This document was created approximately at 01.08.2019.

# Table of contents

Contents

[Table of contents 1](#_Toc15363467)

[Introduction 4](#_Toc15363468)

[Basic operations 6](#_Toc15363469)

[Add File 6](#_Toc15363470)

[Commit File 9](#_Toc15363471)

[How can you check that file you are about to commit was added (to index) ? 12](#_Toc15363472)

[Push (no conflicts) 14](#_Toc15363473)

[Pull (no conflicts) 19](#_Toc15363474)

[Merge 23](#_Toc15363475)

[*What to do if you realize that you’ve messed up with your merge, in other words you did something wrong in your conflict resolution.* 31](#_Toc15363476)

[*What if I’ve messed file with the files that are not present in the list?* 33](#_Toc15363477)

[*What if I want to delete my last commit?* 37](#_Toc15363478)

[*If you have conflict when you’re* ***pulling*** *from git, just resolve the conflict as described.* 38](#_Toc15363479)

[*If you have conflict when you’re* ***pushing*** *to git* 39](#_Toc15363480)

[*All conflicts fixed but you are still merging.* 43](#_Toc15363481)

[Theory 45](#_Toc15363482)

[Distributed Version Control 45](#_Toc15363483)

[Optimistic locking 47](#_Toc15363484)

[Stage/Indexing 49](#_Toc15363485)

[Branches 51](#_Toc15363486)

[Commit History 52](#_Toc15363487)

[Why not master only? 56](#_Toc15363488)

[Feature branch 62](#_Toc15363489)

[New Branch 62](#_Toc15363490)

[Merging Branches 66](#_Toc15363491)

[Rename branch 81](#_Toc15363492)

[Delete branch 85](#_Toc15363493)

[New Tag 88](#_Toc15363494)

[Delete Tag 91](#_Toc15363495)

[Stash 92](#_Toc15363496)

[What if you’re in the middle of the work and you want to make “git pull”? 92](#_Toc15363497)

[What if you have complicated merge and you prefer to make it in more controlled way? 102](#_Toc15363498)

[Create New Project 113](#_Toc15363499)

[From existing project (git clone) 114](#_Toc15363500)

[From Scratch (git init) 117](#_Toc15363501)

[From Scratch using git commands (git init) 127](#_Toc15363502)

[Appendix A 131](#_Toc15363503)

[Appendix B 132](#_Toc15363504)

[Troubleshooting 132](#_Toc15363505)

[If you close merge resolution windows and when you want to commit your merge changes, you don’t see any file that change 132](#_Toc15363506)

[If you successfully resolve all merge conflicts, but you receive error message “You have not concluded your merge (MERGE\_HEAD exists)” 132](#_Toc15363507)

[How do I know what branch is currently checkout? 133](#_Toc15363508)

[If Pull failed, you should merge your changes first and then 136](#_Toc15363509)

[**Git Delete Last Commit** 137](#_Toc15363510)

[Delete all local branches which were delete on the server 137](#_Toc15363511)

[List all tags 138](#_Toc15363512)

[Remove remote tag/branch 138](#_Toc15363513)

[Create remote tag/branch from local tag/branch 138](#_Toc15363514)

[Rename tag 139](#_Toc15363515)

[Removed hidden files from Git (such as .idea) 140](#_Toc15363516)

[Advanced Troubleshooting 141](#_Toc15363517)

[If Pull failed, you should merge you changes first and then 141](#_Toc15363518)

[If you want to move remote branch (for example origin/master), move your local branch to the desired point (Reset current branch to Here…) and then type. 141](#_Toc15363519)

[How to remove 'id\_rsa' or 'id\_dsa' files that was pushed by mistake 141](#_Toc15363520)

[How to cleanup garbage in remote git repo 143](#_Toc15363521)

[Update Git 144](#_Toc15363522)

[SSL certificate problem: self signed certificate 144](#_Toc15363523)

[Appendix C 146](#_Toc15363524)

[Appendix D 154](#_Toc15363525)

# Introduction

This is how-to style document on how to use Git.

The dedicated audience is new Git user.

In general, it has Windows-oriented, but it should be pretty straight-forward to apply it on another platform.

If you have some SVN knowledge, please see [Appendix A](#_Appendix_A).

For troubleshooting see [Appendix B](#_Appendix_B). It uses command line command to resolve some troubles that you can encounter with. Advanced Troubleshooting recipes can damage your repository; they should be used with extra care.

There some choices and assumptions that was made deliberately and only part of them are discussed in the document. For example, “git rebase” was deliberately chosen not to include. There is no assumption, however, on your remote repository, it can be any Git provider such as GitHub, GitLab, etc.

PyCharm is used as git client through this document. See [Appendix D](#_Appendix_D) for details. All examples are deliberately provided on PyCharm.

Document start with [basic opertaions](#_Basic_operations) sush as [“git add”](#_Add_File), [“git commit”](#_Commit_File), [“git push”](#_Push_(no_conflicts)) and [“git pull’](#_Pull_(no_conflicts)).

Merge concept is spread out through the whole document. Basic description of the merge process is [described in merge](#_Merge) section. It described many day-to-day use-cases and pitfalls. It is essential to understand how you can resolve conflicts. [Merging Branches](#_Merging_Branches) section assumes that you’re comfortable with [Git theory](#_Appendix_A); it describe how you should merge branches. Advanced, not recommended technique is described in [stash subsection](#_What_if_you). There are some use-cases in [Appendix B](#_Appendix_B) Troubleshooting. Advanced Troubleshooting recipes can damage your [commit history](#_Commit_History); they should be used with extra care. [Appendix C](#_Appendix_C_1) contain simplified real world example that show that conflict resolution can result with wrong result (specifically, I demonstrate code duplication, but it can lead also to not working code).

Next section is [Theory](#_Theory). I deliberately differ theoretical explanation as far as possible in order that you will have good basic understanding on how you’re going to work with Git. Further section assume that you read it. It is particular important to understand [Branches](#_Branches) section and [commit history](#_Commit_History).

[Why not master only?](#_Why_not_master) subsection has discussion of the best practices of working with Git.

[Feature branch](#_Feature_branch) section describes one of the best practice that I recommend to do. There is link on [alternative branch modules](https://nvie.com/posts/a-successful-git-branching-model/), but they are out of scope for this tutorial. This section describes how to create [new branch](#_New_Branch), how to [merge branches](#_Merging_Branches) (see Merge concept paragraph above), also how to [rename branches](#_Rename_branch). Creating [new tag](#_New_Tag) and [deleting tag](#_Delete_Tag) are described in appropriate section. See also in Troubleshooting 3 subsection on working with tags ([List all tags](#_List_all_tags), [remove remote tag/branch](#_Remove_remote_tag/branch), [create remote tag/branch…](#_Create_remote_tag/branch) and [rename tag](#_Rename_tag). Working with tag is surprising bad in PyCharm beyond simple operation. For example, there is no ability to rename the tag.

Next section is about [Stash](#_Appendix_A). You should use it occasionally. First subsection ([“What if you’re in the middle of the work...”](#_What_if_you’re)) describes typical usage. Advanced, not recommended technique is described in [second stash subsection](#_What_if_you) (see also Merge concept paragraph above),.

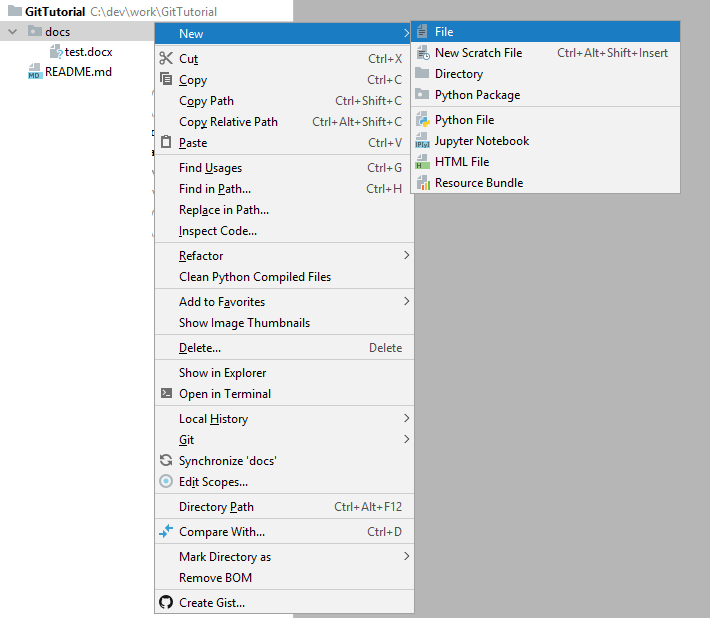
Next section describes how to [create new project](#_Create_New_Project). You are not going to create new project every day, but you definitely should know how to do it. Usually, you will [create new project from existing project (git clone)](#_From_existing_project). Another way is to [create new project from scratch (git init)](#_From_Scratch_(git). Or you even [create new project using git commands (git init)](#_From_Scratch_using). The latest section can be useful also if you want to see what PyCharm are doing under the hood.

