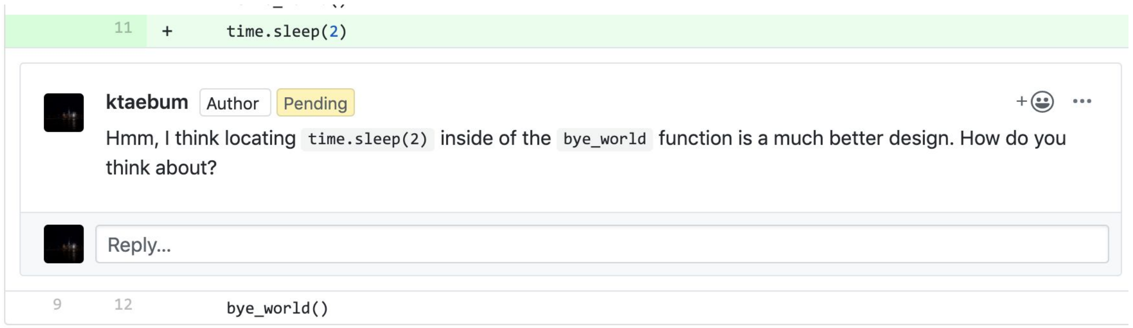
```
 def bye_world():
-     print("bye world");
+     print("bye world" );
```

ktaebum Author Pending

Please remove redundant whitespace

Reply...

- Could be implementation comment



```
+     time.sleep(2)
```

ktaebum Author Pending

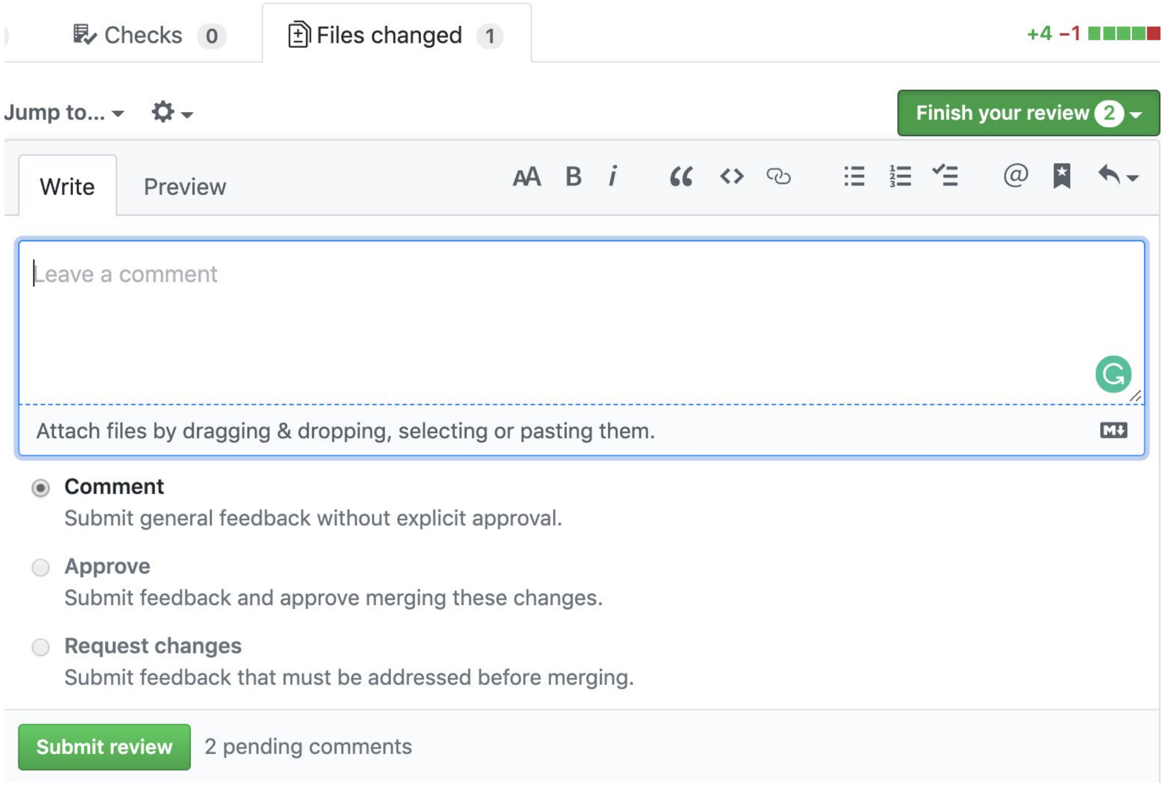
Hmm, I think locating `time.sleep(2)` inside of the `bye_world` function is a much better design. How do you think about?

Reply...

```
      bye_world()
```

Typical Workflow

- Now reviewers submit their reviews



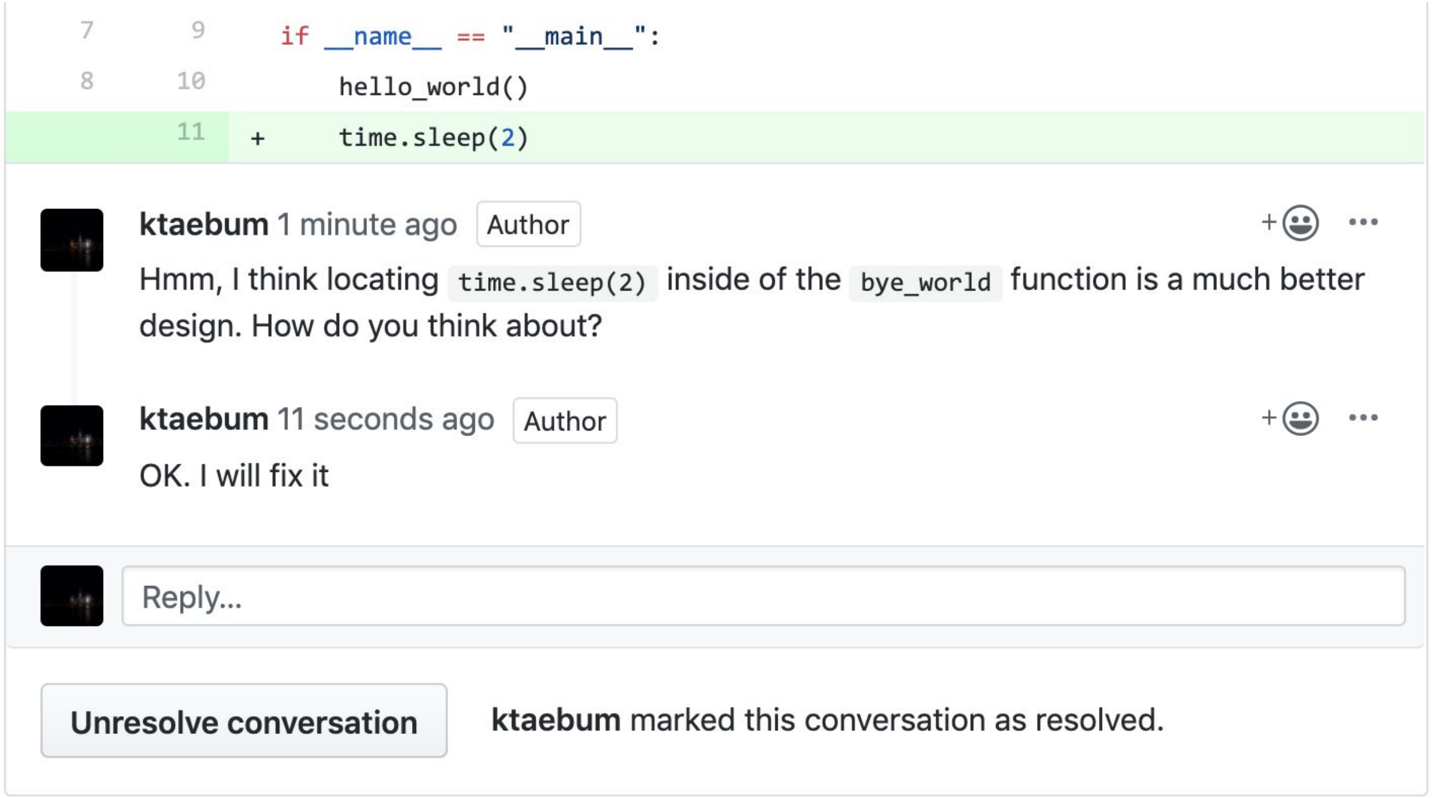
[✓ Comment] Submit general feedback without explicit approval.

Approve Submit feedback and approve merging these changes.

Request changes Submit feedback that must be addressed before merging.

