GitHub Workshop 2020

Marios Fokaefs, Aquilas Tchanjou, Paraskevi Massara, Elena Comelli

marios.fokaefs@polymtl.ca, aquilas.tchanjou-njomou@polymtl.ca, p.massara@mail.utoronto.ca, elena.comelli@utoronto.ca

Day 2

1. Create and use branches (recommend changes within the team)

1. Imagine that you have a repository that is directly used by a considerable number of collaborators. Now, imagine that you learned about a new method and you are excited to implement in your repository, but you are unsure how… You do it anyway, but you notice that some bugs have been introduced and your code does not work any more. Tired after a long night, you decide to commit any changes, make a couple of issue to let others know what you are working on and remind yourself what needs to be done and you go for sleep. The next day you wake up in a flurry of angry mails from your colleagues that you broke the code! Oopsie!
2. To avoid this scenario, it is recommended to employ the concept of branches. A branch is a parallel development stream that remains disconnected and independent from the main branch (the "master"). So, while the latter remains available and well-tested for general use, you can do all the experiments you like in a separate branch. Branches can be used to develop new features, fix bugs or experiment safely with new ideas.
3. We can create a branch from the web interface:
    a. In repository page, locate the button labelled "Branch: master" just above the contents of the repository.
    b. If you click on it, you will see it's actually dropdown menu with a list of all branches (just master for now) and an empty field where you can search branches or create a new one. Enter the name "styling" in this field, so that we can create a branch where we will fix code styling problems as we saw before. Next, simply hit "Enter" and the branch will be created.
    c. In order to work on the new branch from RStudio, we will have to pull the latest version of the repository, which also includes the new branch. Not much should be different now, but this action would be relevant once we commit.
4. …or we can create a branch from RStudio:
    a. At the top right corner, you should be able to see tabs "Environment", "History", "Git". If you cannot see the last one, go to View->Show Git. This is a short of summary view of the Git commit dialog page. When we make changes, they will be listed in this view and we will be able to commit and push our changes.
    b. The important buttons for now are on the write of the menu. You can locate a button labelled "master". If you click it, it will list all the available branches of our repository.
    c. To create a new branch, click on the button next to it. You will be prompted to provide the branch name, in our case "styling". If you want the branch not to be local, but also

   appear in the repository, make sure you check the box “Sync branch with remote”. Hit Create.
   d. Now the list will contain both the “master” and the “styling” branches.
5. When a branch is created it copies the entire content of the repository. The branch button you located before, points you to which brunch you are currently viewing. It is also possible to create a branch from another branch.
6. Just above the content, you will notice the phrase “This branch is even with master”. When we make changes to the branch this message will inform us that the two branches are different.
7. Let us open our clustering script. As we saw before, generic variable names are not very useful and variable names should convey their purpose. In our script we are using the name “my_data” which does not say much. Let’s rename that to “iris_data” and to “iris_petal_data”, as two more informative alternatives. Make sure you change all appropriate references. You can test the code, just to be sure.
8. Now that we have made our change, we can commit and push to our “styling” branch. Go to Tools->Version Control->Commit.
9. In the commit dialog, you will see the modified file and the changes that we have made. Be careful! Our intention is to commit this change to the “styling” branch and not to the “master” branch. So from the top make sure “styling” is selected.
10. Make sure you stage the changed file and write a commit message. Hit Commit when ready. Then hit Push to push to the stream.
11. If we go back to the repository page in our browser, we will notice a few differences. The most obvious is that we now have 3 commits. However, if we switch back to the master branch, we will see that it has only 2 commits. Back to the styling branch, GitHub informs us that “This branch is 1 commit ahead of master.”
12. Right above this, you will see a yellowish ribbon that informs us that we have recently pushed branches. At the end of the ribbon we can see another green button labelled “Compare and pull request”. This will lead us to a future topic on Pull requests.

2. Create and use forks (recommend changes outside a team)

1. What if we want to recommend changes to a project, which is not developed by our team? For example, let’s suppose that we use an R package and we have found a bug in it. You feel that the developers of the package may be very busy to fix this bug, but you need to move on quickly with using the fixed version. At the same time, you have studied the bug and you know how to fix it! Good for you! But how are you going to implement the fix if you don’t have access to the repository? Enter Forks!
2. Forks, pretty much like branches, are exact copies of a repository, which allow you to perform changes, without affecting the baseline project. The purpose behind forking a repository can vary including submitting a fix (as we mentioned on our example) or an improvement, or creating a new project, which we want to inherit part or all of the baseline project and extend it towards a completely new and independent direction.
3. As a simple example let’s try to fork the workshop repository!

4. Navigate to the https://github.com/massaraevi/GitHub_workshop2020 repository. In the top-right corner of the page, click **Fork**.
5. A copy of the repository will be created on your GitHub account, and you can edit it as if you created it yourself. If you just want to contribute to the project, you can create a pull request to the original project for your contribution to be integrated in the original project.