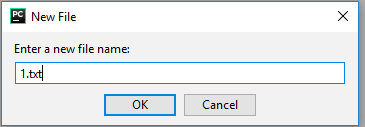
Final note, despite my attempt to structure the text that you can read linearly without jumping back and force, I did add some notes that suppose the knowledge of the subsequent section. My advice is to skip such notes on the first read and then come back and read them again.

# Basic operations

## Add File

* If you are adding new file through PyCharm, it will be automatically added to Git (you will be able to commit it without any addition actions).





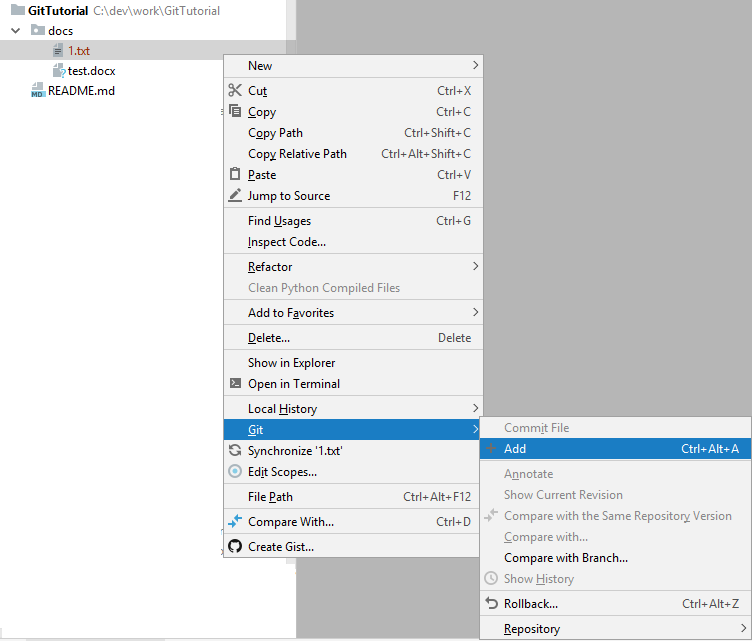
In such a case you just do nothing.

Note: If you copy/move file with PyCharm it will be also automatically added to Git and will be available for commit alter.

This are typical cases of working with files. PyCharm will “git add” it for you.

* Occasionally, you will want to add files outside PyCharm.
* For instance, you will copy this file from another location in Windows.
* Or you’re creating new file in cmd\Terminal.
* Another use-case, when PyCharm is not added this file. See in commit section, how do you know about this.

Right-click on file that you want to add (you can select directory/project instead) -> Git -> Add.



Note: ”Commit file” is not available (it is greyed out).

Note: It is safe to add file multiple times; it will do nothing (after first time).

## Commit File

In order to commit file, it should be added first.

Note: In PyCharm it will be usually added (to [index](#_Stage/Indexing)) automatically.

Note: PyCharm mark not “git added” file by red color.

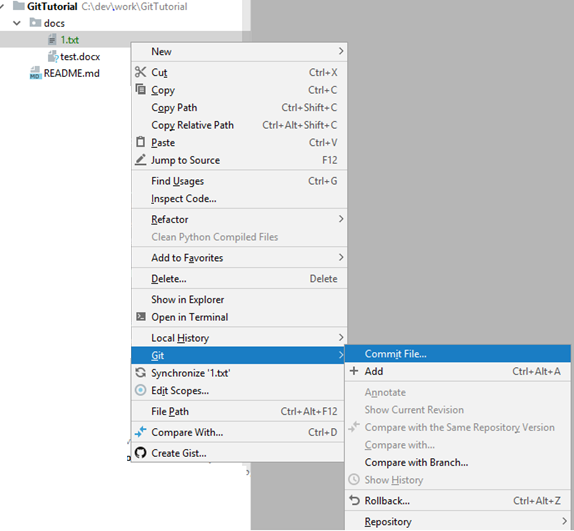


Note: PyCharm mark modified file by blue color.

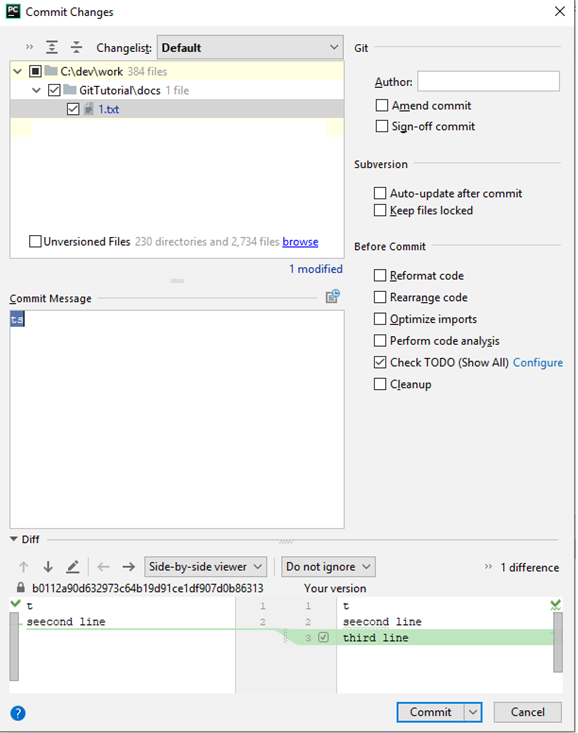


In order to commit file, you can

right-click on file that you want to commit (you can select directory/project instead) -> Git -> Commit.

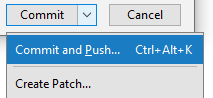


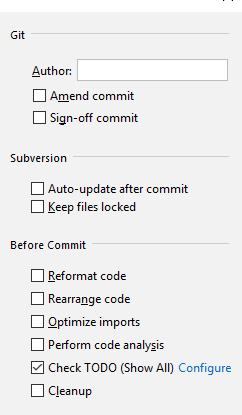
You will see the following window:



Let’s go through the options:

* At the top subsection you see what files are about to commit.
* Commit message it the message that will be associated with this commit. It is intended to be the short description of what this commit changes.
* At the button section you can see the change that you ‘ve done for every file that you chose in the top subsection. See the [merge](#_Merge) section for more detail description.
* Cancel button will cancel you commit.
* Commit button will do the commit operation.
* There is also  near commit. This is not atomic operation, it will be translated to “git commit” and “git push” (last one can fail if there is conflict). If you select it, you will see the following:



* Commit & Push make 2 operation one after the other – it commits the changes and immediately [push](#_Push_(no_conflicts)) it.
* I will not go over the options in the right subsection:  
    
  

I want only to make some notes on “amend commit” option.   
  
**Important: “Amend commit” rewrites the** [**commit history**](#_Commit_History)**.**

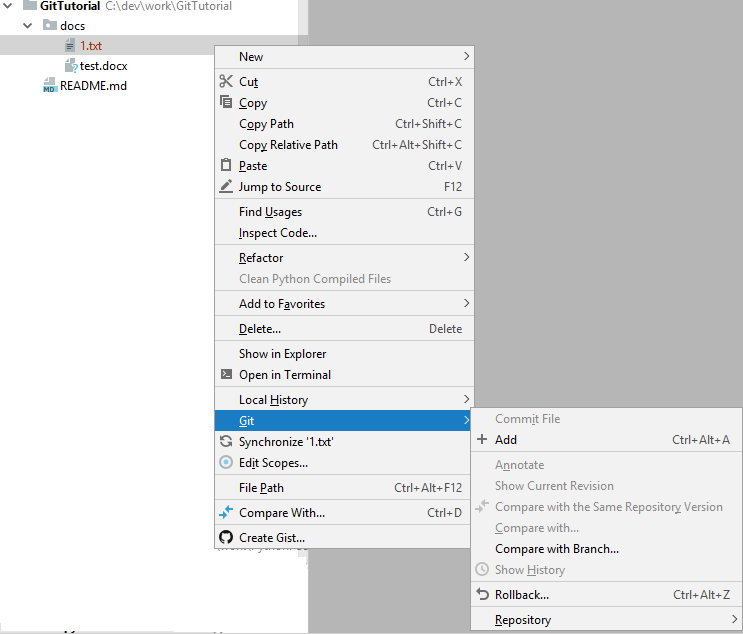
Note: Git documentation highly didn’t recommend to use it if commit was already pushed (more precisely if it is available on remote branch, that somebody else work on it; if it is remote [branch for the feature](#_Feature_branch) you’re developing and you’re the only one that works on this branch – this is fine, you can use “amend commit”). It recommends only to use it if you make commit without push.