Typical Workflow

- You can resolve the review after addressing the issue or leaving comments



```
if __name__ == "__main__":
    hello_world()
+   time.sleep(2)
```

ktaebum 1 minute ago Author

Hmm, I think locating `time.sleep(2)` inside of the `bye_world` function is a much better design. How do you think about?

ktaebum 11 seconds ago Author

OK. I will fix it

ktaebum marked this conversation as resolved.

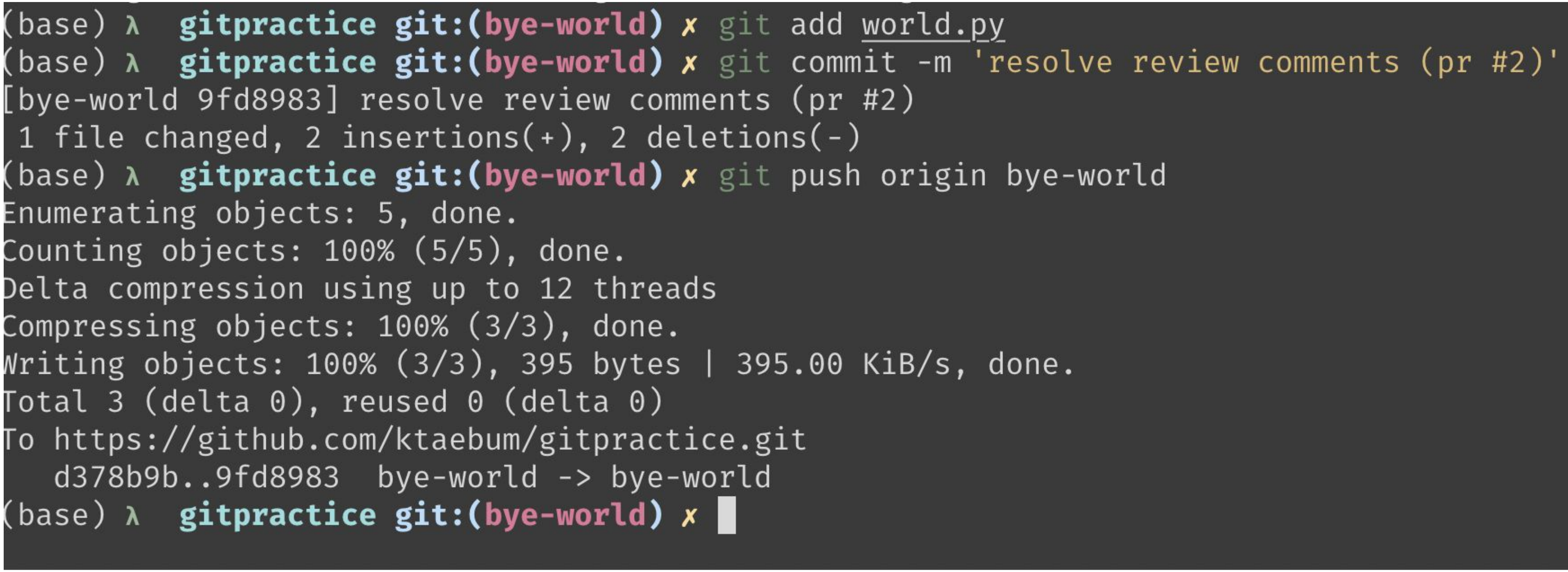
Typical Workflow

- Now, let’s fix the file again as the reviewer requested

```
# world.py file
import time

def hello_world():
  print(“hello world”)

def bye_world():
  time.sleep(2)
  print(“bye world”)

if __name__ == “__main__”:
  hello_world()
  bye_world()
```

Typical Workflow

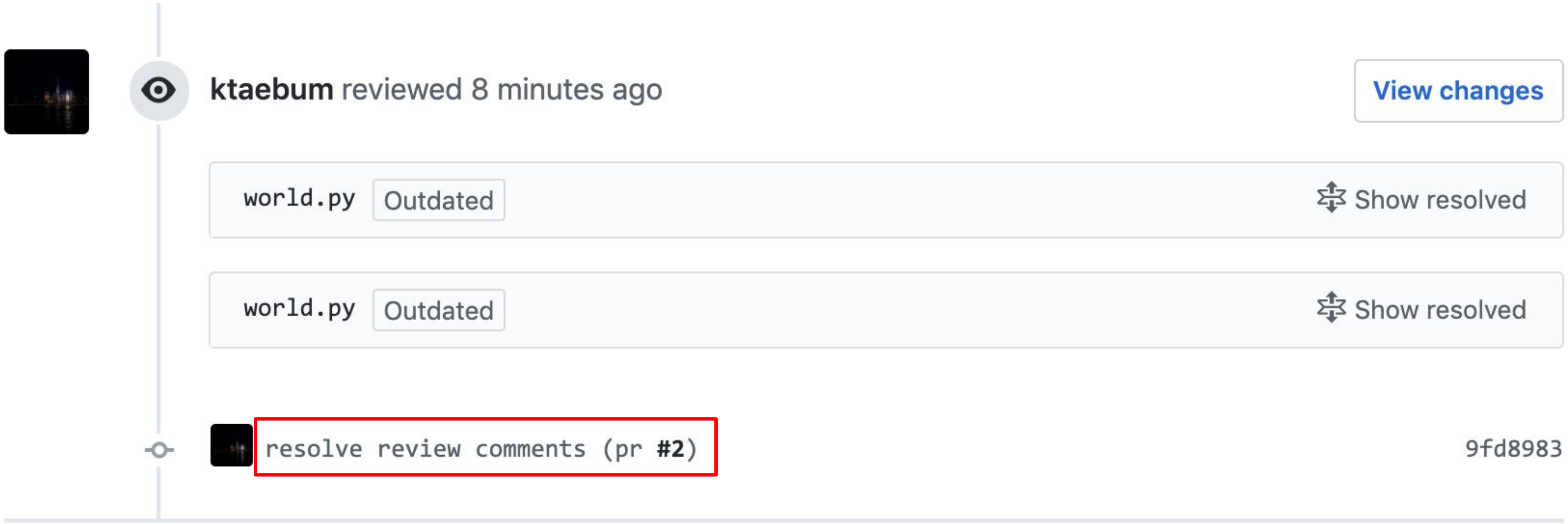
- Again, push your local changes to remote: add -> commit -> push



```
(base) λ  gitpractice git:(bye-world) ✗ git add world.py
(base) λ  gitpractice git:(bye-world) ✗ git commit -m 'resolve review comments (pr #2)'
[bye-world 9fd8983] resolve review comments (pr #2)
 1 file changed, 2 insertions(+), 2 deletions(-)
(base) λ  gitpractice git:(bye-world) ✗ git push origin bye-world
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 395 bytes | 395.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ktaebum/gitpractice.git
   d378b9b..9fd8983  bye-world -> bye-world
(base) λ  gitpractice git:(bye-world) ✗ 
```

Typical Workflow

- When you go back to pull request, you can see that your current commit is stacked