3. Proposing changes with pull requests

1. Whether you made your changes through a branch or through a fork, most probably you'd like these changes to be pushed back to the master branch or the baseline respectively. However, especially in the case of the fork, the developers of the base project need to trust you and they need to verify that what you are trying to push is valid, for example that it does not cause any additional bugs, it goes along with the conventions of the project (e.g. with respect to styling) or most importantly it does not reduce the quality of the project or introduce any security vulnerabilities. Therefore, in these cases, instead of pushing straight to the stream, we instead submit a "pull request".
2. As the name suggests, a pull request is an invitation to the developers to accept your changes to their project. A pull request may consist of multiple commits. Once a pull request is opened, the developers of the base project will review your changes. Should they accept them, they will merge your request in their master branch. When thinking about branches, remember that the *base branch* is **where** changes should be applied, the *head branch* contains **what** you would like to be applied.
3. Let's go back to our repository, where we defined the styling branch and let's open a pull request. At the yellowish ribbon, it's time to click on the green button to "Compare and pull request".
4. The "Open a pull request" page looks very similar to the issue page. First we are asked to select from what branch we want to merge towards what branch. In our case, it is from styling to master. GitHub informs us that the two branches are compatible and they can be merged automatically. In a worse case scenario, there may be conflicts between changes that occurred in the master branch, while we were developing our styling branch. Conflict resolution is a complicated topic, which we may look in the future together.
5. Next, we can add a comment to our pull request to guide the review process. For example, "this was a change to fix styling conventions." If you notice the pull request inherits the title of the commit, we are trying to merge.
6. Additional properties can be defined, as in the case of issues, including assignees, labels, projects and milestones.
7. In addition to issues, though, we can also define reviewers. These are developers of the target stream (other than the assignee), who will need to review our code before it gets accepted for merge.
8. Once everything is done let's hit the green button to create our pull request. The pull request page has a couple more tabs compared to the issue page, one of which is checks. This tab essentially coordinates the review efforts and it works in tandem with the Actions tab we discussed earlier (and unfortunately is out of scope for this workshop).
9. Given that the branch has no conflicts and can be merged automatically, GitHub gives us another green button, but this time it corresponds to four options:

    a. “Merge pull request” - “Create a merge request” (default option, if we click on the button): GitHub automatically merges the pull request to the main branch and records the merge event. In this case, the history (individual commits) and the authorship are maintained.
    b. “Squash and merge”: The several commits included in the pull requests are first merged as a single commit in the cloned branch/fork and that commit is merged to the master branch. In this case, both the history and the authorship of the contributions are lost.
    c. “Rebase and merge”: The individual commits are merged one by one in the master branch and no merge event is recorded.
    d. Using, Git, it is possible to select only certain commits to merge to the base branch. This is called cherry-pick. Unfortunately, GitHub doesn’t have any feature for this, and it should be done using the Git command line or a Git GUI.

10. In our case we can hit the default option. Once the merge is completed, we can see that the labelled as merged. Since no other commit exists in the branch, GitHub informs us that we can now safely delete the branch. If we want we can continue the development and merge again in the future.
11. Should we regret the merge, we can always revert it by hitting the button next to the last event in the pull request history. This process will initiate a new pull request.

4. Merge and conflict resolution

1. After a pull request has been opened, you (as the repository administrator) can review and discuss the set of proposed changes, then merge the changes to the specified branch.
    a. On GitHub, you can see the changes made by the pull request creator, suggest modifications, add comments, mark a file as viewed and submit your review
2. By default, any pull request can be merged at any time, unless the head branch is in conflict with the base branch. Let’s merge pull request 1 to the master branch.
    a. Navigate to the main page of the repository and select the “Pull Requests Tab”. By default, it will display the opened pull request waiting to be merged.
    b. Select “pull request 1” to open it. You will se the details about the pull request such as description and the commits included in that pull request.
    c. To merge the pull request, you can click the green drop down button “Merge pull request and select the merge method you want.
        i. the default **Merge pull request** option on a pull request on GitHub, all commits from the feature branch are added to the base branch in a merge commit
        ii. When you select the **Squash and merge** option on a pull request on GitHub, the pull request's commits are squashed into a single commit. Instead of seeing all of a contributor's individual commits from a topic branch, the commits are combined into one commit and merged into the branch.
        iii. When you select the **Rebase and merge** option on a pull request on GitHub, all commits from the topic branch (or head branch) are added onto the base branch individually without a merge commit.

        iv. Using, Git, it is possible to select only certain commits to merge to the base branch. This is called cherry-pick. Unfortunately, GitHub doesn't have any feature for this, and it should be done using the Git command line or a Git GUI.

    d. Let's select the default Merge pull request option, then select confirm merge to merge and automatically close the pull request.

3. Merges (from pull request on GitHub or from simple pull in RStudio) can create conflicts. They occur when GitHub can't make a decision about what should be on a particular line of a project file modified by different collaborators (or when you work on the same file from different pc) and needs a human (you) to decide. Let's create and resolve a merge conflict.
    a. go on GitHub, switch to features branch and change the line 2 of readme.md file to "Changed line number 2 from GitHub", then commit with commit name "readme GitHub update"
    b. Now let's go to RStudio, make sure we are on features branch and replace the line 2 of readme file with "Changed line number 2 from RStudio", then save the file. Now let's start the commit pipeline by first pulling from remote repository
    c. We get and error! There is a merge conflict. So, we are not allowed to pull, it failed. GitHub is protecting us because if we did successfully pull, our work on RStudio would be overwritten by whatever we had written on GitHub. So, GitHub is going to make us decide. GitHub says, either commit this work first, or "stash it" (It means saving a copy of the README in another folder somewhere outside of this GitHub repository).
    d. Let's follow the advice and commit.
    e. Now let's pull again, we still get an error. Actually, we're just moving along this same problem that we know that we've created: We have both added new information to the same line. This kind of conflicts often occur when different collaborators are working on the same line in the same file. Let's close that window and inspect. Notice that in the git tab, readme.md is tagged orange `U`; this means that there is an unresolved conflict, and it is not staged with a check anymore because modifications have occurred to the file since it has been staged.
    f. Let's look at the README file. We got a preview in the diff pane that there is some new text going on in our README file showing the local and remote changes.
    g. So, to resolve this merge conflict, we have to choose, and delete everything except the line we want. So, we will delete the `<<<<<<HEAD`, `=====`, `>>>>long commit identifier`
    h. Now let's commit and push. It works.

4. Pulling, syncing and committing often will help reduce the number of conflicts.