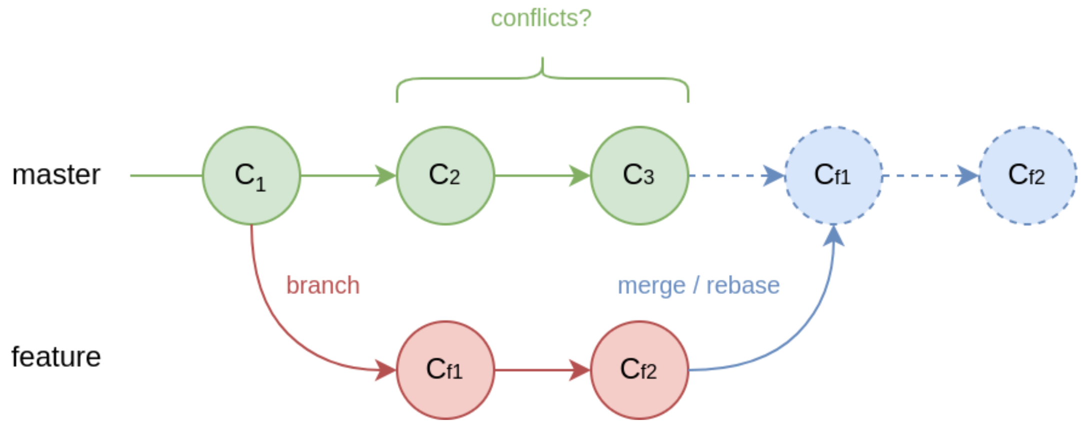
03. [50p] Adding & changing features

When you want to add a new feature to your project, you should first develop it in a branch [https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches]. A branch is a named copy of the deltas that comprise your codebase up to a certain point. By adding commits to this copy, you won't interfere with other people trying to do their own thing. Note that a branch can be created from any other branch, including *master*.



Eventually, you will want to merge your commits with the original branch. This can be done in two ways: `git merge` or `git rebase`. Here is a discussion [https://www.atlassian.com/git/tutorials/merging-vs-rebasing] on which is better. You should probably read it at some point. In this lab we will be focusing on `git rebase` since it is more interactive and provides many functionalities that you will need when trying to get your changes accepted by the maintainer / reviewer.

[30p] Task A - Add token flag

The feature that you'll want to add to your project is a command line argument parser that will accept an optional `-t, --token [TOKEN]`. We suggest that you use argparse [https://docs.python.org/3/howto/argparse.html]. In absence of this token, you will fall back to fetching it from the environment variable.

```
# first, create a new branch from HEAD
$ git branch feature

# next, switch to the feature branch
$ git checkout feature

# check that the branch you are on is actually feature and not master
$ git branch
  * feature
    master

# edit and test your script
# argparse should add a default '--help' option

# commit changes and push them to the remote feature branch
# first push means that the branch needs to be created (follow the command's hints)
$ git add ${BOT_SCRIPT}
$ git commit -s
$ git push
```

At this point, you have created a separate *feature* branch, added a (hopefully) working CLI argument parser, and pushed the newly created branch to your *remote*. When working with other people, now would be a good time to create a Pull Request [https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request] (PR). This is a request to the maintainer of the project to pull your *feature* branch, check that everything is working, and give feedback if changes need be made. If changes are indeed requested, all you have to do is address them in a new commit which you'll push into your *feature* branch. The PR will be updated automatically. Once the reviewer gives his ok, your changes will be applied to the *master* branch.

Since this is your repository and you have to deal with integrating the changes, let's use `git rebase` to do just that.

```
# switch back to the master branch
$ git checkout master

# apply the extra commits from feature onto master
$ git rebase feature
Successfully rebased and updated refs/heads/master.

# remember to push the newly integrated changes to remote
$ git push
```

"Wait. That's it?" Well… yeah. Luckily, you did not have any conflicts with *master*. If you did, `git rebase` would have told you exactly where those conflicts were located. Moreover, it would have modified your files to look something like this:

```
<<<<<<< HEAD
Changes made to master since branch.
=======
Changes made to feature since branch.
>>>>>>> feature
```

In order to resolve the conflicts, you would have to remove the lines with *"<<<"*, *"==="*, *">>>"* and rewrite the conflicting code so that it incorporates both your changes, and those already pushed to *master*. Finally, mark the conflicts as resolved by re-adding the files, and continue your rebase.

```
# re-add files with solved conflicts
$ git add ${CONFLICTING_FILES}

# continue the rebase process
$ git rebase --continue

# alternatively, you can just give up and go back to how things were (no harm done)
$ git rebase --abort
```

This part is now optional, but it would be nice to clean up and delete the *feature* branch both locally and remotely. All changes that *feature* held are now part of *master*, so what is it good for anymore?

```
# delete feature branch on remote (origin)
$ git push -d origin feature

# delete feature branch locally
$ git branch -d feature
```

[20p] Task B - Edit older commits

In the beginning we said that `git rebase` is interactive and fun. But we never had the chance to show it. Remember in the previous exercise when we added the COPYING file and the copyright notice to the *Python* script? Let's say that the reviewer changed his mind about this and now wants us to create two separate commits. One for the script and one for the copy of GPLv3. How would we go about solving this problem?

.venv mants@victim ii/II-bot master
0 0* zsh
↑ 2h 05m 31s 1.2 1.2 1.1 2021-12-06 18:27 victim

Click GIF to maximize.

Why, using `git rebase`, of course!

```
# take a look at the commits we have so far
#    #1: adding the bot
#    #2: adding the GPL license
#    #3: adding the argument parser
$ git log

# launch git rebase in interactive mode (-i)
# and tell it we want to revisit the last 2 commits relative to our head
$ git rebase -i HEAD~2
```

After running `git rebase -i`, it should have opened your default CLI file editor (same as with `git commit`). Notice that you have two lines that look something like this, followed by multiple lines describing commands.

```
pick 4864b9d Added GNU General Public License.
pick 37d816c Added cli argument parser for token.
```

Once we save this file, **git** will parse it's non-comment contents line by line and execute the commands on the given commits, in the order that they were specified. The **pick** command just selects a certain commit. By swapping lines, you will tell git to **pick** commits in a different order, thus reordering them on your current branch. Deleting a line will effectively delete the changes made by that commit in the current repository. What we're interested in, however, is the **edit** command. This command tells **git** to stop the rebasing process at that specific commit and let you make changes to it before proceeding.

```
# HEAD is now on commit 4864b9d which we marked for edit

# revert the changes made with this commit ==> files no longer added in staging area
$ git reset HEAD~1

# check the status of the files; see how COPYING and the Python script are now untracked
$ git status

# add the files one at a time; and commit them separately
$ git add COPYING
$ git commit -s
$ git add ${BOT_SCRIPT}
$ git commit -s

# from one commit, we now created two; continue the rebasing process
$ git rebase --continue

# check to see that two new commits were indeed created
$ git log

# push this changes to remote, thus rewriting history
$ git push --force
```

Once again, force pushing a different commit history onto the *master* branch is a bad idea if working with other people. But doing it onto your own branch is not only fine, but sometimes necessary in order to address the reviewer's requests.