Note: (advanced, you can skip it). In particular, it can be useful when you have [merge conflict](#_Merge), you resolve them and did merge commit. And then you want to make extra manual change to some file, but this change is part of the conflict resolution. Than you can amend commend and only then push it.

### How can you check that file you are about to commit was added (to [index](#_Stage/Indexing)) ?

One way is simply try to commit:

right-click on file that you want to commit (you can select directory/project instead) -> Git -> if commit is not available (is greyed out) than you should add the file (directory/project) first.



## Push (no conflicts)

After you commit your changes, you want them to be available to everybody. This is what “git push” does. Before, “git push”, all your commits are “visible” only for you.

Note: **If push failed due the conflict,** see [“merge” section](#_If_you_have) for details.

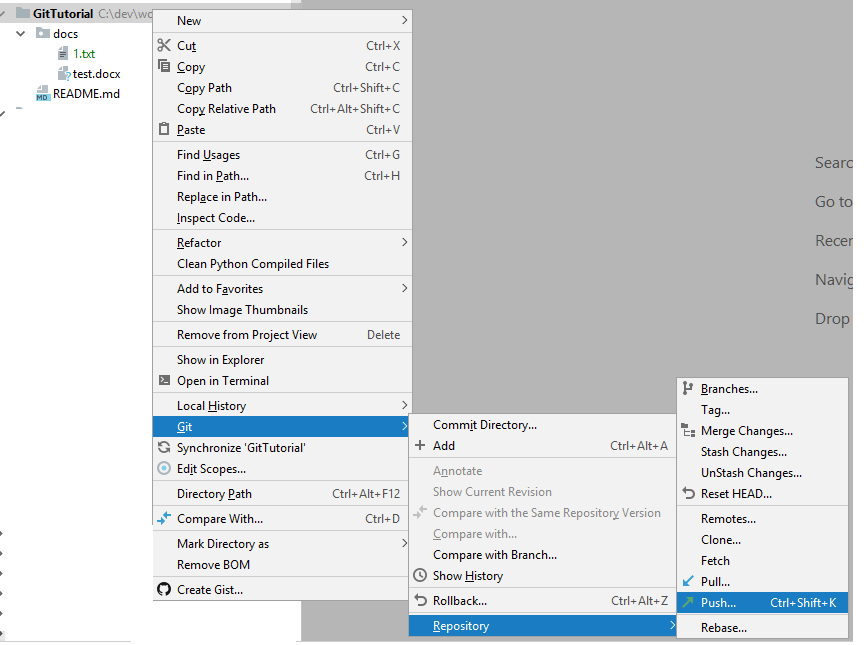
Note: Typically, you should do [“git pull’](#_Pull_(no_conflicts)) first.

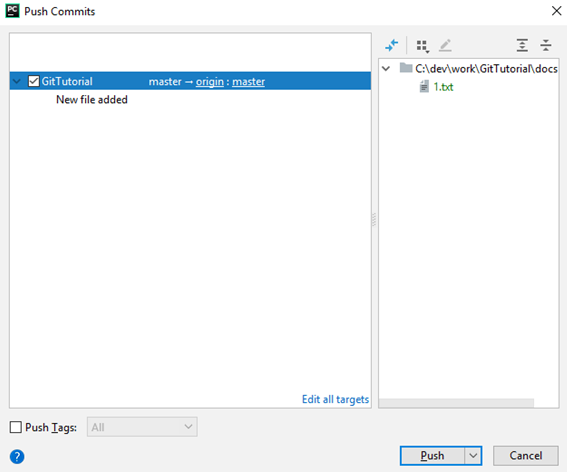
Note: Anyway, in this section I’m assuming that there are no conflicts in “git push”. Conflict resolution is walked through in “[Merge” section](#_Merge). See subsection “[If you have conflict when you’re pushing to git”](#_If_you_have).

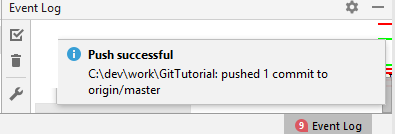
Right-click project -> Git -> Repository ->Push

In the popup window check the project you want to Push. Click on “Push” button.

You should see in the right left corner in the floating window “Push successful”.





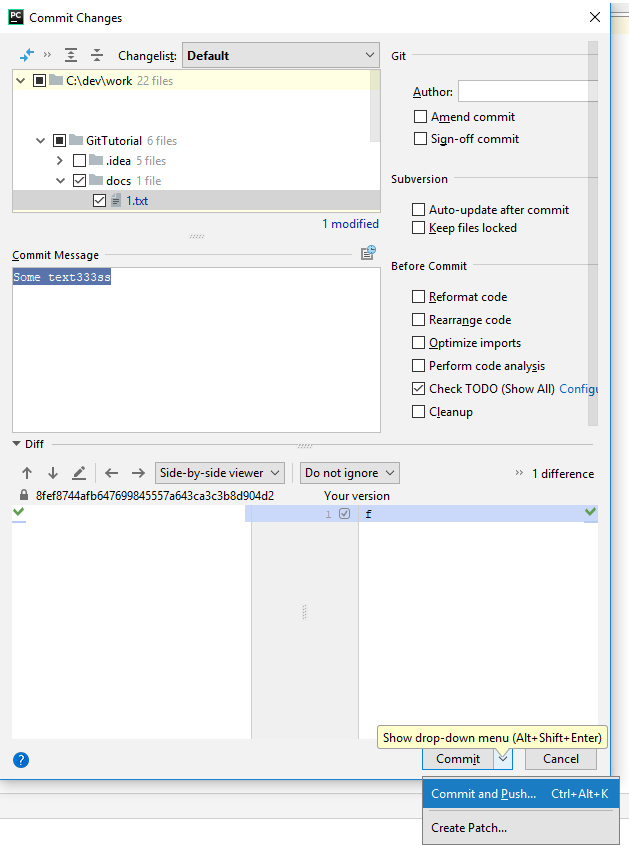


In the push commits window, you see all the Git project. In every Git project you see message of every commit that you’re about to push. On the right you can see from what files each commits consists of.

Note: **If push failed due the conflict,** see [“merge” section](#_If_you_have) for details.

Note: You can also commit&push together. This is not atomic operation, it will be translated to “git commit” and “git push” (last one can fail if there is conflict).

In the “commit changes” window near the commit button, you can click on  and you will see sub-menu in which you can chose commi&push.



## Pull (no conflicts)

This operation takes changes that somebody else did and integrate them to your project.

Note: In this section I’m assuming that there are no conflicts in “git pull”. Conflict resolution is walked through in [“Merge” section](#_Merge), see subsection [“if you have conflict when you’re pulling from git, just resolve conflict as described”.](#_If_you_have_1) There is also some advanced not best practice technique [here](#_What_if_you).

Note: It is advised to do “git pull” once in a while.

Important: It is better that **you will not have any uncommitted files before** you do “git pull”.

Note: Pulling from Git in PyCharm by default also pulls tags&branches from remote repository (You will have tags&branches in your local repository if they are defined in remote repository).

Note: If you make some changes, but didn’t finish you work and you want to pull the changes, you can stash (see [section on Stash](#_What_if_you’re)) your changes first, then make pull (resolve conflicts with your previous commits, if any), unStash (it will apply you changes on updated code-bases; you may require to resolve conflicts (again)). It is better to have all your changes committed.