ktaebum reviewed 8 minutes ago

View changes

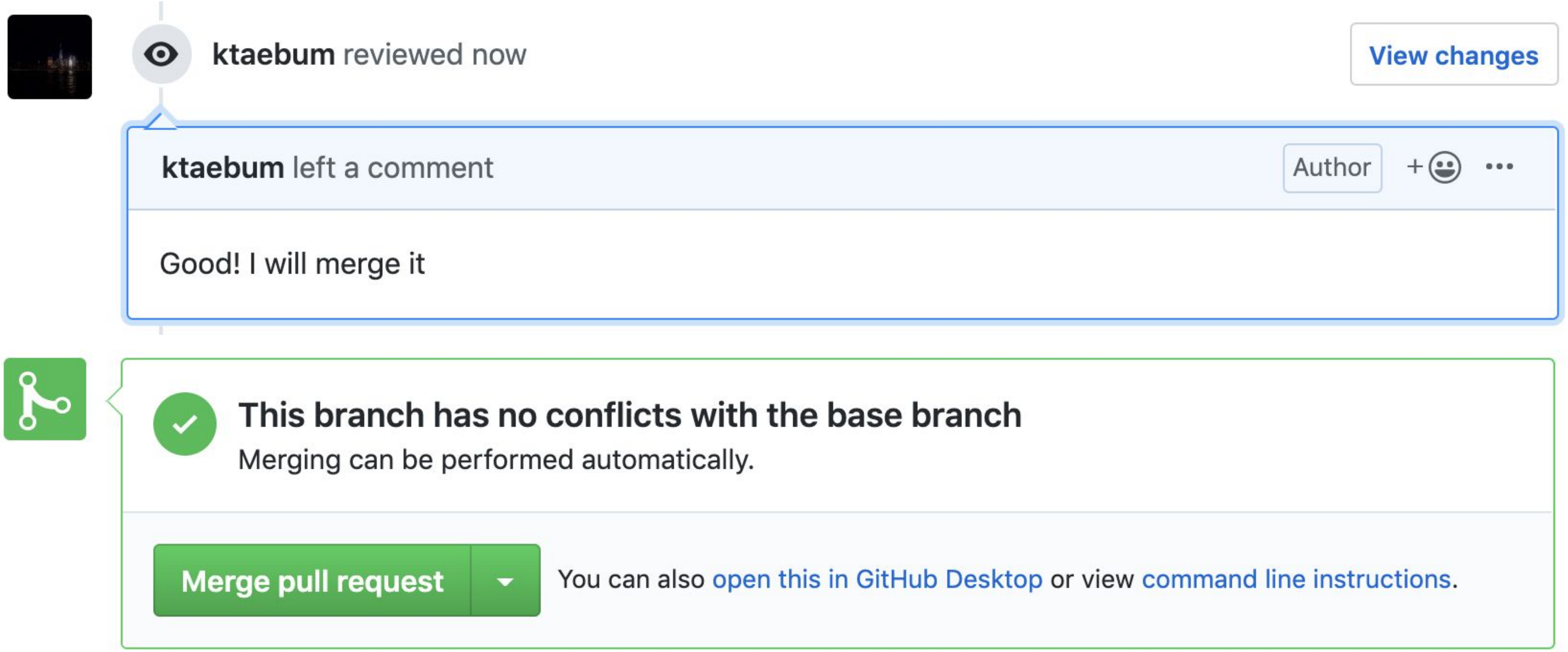
world.py Outdated Show resolved

world.py Outdated Show resolved

resolve review comments (pr **#2**) 9fd8983

Typical Workflow

- If reviewers are satisfied with the changes, they will merge it :)


ktaebum reviewed now

ktaebum left a comment

Good! I will merge it

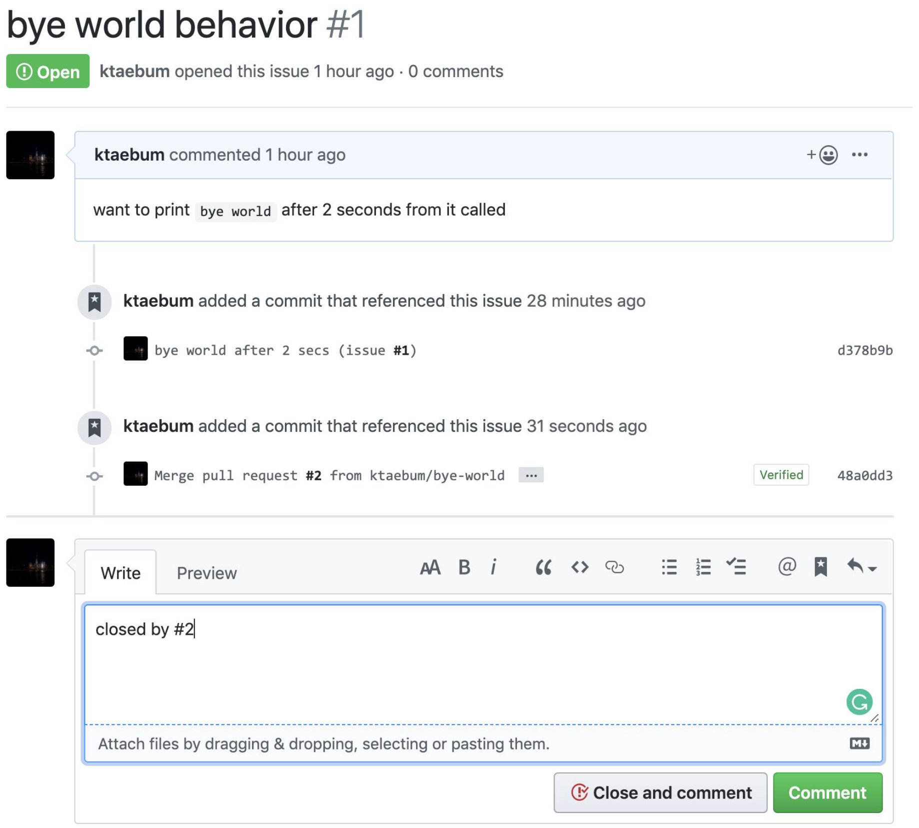
This branch has no conflicts with the base branch

Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

Typical Workflow

- Now, since the issue is addressed, it can be closed



bye world behavior #1

Open ktaebum opened this issue 1 hour ago · 0 comments

ktaebum commented 1 hour ago

want to print `bye world` after 2 seconds from it called

ktaebum added a commit that referenced this issue 28 minutes ago

```
bye world after 2 secs (issue #1)                    d378b9b
```

ktaebum added a commit that referenced this issue 31 seconds ago

```
Merge pull request #2 from ktaebum/bye-world …      Verified   48a0dd3
```

closed by #2

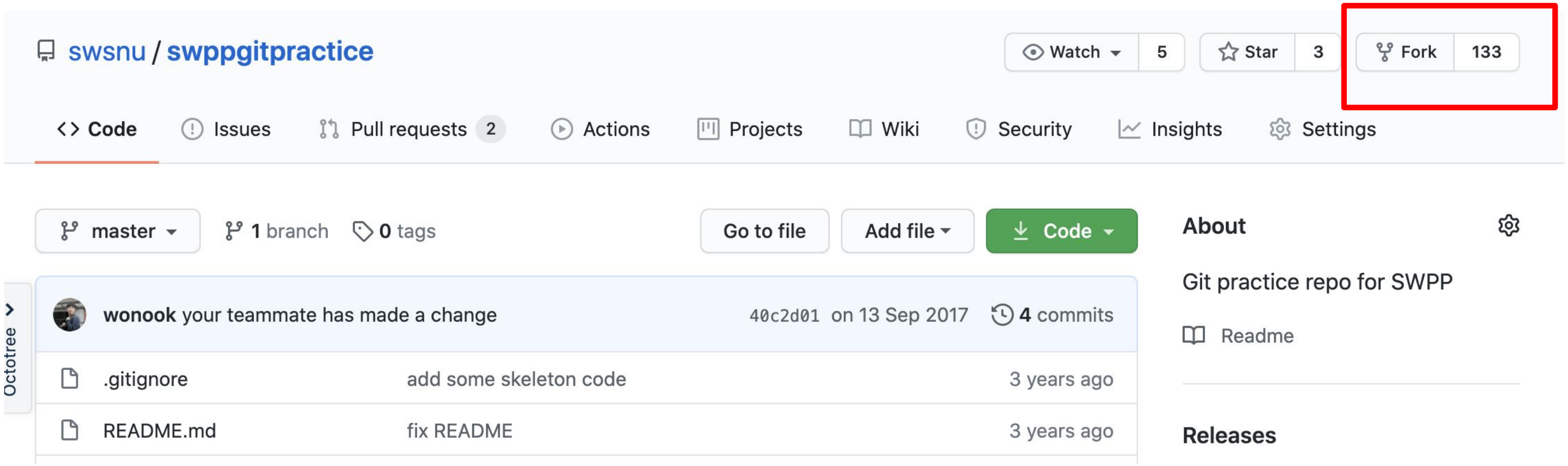
Attach files by dragging & dropping, selecting or pasting them.
Close and comment
Comment

Typical Workflow

- Now that we have changes to the remote main, we can update (sync) the local main branch with remote main
- **`$ git checkout main`** `# move to main branch`
- **`$ git pull origin main`** `# pull gets contents from remote branch`