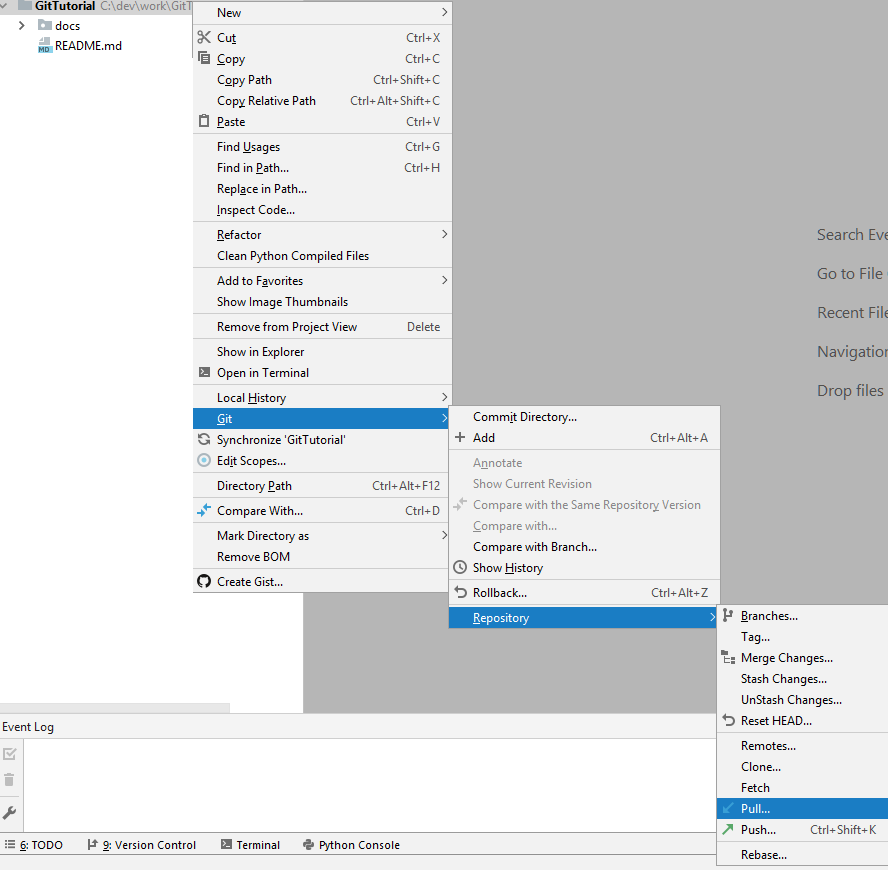
Note: “Git Pull” is combination of “git fetch” (this fetches changes and storing them aside (this operation can’t have failed because of conflicts)) and applying changing on workspace (this can lead to conflicts).

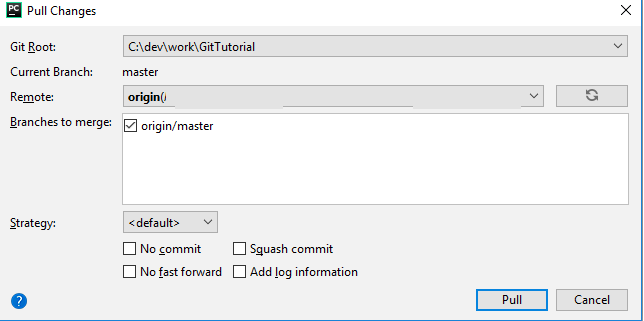
Righ-click on the project->Git>Reposiotry>Pull…

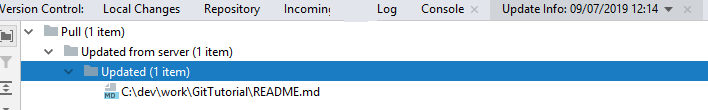
In the popup window click on “Pull” button.

Note: In Branches to merge you should choose the corresponded remote branch for your local one. Typically PyCharm chose this for you. For example, of you’re working on master, chose origin/master.

You should see “Updated Info: *some\_datetime”* tab in the button of the PyCharm.







# Merge

This operation is crucial for day-to-day work. There are many use-cases when you need to do merge. They will be discussed below.

Merge is the operation that you need to do in order to *resolve conflicts*. What conflicts? Conflict between what do you have and what another people have.

How conflicts created? There are multiple ways, below only some of them are listed:

* Suppose, you change some file, say 1.txt. You didn’t commit it. Now, you’re **pulling** the other’s changes, and somebody else also change 1.txt. That is, **you and somebody else change the same file**. There is conflict.
* Suppose, you change some file, say 1.txt. You commit the change. Now, you’re **pulling** the other’s changes, and somebody else also change 1.txt. That is, **you and somebody else change the same file**. There is conflict.
* Suppose, you change some file, say 1.txt. You commit the change. Now, you’re **pushing** your changes (maybe by commit&push option), but somebody else also change 1.txt. That is, **you and somebody else change the same file**. There is conflict.

Note: it is recommended to always pull before push, but there is still time window between pull and push, that you can receive new changes.

Note: If you change some file 1.txt and somebody else changes 2.txt, and you pull your changes, there will be no conflict. **Conflicts are detected on per file bases.**

Note: Because conflict detection is on per file, this also means that if you’re doing refactoring moving some part of the file to another file and there is conflict with second file it is hard to **detect**& resolve such a conflict. This use-case will be discussed in [Appendix C](#_Appendix_C_1).

You can need to make merge in these cases:

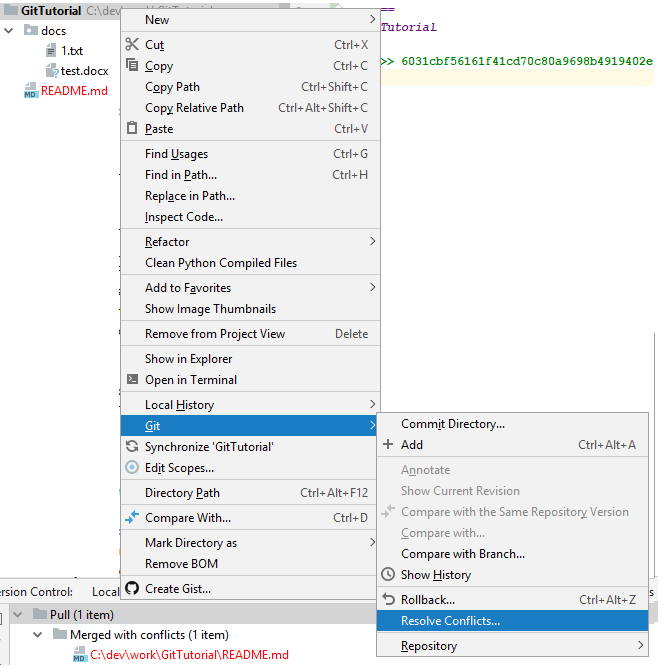
* On [“git pull’](#_Pull_(no_conflicts)) – this is typical case.
* On [“git push”](#_Push_(no_conflicts)). This can happen, but should be rare.
* When you specifically merge different branches (this will be discussed in [separate section](#_Merging_Branches)).
* More [advanced](#_Stash_1) cases.

Before we will go through the merge flow I want to emphasis when *conflicts* can occur:

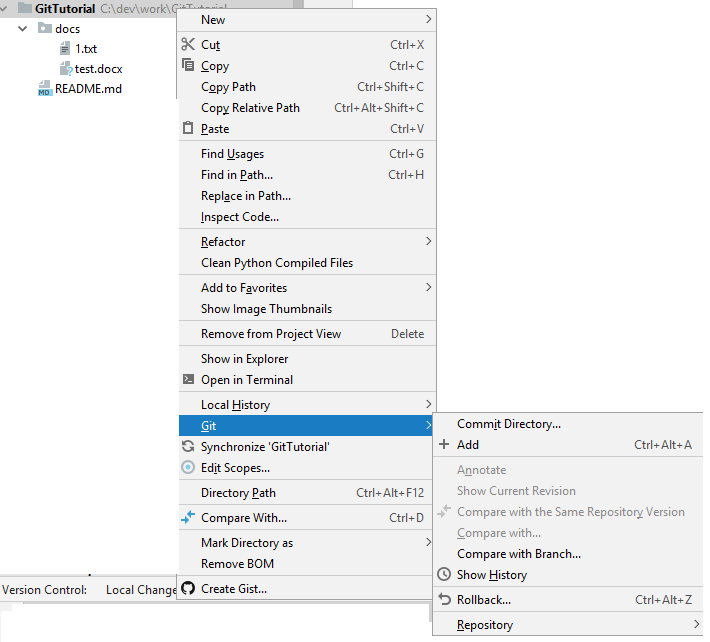
* If you add file to git, no conflict can occur on this operation.
* If you commit file, then on commit operation, no conflict can occur.

Typically, PyCharm will open popup window “Files Merged with Conflicts” automatically.

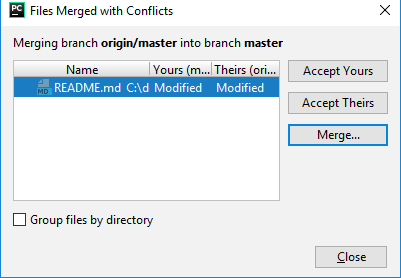
If not, right-click on the project->Git->Resolve conflicts…



Note: “Resolve Conflicts...” option will not be available (even as greyed out) if you don’t have any conflict to resolve.



What you should see is the following window:



Let’s examine what we actually see.

1. You see list of files that have conflicts. In this example, there is only one file README.md.
2. You see that you modified the file.
3. You see that “they” modified the file
4. For each file in the list you can:

* click on "Accept Yours” button – this will leave all you did, and will discard others changes.

Note: There is some [caveat](#_All_conflicts_fixed) here. It will be discussed later.

Note: Don’t do it blindly. Otherwise, you will discard some other changes.

* click on "Accept Theirs” button – this will discard all your changes, and you will receive all “theirs” changes.

Note: Don’t do it blindly. You should know why it is ok to discard your changes (maybe your changes are meaningless, maybe you know what is incoming change, etc).

* click on “Merge...” button.

This preferred option.

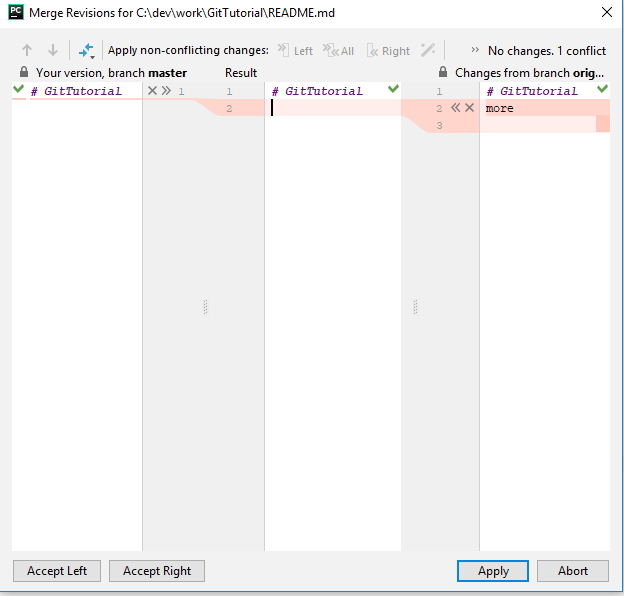
After clicking on “Merge...” button new windows will be opened:

Note:

* + If the file that has the conflict is **binary** (for example, MS Word document, MS Excell, zip, jpeg, etc.) You will not the diff (all panels will be empty).
  + In such a case, one option will be checkout previous version of the file and compare it **manually** with current one. Make another commit (or, may be, [amend the last commit](#_Commit_File)). Push the changes.
  + May be better option will be to:

1. Take their” changes, commit them.
2. Put the resulting file aside.
3. [Reset the head](#_What_if_I’ve) to the parent commit with you changes.
4. **Manually** compare them, make final merge version of the file.
5. [Reset head back](#_What_if_I’ve) to the last commit.
6. Make another commit (or, may be, [amend the last commit](#_Commit_File)).
7. Push the changes.

Note: You can look on [commit history](#_Commit_History) of the file, but you will not be able to open a **binary** file without checking out the whole repo on appropriate commit.



Let’s explain what are you looking on:

* The Left panel contains your version of the file (it has your changes).
* The Right panel contains “theirs” version of the file (it has others changes).
* The middle panel contains result of the merge.
* Near every difference you can see “X”, “>>”, “<<” signs.
* “X” means – discard this change. This change will not get in to the result.
* “>>” means – take the change from the left panel. This change will get in to the result.
* “<<” means – take the change from the right panel. This change will get in to the result.

Note:

1. You can write free text on Result. In general, it is not advised.
2. **You should go over every conflict and either take it or discard it**. In other words, you should click on every “X/”>>”, “<<” sign (On every conflict you should chose on these options).

* click on "Accept Yours” button – this will leave all you did, and will discard others changes.

Note: There is some [caveat](#_All_conflicts_fixed) here. It will be discussed later.

Note: Don’t do it blindly. Otherwise, you will discard some other changes.

* click on "Accept Theirs” button – this will discard all your changes, and you will receive all “theirs” changes.

Note: Don’t do it blindly. You should know why it is ok to discard your changes (maybe your changes are meaningless, maybe you know what is incoming change, etc).

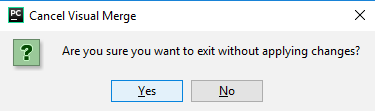
* “Abort” button will abort merging, See details below.
* “Apply” button will save you changes and close the merge window.

Note: As far as git concern, the **merge process will end only after you will commit your changes** (git commit)**.**

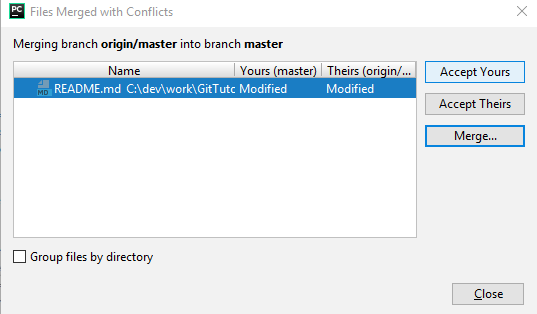
Note: If for all files in the merge you click on "Accept Yours” button, you will have no files to commit. So, PyCharm will not let you commit your merge. This is [caveat](#_All_conflicts_fixed) from above. I will discuss it more [below](#_All_conflicts_fixed).

## *What to do if you realize that you’ve messed up with your merge, in other words you did something wrong in your conflict resolution.*

1. Click on “Abort” button
2. This will open the following popup



Click yes to abort the merge, click on no - if you want to undo you clicking on “Abort” button.



1. Now, you can click on “merge” button again and restart the merging of the changes of file

Note: You can also type in cmd/Terminal

git merge --abort

to abort currently running merge and then restart the process that lead to merge.

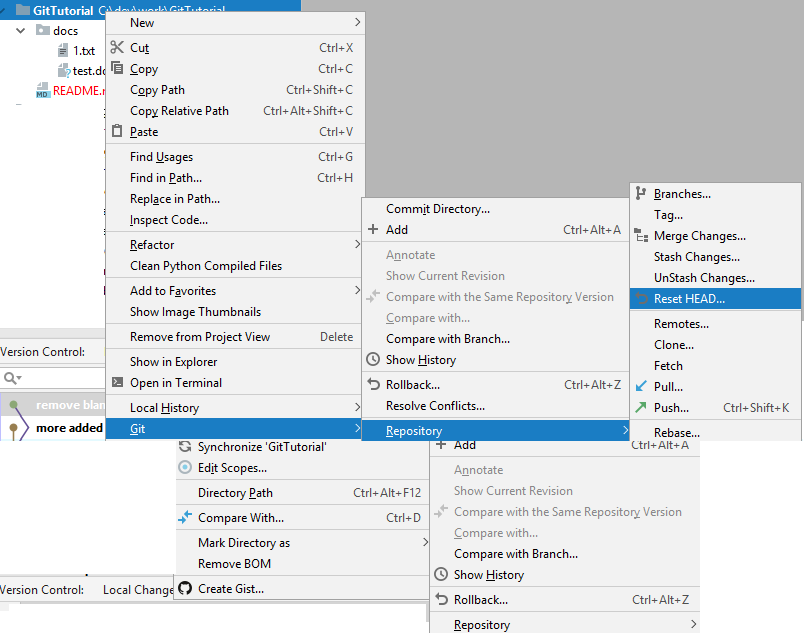
## *What if I’ve messed file with the files that are not present in the list?*

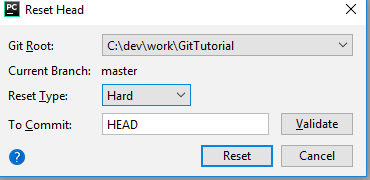
In such case you should “RESET head”.

Note: When you do it, you should be aware on [what branch you’re currently working on](#_How_do_I). For example, if you’re working on master, the head of master will be moved.

Right-click project -> Git -> Repository ->Reset head..

“Reset Head” window should be opened.