Typical Workflow

- For repositories which you do not have authority to push (e.g. other people's repository) you can also create pull request:
  a. Fork target repository (forked repository works like separate branch)
  b. Push your changes into your forked repository
  c. Create pull request to original repository based on your forked repository


swsnu / swppgitpractice

wonook your teammate has made a change 40c2d01 on 13 Sep 2017 4 commits

.gitignore	add some skeleton code	3 years ago
README.md	fix README	3 years ago

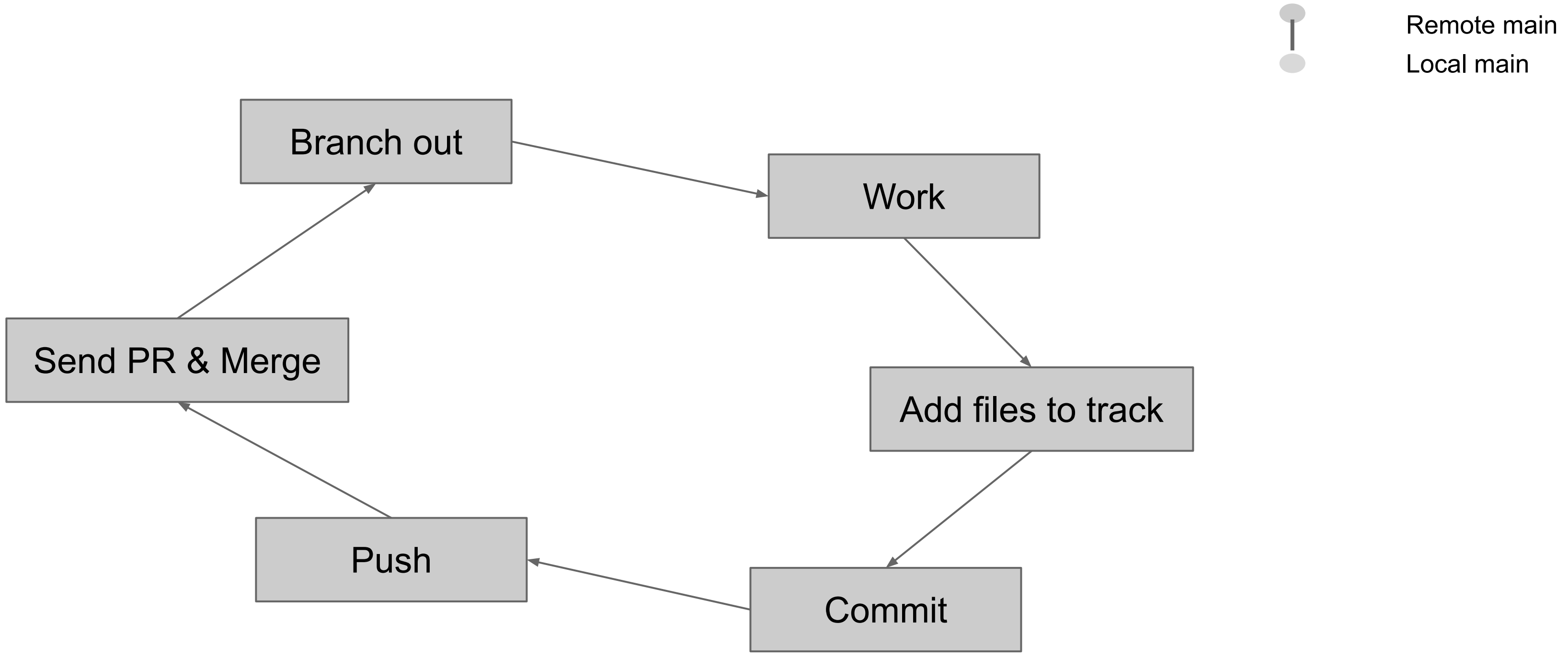
About

Git practice repo for SWPP

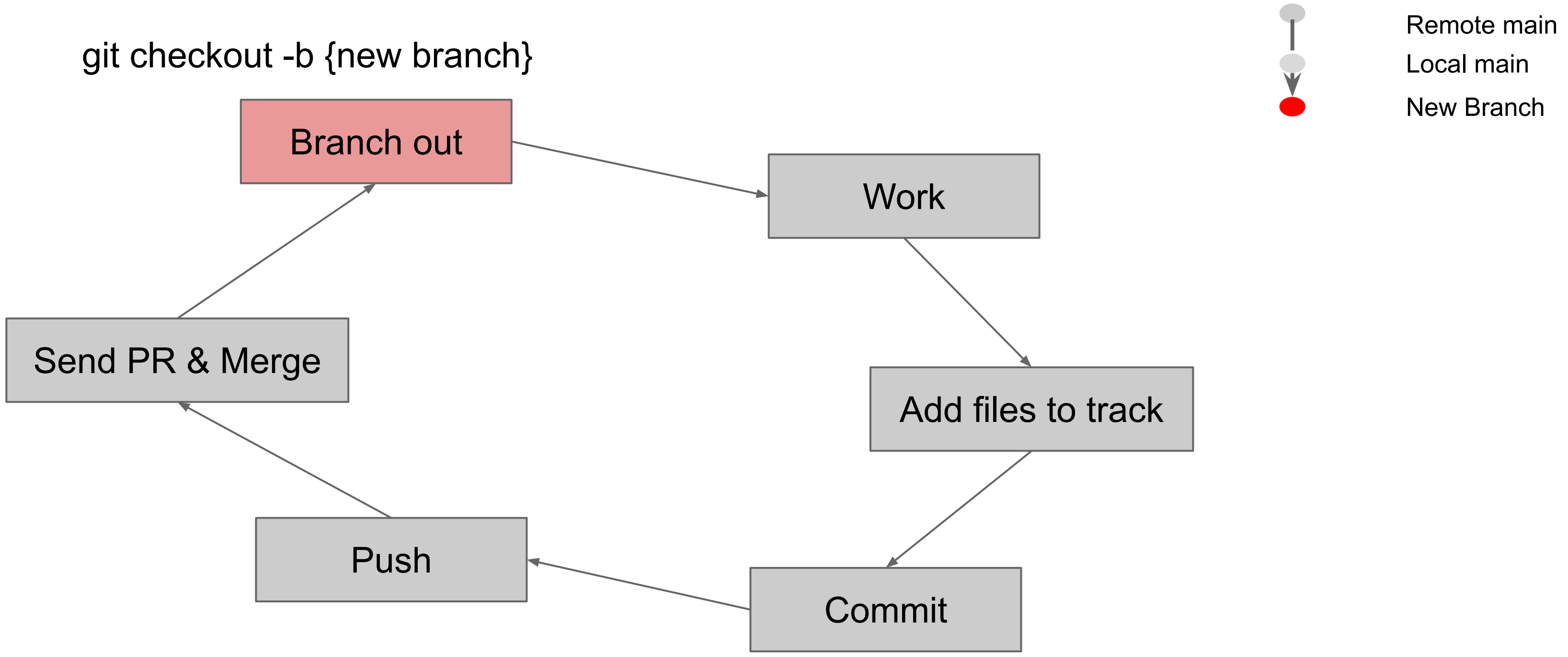
Readme

Releases

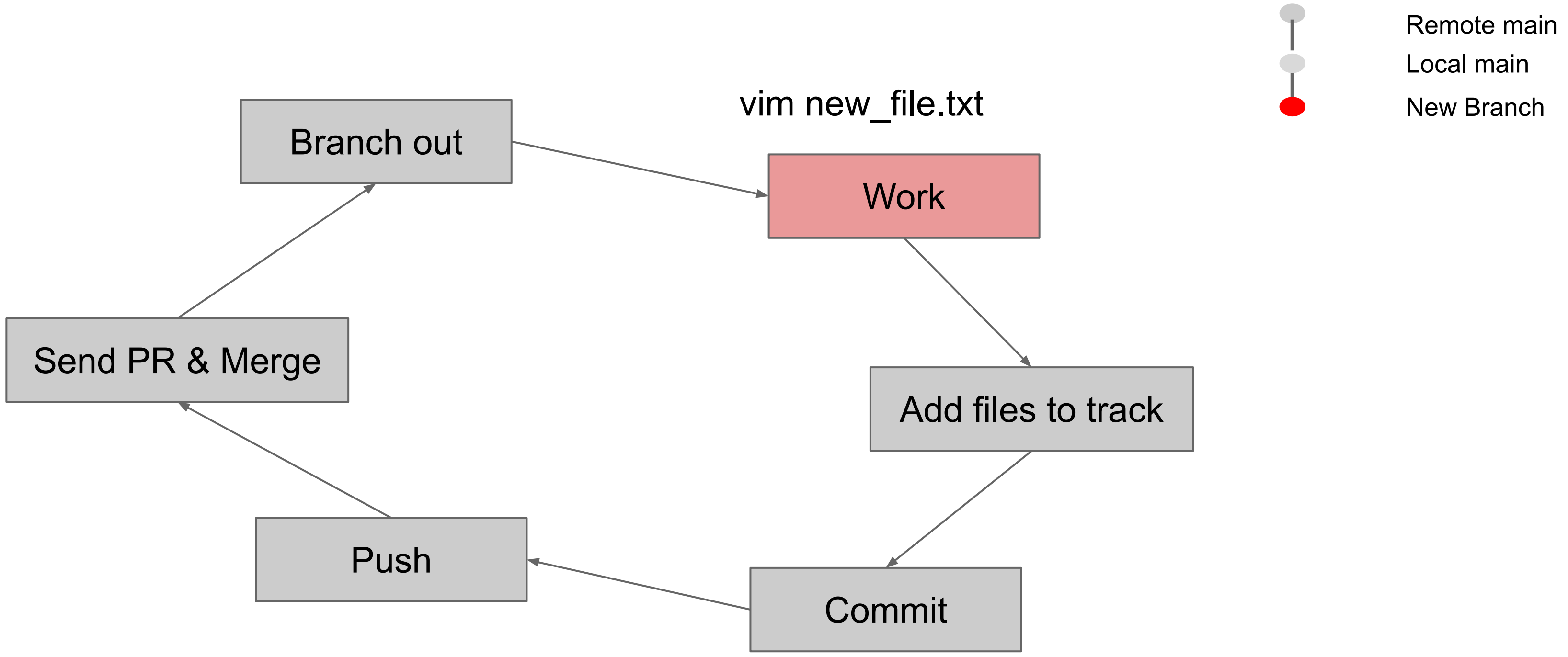
Typical Workflow (Review)