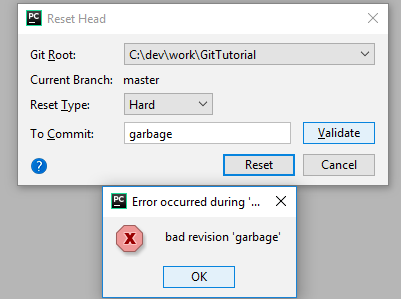




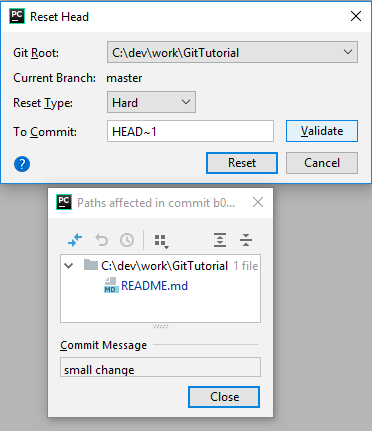
* Reset Type “Hard” means that any change that was made locally will be lost. This is typical case.
* Reset Type “Soft“ means that no changes to files will be done. If you want to “undo” merge operation, this is not very helpful.
* Reset Type “Mixed” means that any change (with exception of files that wasn’t “git added”) that was made locally will be lost. If you have some files that wasn’t “git added” they will remain. This is PyCharm default. Typically, you don’t have files that are not “git added”, in this case the behavior will be identical to “Hard”.
* To commit: to what commit to reset the head. Typically, you will write HEAD here, this means, typically, the latest commit.

You can also write HEAD~1. HEAD~1 is a shorthand for the commit before HEAD. Alternatively, you can refer to the SHA-1 of the hash you want to reset to.

* You can optionally validate that your type valid value to “To Commit” box. Click on “validate” button.  
    
  - If you put invalid commit identifier, you will receive “Error occurred during..”



* If you put valid commit identifier, you will receive “Paths affected in commit …” windows with commit message and files affected by the commit.



Note: It will generate “git reset --hard HEAD~1” command under the hood (or “git reset --hard HEAD” if you will type HEAD~1 in “To commit” row).

Note: See also [“Git Delete Last Commit”](#_Git_Delete_Last) in [Appendix B](#_Appendix_B) [Troubleshooting](#_Troubleshooting).

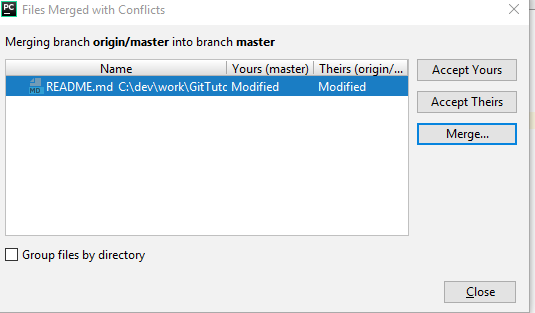
## *What if I want to delete my last commit?*

This should be very rare. But if you want not do it, see [“Git Delete Last Commit”](#_Git_Delete_Last) in [Appendix B](#_Appendix_B) [Troubleshooting](#_Troubleshooting).

## *If you have conflict when you’re* ***pulling*** *from git, just resolve the conflict as described.*

Note: I’m assuming that you don’t have any **uncommitted files before** you do [“git pull’](#_Pull_(no_conflicts)).

You should see the following window after you pull the changes.



[Resolve conflicts](#_If_you_have_1). Don’t forget to do “git commit” at the end.

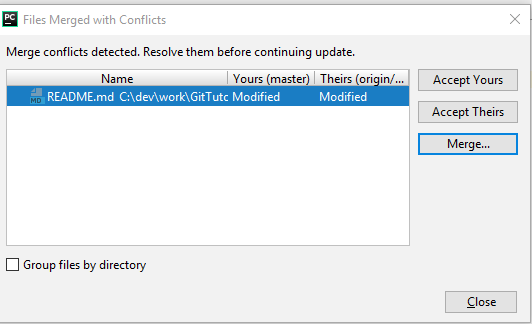
Note: See also [*Troubleshooting*](#_If_Pull_failed,_1) section.

## *If you have conflict when you’re* ***pushing*** *to git*

Note: I’m assuming that you don’t have any **uncommitted files before** you do “git push”.

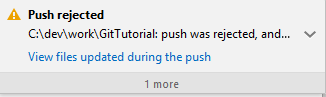
Note: Typically, you should do [“git pull’](#_Pull_(no_conflicts)) first.

You should see the following window:



**Click on Close button.** On this stage, we’re refusing to do merge.

You should see the following pup window in the right corner after you pull the changes.

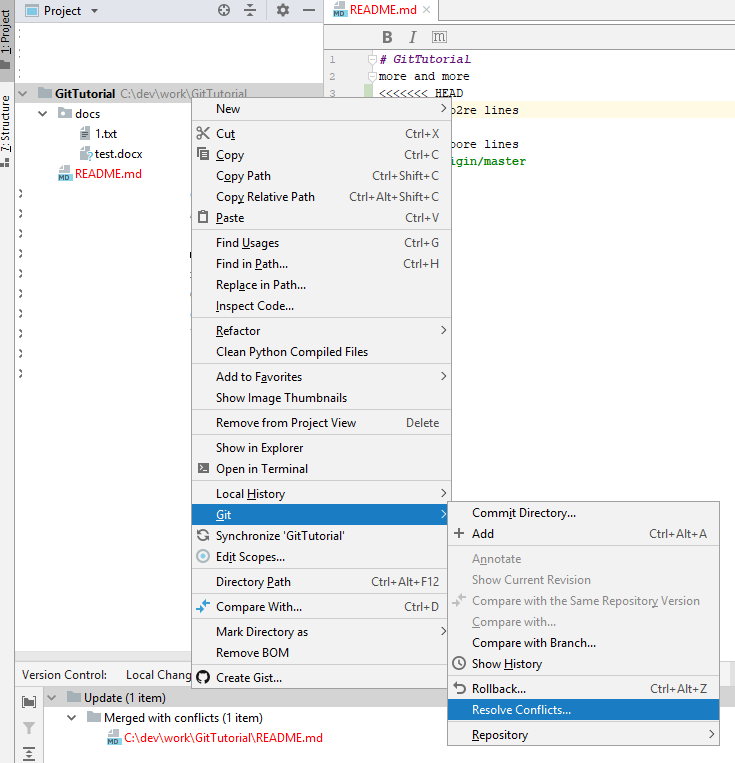


Now, PyCharm pulled the changes to your [workspace](#_Stage/Indexing).

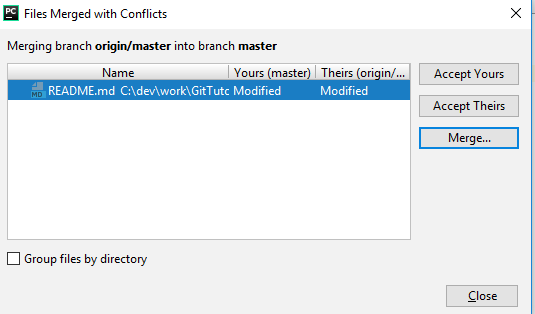
You should manually open the window and start merging process.

**Right-click project -> Git -> Resolve Conflicts…** This will create new commit with the merge changes.

**Push your changes again.**



You should see the following window:



Resolve conflicts. Don’t forget to do “git commit” at the end.

## *All conflicts fixed but you are still merging.*

What do you do if merge wasn’t concluding? How this can even happen?

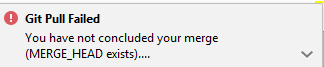
In theory, you can open cmd/terminal and type

git status

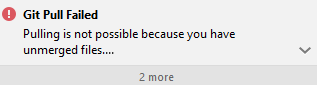
If you see something like



In practice, you will encounter with this situation, when some action that you want to do with git will fail. For example, when you make [“git pull’](#_Pull_(no_conflicts)) you may see the following message in the right corner of the window:



Note: If you didn’t resolve some conflict, you will see the following message instead:



What should you do?

1. First of all, in the future, when you’re solving merge conflicts it is better to introduce at least one change in some file.

* If you want to accept only your changes and ignore other’s changes (please verify, that this is indeed what you want to do), find some file and make some innocent change (such as adding new line or comment or removing new line).

1. Find some file in your project and make some innocent change (such as adding new line or comment or removing new line). Make commit. This will resolve your merge conflict and commit your changes.
2. Alternatively, you can open cmd/terminal and type

git commit  
  
Close opened editor, ignore it.

You should have additional commit “Merge branch ‘master’ of”.

or to abort merge

git merge --abort

* If you abort your merge, you can redo it implementing recommendation in 1. above.
* Alternatively, you can reset the HEAD. See subsection “[What if I’ve messed file with the files that are not present in the list?”](#_What_if_I’ve) above.

Note: If for whatever reason you want to postpone full conflict resolution you can use this [advanced](#_What_if_you) technique.

# Theory

## Distributed Version Control

It does not necessarily rely on a central server to store all the versions of a project file.

In **Distributed** Version Control Systems, every contributor has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.

Everybody maintains a local repository on its own, which is actually the copy or clone of the central repository on their hard drive. They can commit and update their local repository without any interference.

They can update their local repositories with new data from the central server by an operation called “**pull**” and affect changes to the main repository by an operation called “**push**” from their local repository.

The act of cloning an entire repository into your workstation to get a local repository gives you the following advantages:

* All operations (except push & pull) are very fast because the tool only needs to access the hard drive, not a remote server. Hence, you do not always need an internet connection.
* Committing new changes can be done locally without manipulating the data on the main repository. Once you have a group of changes ready, you can push them all at once.
* Since every contributor has a full copy of the project repository, they can share changes with one another if they want to get some feedback before affecting changes in the main repository.
* If the central server gets crashed at any point of time, the lost data can be easily recovered from any one of the contributor’s local repositories.

In practice, however, there is usually **main** repository. This main repository acts as remote repository and everybody has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.

This main repository is “single source of truth”. This is the “correct” version of the data.

Note:

1. If main repository is corrupted/failed it is possible to recreated it from any other local repository (local repository contain all information that main repository has). Of course, you will lose some data.
2. You can share your data directly, not through main repository.

For example, you have Git project at your work and you’re working on some feature. The feature is not complete yet, and you want to continue to work on it at another computer at home. You can setup new git local repository at your home and define it’s remote repository your git repository at your work.

## Optimistic locking

Suppose there is some file README.md in the remote repository. Suppose that 2 people want to change it.

It is clear, that there is problem here. If Git will do nothing, then second push will overwrite the first one and all changes that was done by first committer will be lost.

Let’s illustrate this with example.

* Commiter1 reads/pull README.md.
* Commiter2 reads/pull README.md.
* Commiter1 modifies&commit&push README.md.
* Commiter2 modifies&commit&push README.md.

Commiter2 has now over-written the changes that Commiter1 made. They are completely gone, as if they never happened.

There are 2 basic approaches to address this problem:

* To prevent the problem on the first place. That is only 1 committer can change the file. All other’s attempts will be rejected (on their push). This is **pessimistic locking**.
* To *detect* that there is a problem (at push) and have some plan to deal with it. This is **optimistic locking**.

Pessimistic Locking is when you lock the file by putting an exclusive lock on it until you have finished to work with it. If anybody else will try to modify this file, it will fail (or he/she should wait, that can take unbound amount of time). It has much better integrity than optimistic locking, but can be very slow.

Optimistic Locking is strategy where you read a file, *detect*s if somebody else is also work on this file (by taking a note of a version number; other methods to do this involve dates, timestamps or checksums/hashes), if there is no problem - you modify file (together with new version number or another attribute that is used for *detection*; these 2 updates should be atomic) and if you *detect* that somebody else modifies your file, you abort your operation, make [merge](#_Merge) (this integrates you changes with others changes; so others changes will not be lost) and then you restart your operation.

**Git uses optimistic locking**. Git assumes that the conflict should be rare, so while the cost of each single merge is high, the overall cost of conflict resolution is low. In practice, you should make active steps to reduce the probability of conflicts, they will be discussed in the chapters [Why not master only?](#_Why_not_master) and [Feature Branch.](#_Feature_branch)

In practice, working with optimistic lock means, that you can always locally modify any file. You can add/delete/update file and even commit your changes to the local repository. But **you should be ready to handle that your “git push” operation will fail** (even if you will pull before the push and resolve every conflict, subsequent push can fail **because of the conflicts**).

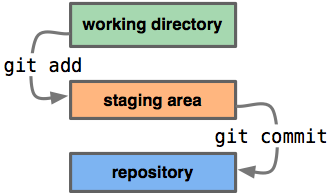
## Stage/Indexing

Some commonly used terminology:

1. the **workspace** is the directory tree of (source) files that you see and edit.

Note: I use workspace as alias for “working directory” throughout the document, stickily speaking they are not the same. PyCharm (and Git) has some background activity that sync them. Technically, “working directory” is a “git view” on your workspace (where you make actual changes). “Git view” here means “what git knows that you’ve changed.

1. The **index** is a single, large, binary file in <baseOfRepo>/.git/index, which lists all files in the current branch, their sha1 checksums, time stamps and the file name -- it is not another directory with a copy of files in it.
2. The **local repository** is a hidden directory (.git) including an objects directory containing all versions of every file in the repo (local branches and copies of remote branches) as a compressed "blob" file.



So, **"Staging"** is the process that one performs when selecting which modified files (or portion of files) will be part of the next commit.

The **"Index"** is the file into which Git stores the files that have been staged. It's also called the **"Staging area"**.

Adding a file to the index is merely saying "Git, this version of this file in my working directory should be part of the next commit I'm about to create".

## Branches

Branches in Git are very lightweight. A **branch in Git is only a reference** to a single commit (the tip of the branch). With its parental commits, the full branch structure can be constructed. It takes only few seconds to create and delete branches. [Merging branches](#_Merging_Branches) can take some time.

There are number of special references:

**HEAD**: the current commit your repo is on. Most of the time HEAD points to the latest commit in your current branch, but that doesn't have to be the case. HEAD really just means "what is my repo currently pointing at".

**master**: the name of the default branch that git creates for you [when first creating a repo](#_From_Scratch_using). In most cases, "master" means "the main branch". Your local repo has its own master branch, that almost always follows the master of a remote repo.

**origin**: the default name that git gives to your main remote repo. You have local repo, and you most likely push out to some remote repo that you and all your coworkers push to. That remote repo is almost always called origin, but it doesn't have to be.

In the event that the commit HEAD refers to is not the tip of any branch, this is called a *"detached head".*

HEAD is an official notion in git. HEAD always has a well-defined meaning. master and origin are common names usually used in git, but they don't have to be.

I want to reiterate. You *can* [move the HEAD to any commit](#_What_if_I’ve) and even you can [create tag](#_New_Tag) or [start new branch](#_New_Branch) from here, but usually you will not rewrite the [commit history](#_Commit_History).

Again, when HEAD didn’t point to the tip of any branch it is called *detached head*. Any commit to *detached head* will fail.

## Commit History

It’s natural to think of a commit as a diff. It turns out that the “real” model is not this, it’s actually that each commit is a snapshot of the whole repo state at the time. But actually, it isn’t, the underlying implementation does make use of deltas in packfiles and some other tricks like copy-on-write forking. The documentation tends to implicitly think of them as “full copies of the entire file tree”. But often it’s important to picture them as diffs, too.

Note: Next section is optional and can be skipped entirely.

*How are commits actually stored?*

*The way the actual implementation of a commit works is that each file being stored is hashed and stored in a compressed format, indexed by the hash. A directory (“tree”) will be a list of hashes, one for each file/directory inside it, alongside the filenames and other metadata. This list will be hashed and used everywhere else to refer to the directory.*

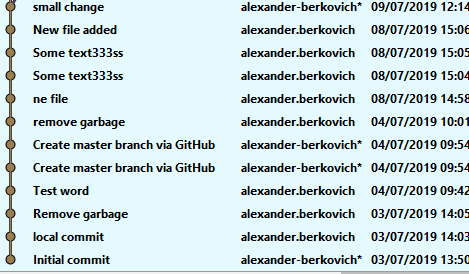
*A commit will reference the “tree” object for the root directory via its hash.*

*Now, if you make a commit changing some files, most of the files will be unchanged. So will most of the directories. So the commits can share the objects for the unchanged files/directories, reducing their size. This is basically a copy-on-write model. Furthermore, there’s a second optimization called a “packfile”, wherein instead of storing a file git will store a delta (a diff) and a reference to the file the diff must be applied to.*

*For more details see*

[*https://manishearth.github.io/blog/2017/03/05/understanding-git-filter-branch/*](https://manishearth.github.io/blog/2017/03/05/understanding-git-filter-branch/) *Appendix: How are commits actually stored?*

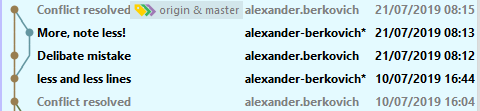
This is example of the commit history view:



This is very simple history. You should look on it from button to top (time axis is from button to top). Each point represents separate commit. You have commit message in the second column. In the last column, you have date, when the commit was done. In the middle column you have user name that did the commit. Asterisk near the user name means that commit was done on remote repo (this is not typical case, typically it is done only to create master branch on remote repo that is origin/master).

Let’s examine more complex commit history:

Note: I’ll refer to commits by their message just for simplicity. Messages are not unique, of course. Each commit has its own id such as a27586d5932d60f4b1b23e05c426fce8cb2b4fad. Note also, it is common to see only first couple of symbols such as a27586d.



On July 10 commit with message “less and less line” was made. It has single parent commit with message “Conflict resolved” (greyed out).