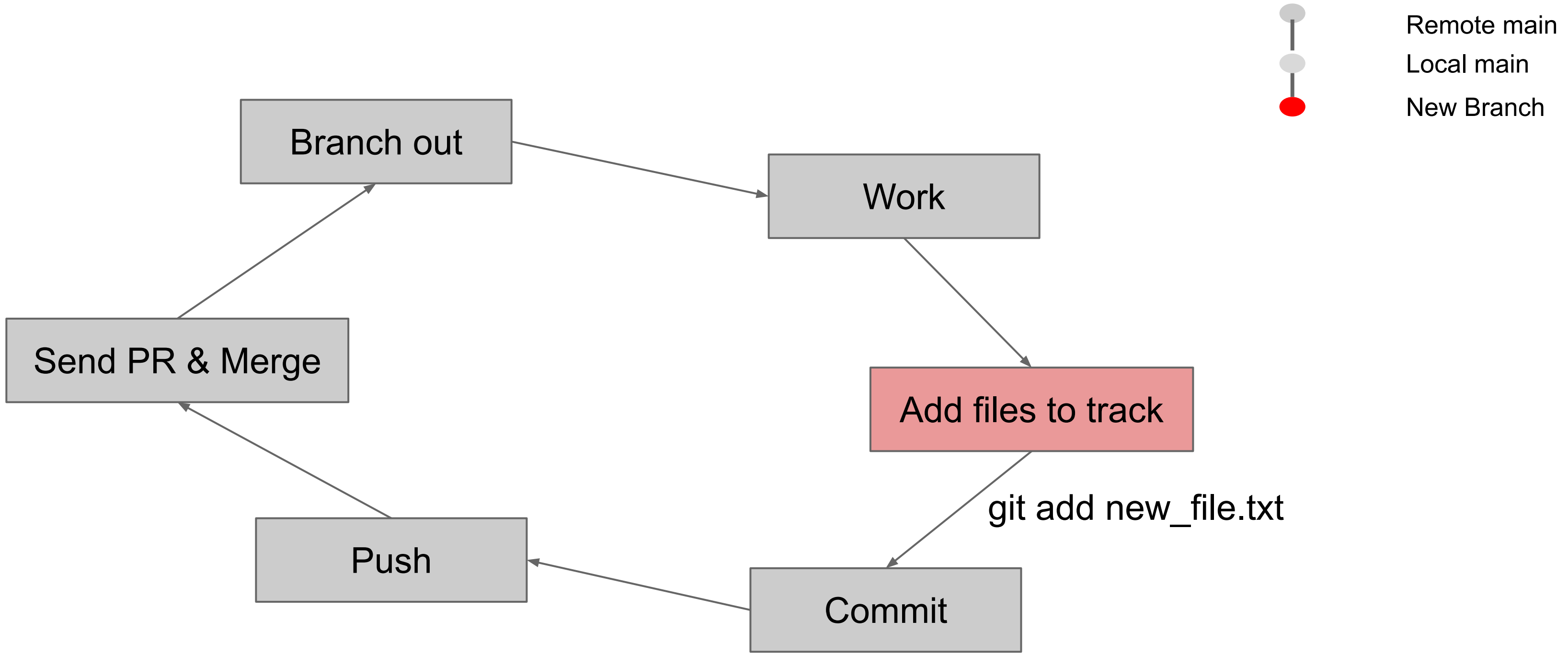
Typical Workflow (Review)



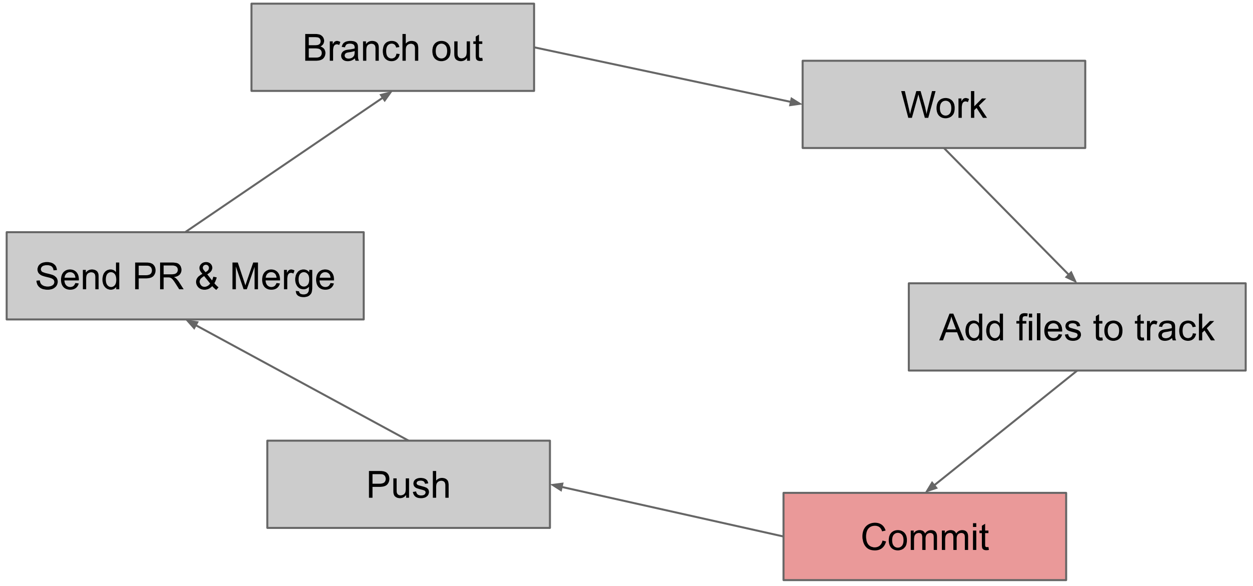
Typical Workflow (Review)

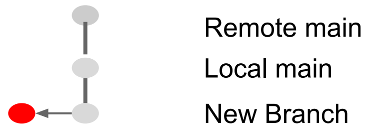


Typical Workflow (Review)



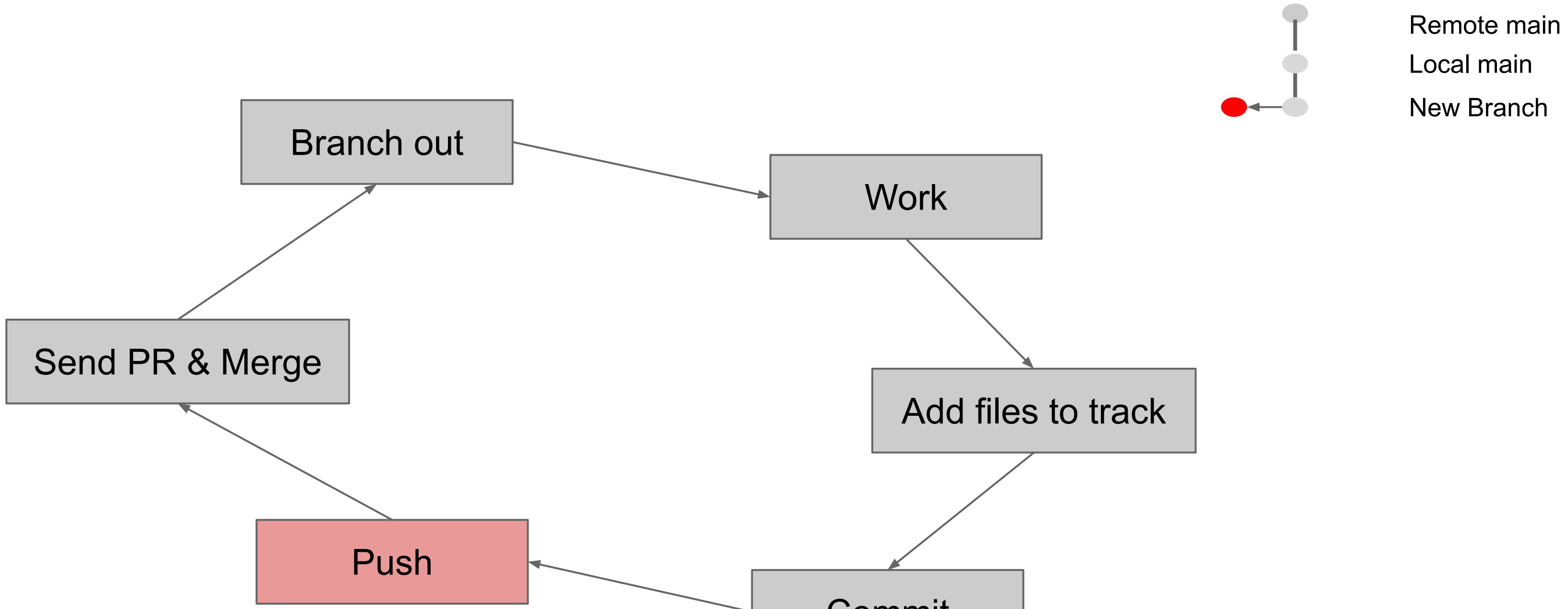
Typical Workflow (Review)





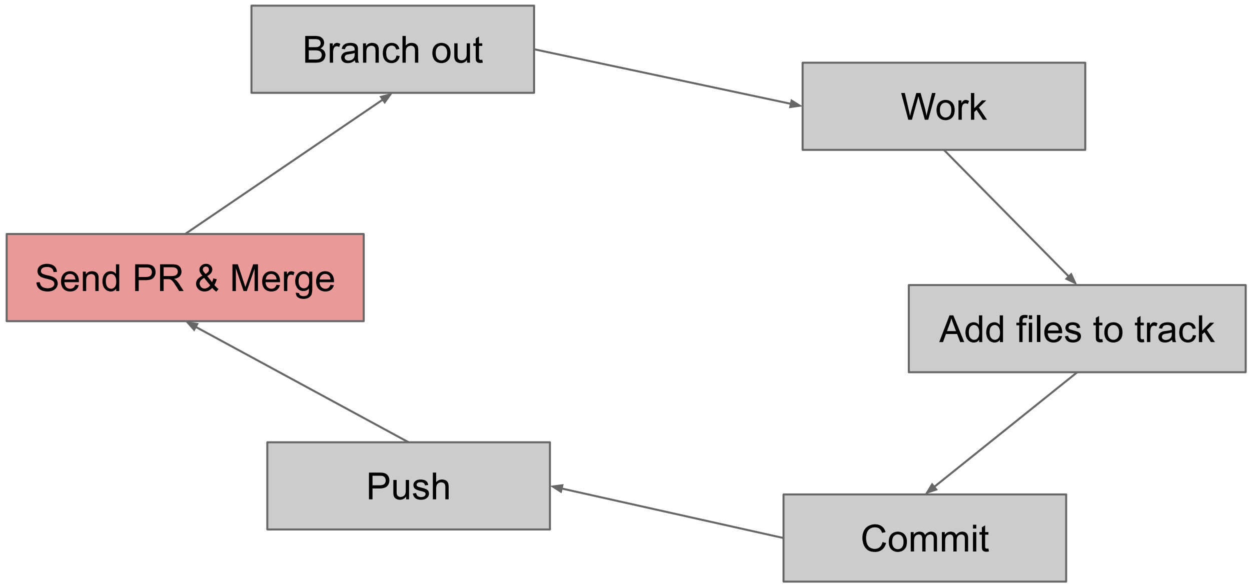
git commit

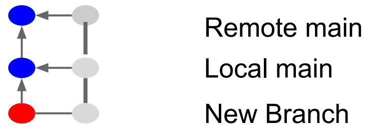
Typical Workflow (Review)



git push

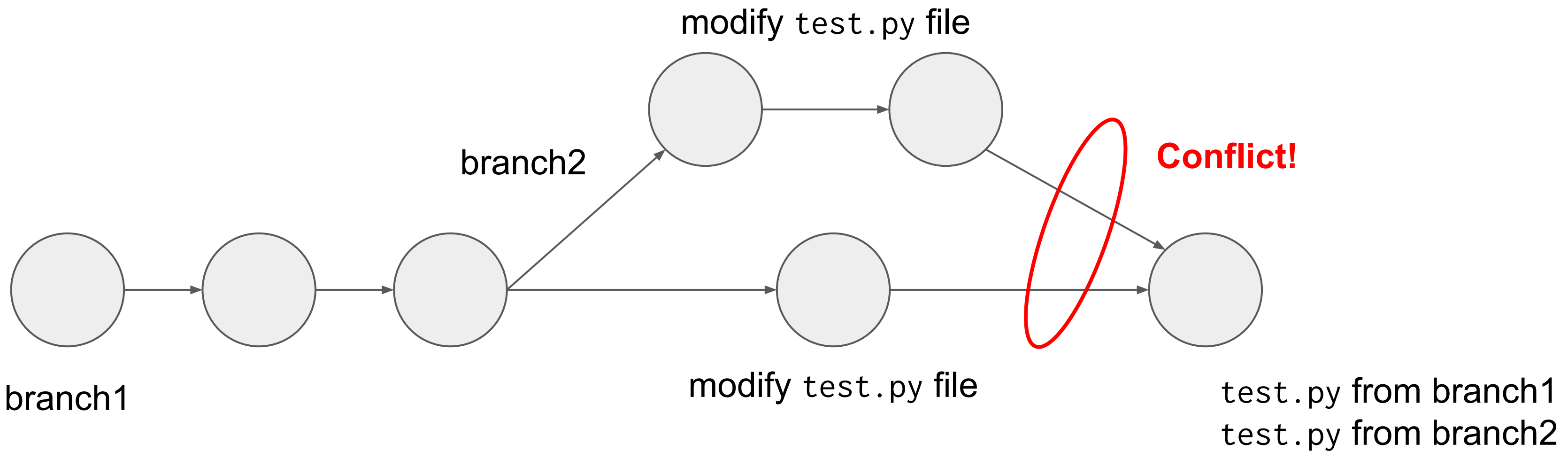
Typical Workflow (Review)





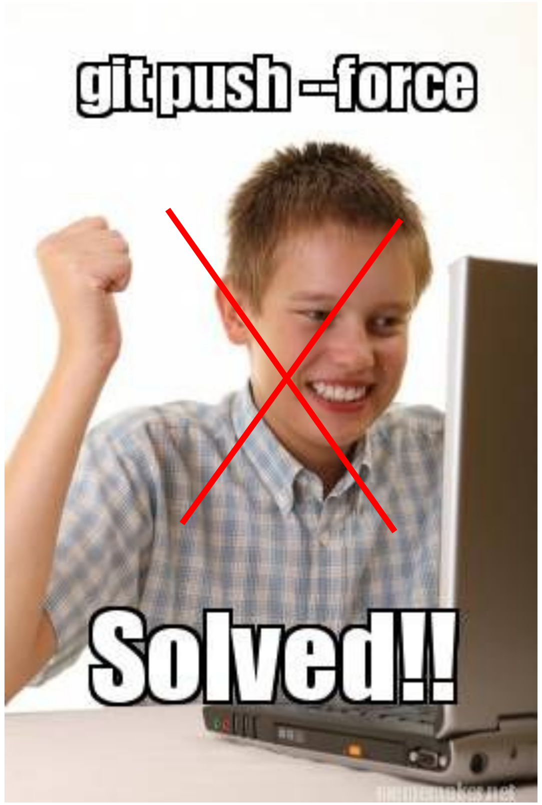
Resolving Merge Conflict

When Does Conflict Occur



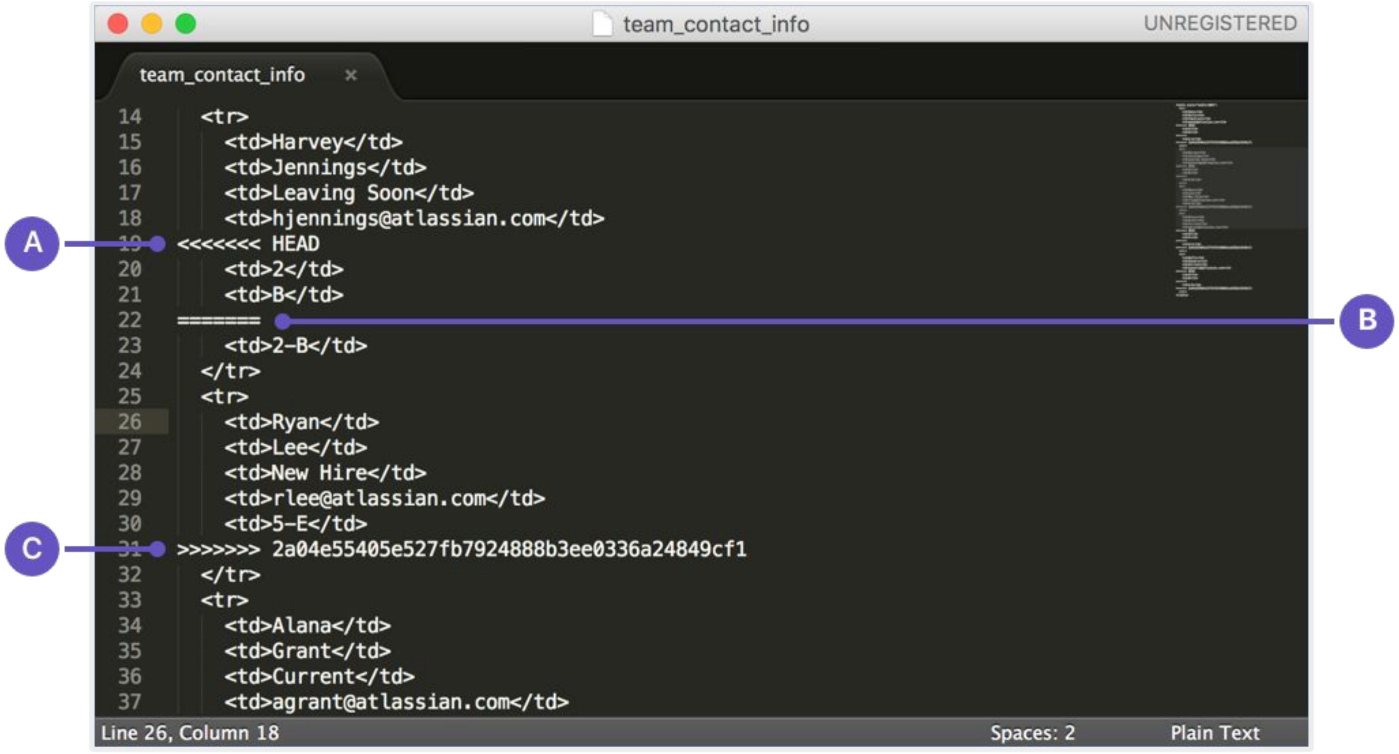
How to Resolve Merge Conflict

- ~~git push --force will solve all~~ (No!)
- Actually, we must resolve all conflicts by hand
- Thus, we need to minimize conflicts
    - Pull (sync) main (upper) branch frequently
    - Keep single branch's lifetime short
    - Do not make redundant changes
        - Line breaks, additional spaces, etc...



How to Resolve Merge Conflict

- Fortunately, git shows where conflicts occurred like



```
  <tr>
    <td>Harvey</td>
    <td>Jennings</td>
    <td>Leaving Soon</td>
    <td>hjennings@atlassian.com</td>
<<<<<<< HEAD
    <td>2</td>
    <td>B</td>
=======
    <td>2-B</td>
  </tr>
  <tr>
    <td>Ryan</td>
    <td>Lee</td>
    <td>New Hire</td>
    <td>rlee@atlassian.com</td>
    <td>5-E</td>
>>>>>>> 2a04e55405e527fb7924888b3ee0336a24849cf1
  </tr>
  <tr>
    <td>Alana</td>
    <td>Grant</td>
    <td>Current</td>
    <td>agrant@atlassian.com</td>
```

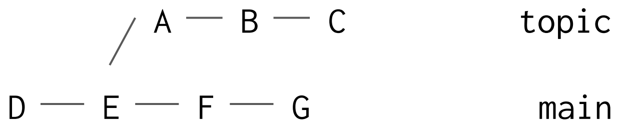
Some Other Commands

Other Commands (`git fetch`)

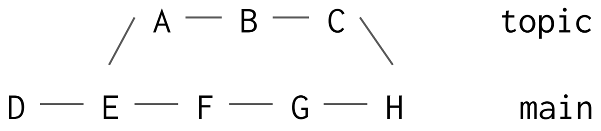
- Download objects and refs from another repository
- Assume your teammate pushed some changes in ‘branch1’ of your team repository
- However, you can see only main branch of your team repo (no branch1) in your local workspace
- If you type `$ git fetch origin branch1`, it will download branch1’s contents into your local workspace

Other Commands (`git merge`)

- You can merge 2 branches directly without creating PR

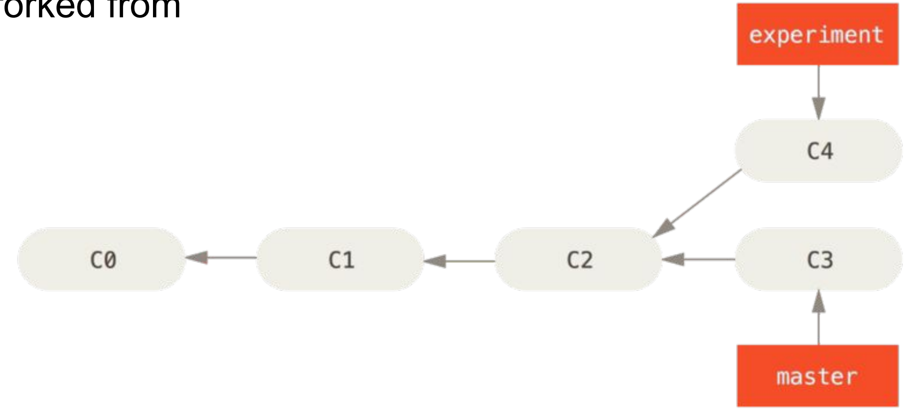


- **`$ git merge topic`** `# at main branch`



Some *Empirical git usage tips* I earnt from my experience

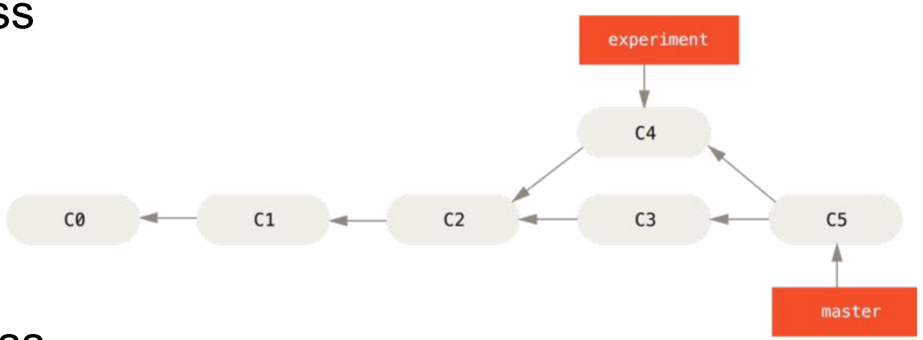
- Use git rebase, not git merge
  - merge is typically used when the head of the project differed from version of the branch you forked from



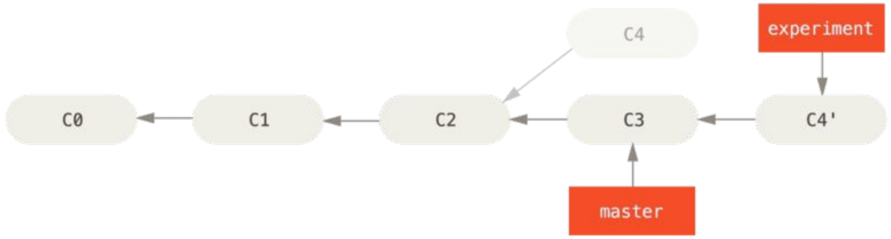
  - If different changes were made to the same file, you have to resolve conflicts to make a PR.
  - So, people first merge main branch to their branch, and resolve conflicts.
  - But this is very bad, since the changes that people reviewed already is again added as a ‘new change’ to your commit.
  - rebase prevents this.

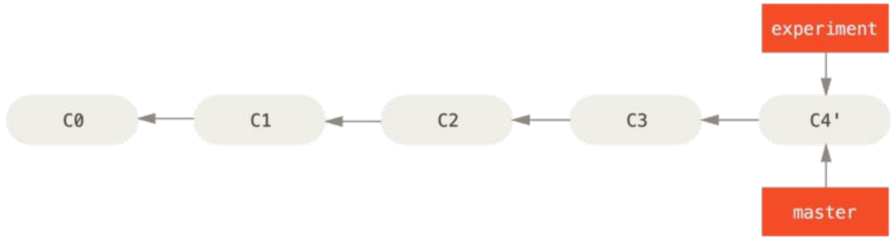
Some *Empirical git usage tips* I earnt from my experience (cont)

- merge process



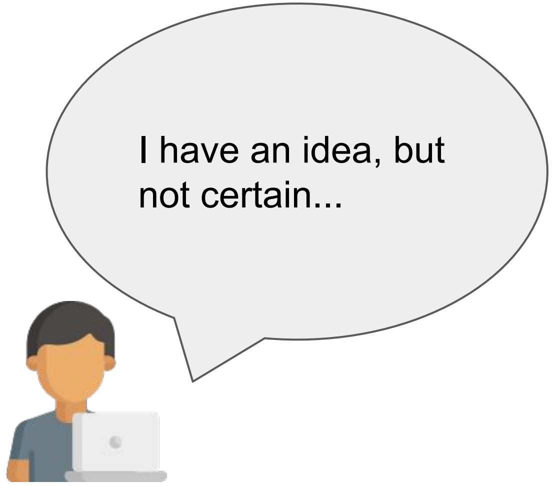
- rebase process





Other Commands (`git checkout`)

- Assume you want to test your idea (experimental implementation)
- After implementation, you found that it does not work
- However, since you modified files a lot, you cannot roll-back to the past version manually
- **`$ git checkout .`** `# or specific filename`
  will throw all unstaged (untracked) changes away!
- **`git checkout ${BRANCH_NAME}`**
  will switch branch
- **`git checkout ${COMMIT_ID}`**
  will switch working directory into specific commit



Other Commands (`git stash`)

- Assume you are working in `branch1`

```
$ git status
On branch branch1
Changes not staged for commit:
 (Use "git add <file>..."...)
     file1.py
     file2.js
     file3.py
```

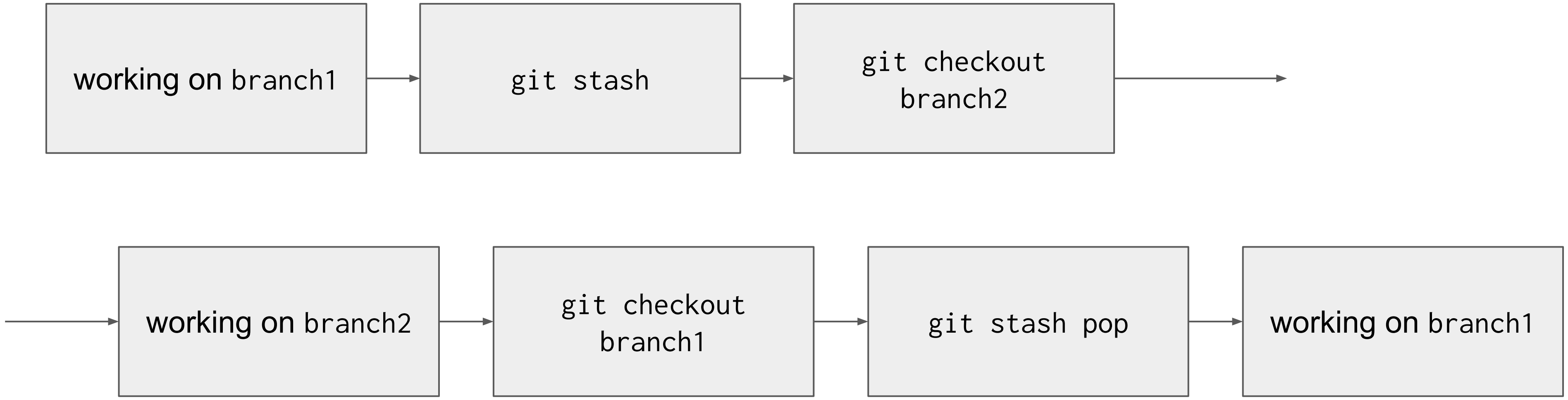
However, since you already modified lot in your branch, you cannot checkout to another branch before commit

Other Commands (`git stash`)

- You can save your local changes in stack using `git stash`
- **`$ git stash`**
- You can see stacked items
- **`$ git stash list`**
- You can reload stacked item
- **`$ git stash pop`** `#or $ git stash pop ${ITEM_INDEX}`

Other Commands (`git stash`)

- Workflow in this scenario:



Other Commands (`git reset`)

- You already added, or committed some files but you want to cancel it
- Cancel staged (added) file
- **`$ git reset HEAD ${filename}`** `# HEAD is reference to the current commit`
- Cancel recent commit
- **`$ git reset HEAD^`** `# HEAD^ is reference to the previous commit`
- Modify commit message
- **`$ git commit --amend -m “new commit message”`**

Other Commands (`git remote`)

- You forked repository from other repository, and you want to sync with the original repository
- **`$ git remote add upstream ${UPSTREAM_URL}`**
- **`$ git fetch upstream`**
- **`$ git pull upstream main`**

Other Commands (`git rm`)

- You want to remove redundant file in remote repository
- **`$ git rm --cached ${TARGET_FILENAME}`** `# remove from git only`
- **`$ git rm`** will remove local file too!
  (Use when you want to remove file from track & remove in local also)

Wrap Up

- Please keep these in mind and try to practice today’s materials on your own
- Now, please do not this…
- `$ git commit -m ‘Yeah~~~ finished!!!’`

I do not know why it does not work

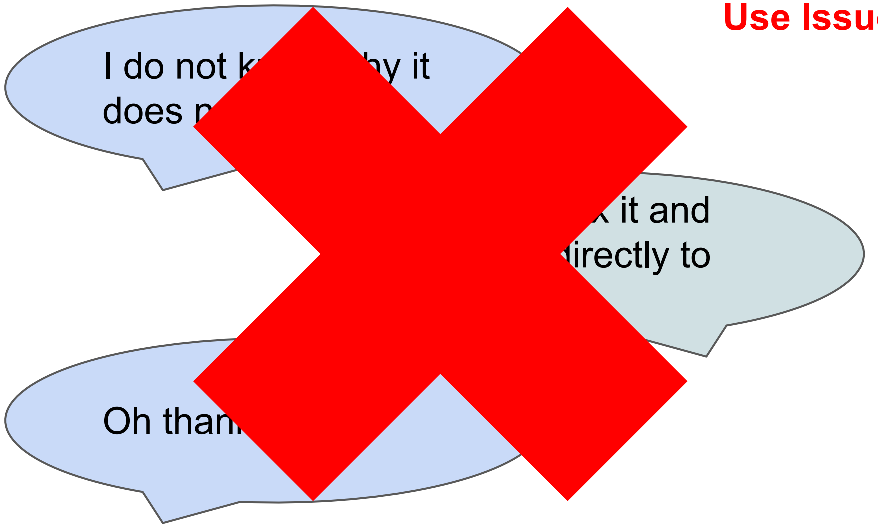
I will just fix it and commit directly to main

Oh thanks!

Wrap Up

- Please keep these in mind and try to practice today's materials on your own
- Now, please do not this…
- `$ git commit -m 'Yeah~~~ finished!!!'`

Use Issue and PR!

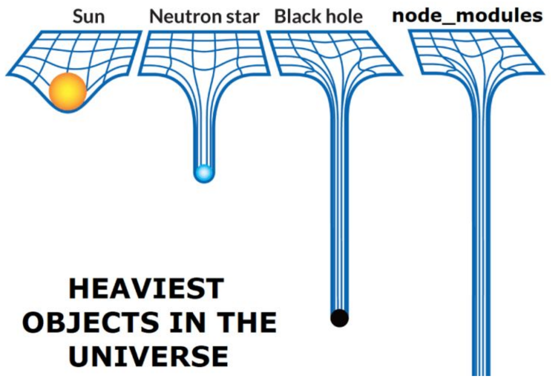


Some Tips for Github Management

1. Make branch naming convention with your teammates
    - e.g.) **`name-feature-issue`** `# e.g. haeyoon-login-#3`
2. Use shared linter and formatter
    - If each person uses different linter and formatter, lots of redundant conflicts will occur
    - You can set formatter config like `.style.yapf`, `.jsbeautifyrc`, `.clang-format` for various formatters (yapf, jsbeautify, clang-format for each)
    - You can set linter config like `.pylintrc, .eslintrc` for various linters (pylint, eslint for each)

Some Tips for Github Management

1. Use shared, well-defined `.gitignore` file
   - files listed in .gitignore file will not be tracked by git
   - Merge conflicts in redundant file / directory (like `node_modules!!`) is a disaster :(
2. Generate commits frequently, in small steps
3. Keep single branch (maybe named `main`, or `deploy`, or `v#.#`) always stable
   - Good example by some institutes is that they deploy their program in two version (stable, nightly)
   - Maaaaaybe TAs will check your current app's status before each sprint meeting



Next Session

- We will learn basic React
- Please finish installing `Node.js, npm, yarn, create-react-app`
- Please setup your IDE (Intellij or VSCode) with proper React/Javascript plugins