Than we have “split”. Chronologically next commit was “Deliberate mistake” that was done on local repo (no asterisk near user name). Than (chronologically) commit “More, note less!” was done on remote repo (note, asterisk).

Note, we see to edges: from “less and less lines” to “Deliberate mistake” and from “less and less lines” to “More, note less!”. The second edge is twisted in order to show chronological order of operation; it doesn’t have any other meaning (it is good as straight line).

Commit with message “Conflict resolved” is *merge* commit. **It has 2 parents**: “More, note less!” and “Deliberate mistake”. What I did, I did “git pull” operation and receive message that I have merge conflict, I resolved it and then make “git commit” and “git push”.

Note: Merge conflicts appeared to be greyed out on commit history. Sometimes, you can see also auto-generated messages theire.

Note (advanced): It is sometimes useful to [amend this *merge* commit](#_Commit_File). This is advanced technique that can be safely applied if this *merge* commitwasn’t pushed.

Note: Reference to “origin” and to “master” point out to latest commit.

Note: C**ommits content are immutable**. **You can’t modify commit’s content you have done**. You can add more commits only. In practice, however, you can rewrite the history (both in your local repository and in remote repository) that *effectively* will remove old commits and will add new ones. I want to reiterate technically, commit *content* is not changed, new commits can be created. The reference to the parent commit can be changed. But **it may *look like* as if you change the commit** (however, id of commit will be different).

* While [changing commits on your local repository that wasn’t pushed](#_What_if_I) is pretty safe, you should better avoid it.
* Rewriting remote repository history is possible, but should be done with care and with coordination with all team.

See [Why not master only?](#_Why_not_master) section for the step-by-step example of changing history (also it shows how references are changes).

See [Feature branch](#_Feature_branch) section for the details on how to create/merge branches and also how to create tags.

# Why not master only?

So, why it is bad idea to work with only master branch.

Note: I’m assuming you are familiar with [commit history view](#_Commit_History).

Let’s say, that I start to work on commit 5d0c2c0 on the master.

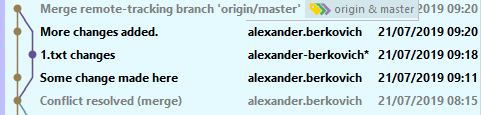
1. I committed some change on README.MD file.



Note: that master reference is ahead of origin/master reference. This is because we didn’t push our change, the remote repository is not aware of our change.

1. Now, we will simulate change that someone did in its local repository and then push it to main repository. For simplicity, I will make it directly on main repository.   
   I will modify some unrelated file, commit&push the change.
2. Now, I’m modifying on README.MD file again (I’m not aware on change in the remote repository). Now I’m commit&push the changes.

The commit history looks like this



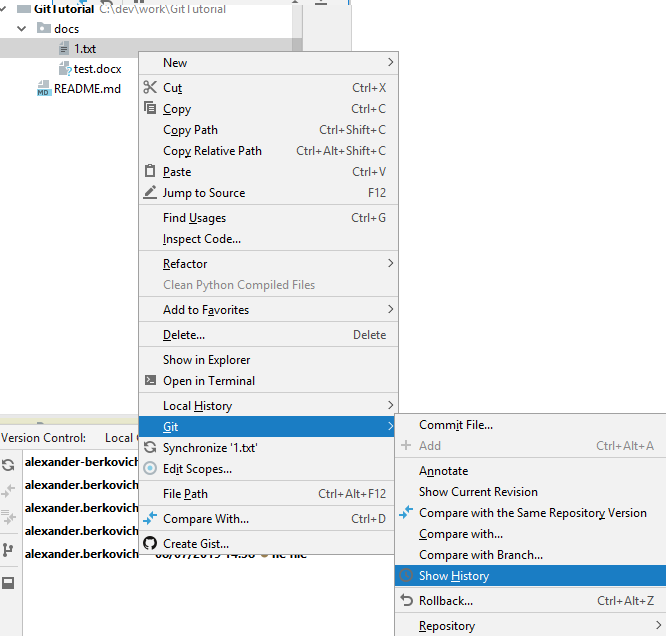
* My first commit was “Some change made here”. Its parent is “Conflict resolved (merge).
* My second commit was “More changes added”. Its parent is “Some change made here”
* But **what is “Merge remote-tracking branch ‘origin/master” commit (greyed-out)?** I wasn’t asked to do any merge (because there were no conflict), but actually Git created *merge commit* for me. We have 3 issues:

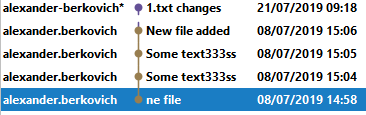
1. Git created *merge commit* automatically.
2. Our [workspace](#_Stage/Indexing) contains something we’re not aware for.
3. What if we do have merge conflict?

So, let’s examine all issues one-by-one.

1. Between my commits there was also commit with message “1.txt changes” that was done by somebody else. So, when I push my second commit, git recognize that from the local repository point of view the last commit in the remote is “conflict resolve (merge)” conflict (it is often refers as *base* commit), but in reality the HEAD of remote repository points to “1.txt changes” commit.  
     
   There is **mismatch**. So, what’s Git do, it **pull’s the changes (**“1.txt changes” commit**), merge them** (there is no conflicts, so it’s done silently), **makes additional merge commit and push 3 commits** (2 originals one and merge commits).
2. I want to re-iterate, you did commit&push, but Git on behave of you did [**“git pull’**](#_Pull_(no_conflicts))**.**In order to prove it will examine the history of 1.txt file.

Right-click on 1.txt file -> Git -> Show history.





This is complete history of 1.txt file. Note: you see commit “1.txt changes” that was pulled out (Also, you can see the actual file content was changed).

1. In this case, you will see the merge window.

* The good news: you’re aware that you have merge conflict, and that Git are making [“git pull’](#_Pull_(no_conflicts)) on your behalf.
* The bad news are:

1. You should resolve the conflict.
2. After conflict resolution, Git will continue the “git push” operation that will fail.
3. You should make “git push” again and if nobody has changed your remote repository, it will succeed.
4. You still receive other changes.

Note: You should “git push” twice. If somebody did make some changes on remote repository, you should merge your changes again.

**Summary: If you’re working with master only and you’re not alone, you will be forced to manage conflicts on every push operation.** Sometimes, it can be done automatically (see the drawbacks above) and sometimes you should do many additional operations to handle this.

**Alternative A:**

1. Make your changes.
2. Make separate commits.
3. [Git pull](#_Pull_(no_conflicts))
4. Resolve the conflicts if any.
5. If you have conflict go to 2 (maybe somebody push new changes when you did conflict resolution).
6. Push your changes.

Drawback:

* You make a lot of operation with Git
* You have potentially complex merges.

Note: There is still time window between your last pull without merges and push that when commit can sneak into.

**Alternative B:**

1. Make your changes.
2. [Git pull](#_Pull_(no_conflicts))
3. Resolve the conflicts if any.
4. If you have conflict go to 2 (maybe somebody push new changes when you did conflict resolution).
5. Commit&push your changes.

Drawbacks:

* You make a lot of operation with Git.
* Your merge conflict will be particularly nasty, because you have uncommitted changes. You may be required using [stashing](#_Stash_1) in order to even to begin the merge operation.

Note: There is still time window between your last pull without merges and push when new commit can sneak into.

**Alternative C:**

See [Feature branch](#_Feature_branch) section.

Note: For more elaborate Git branching model you can see for example here <https://nvie.com/posts/a-successful-git-branching-model/>

# Feature branch

Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something, no matter how big or small, they create a new branch.

Note: For more elaborate Git branching model you can see for example here <https://nvie.com/posts/a-successful-git-branching-model/>

The main idea that you have separate branch that only you commit into. In such a case, no conflict will occur and you can commit or even commit&push freely. When your feature is ready (and not on every push), you will take your time and you will merge your branch to the master, resolving any conflict that you’ll have.

Note:

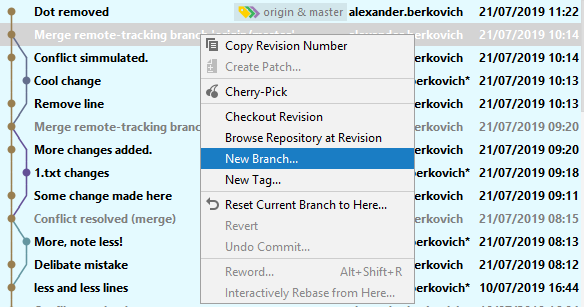
* In order to reduce the complexity of the merge of the feature branches it is advised to take the changes from master to your feature branch once in a while (say, every 2-3 days).
* If everybody works with feature branches, then you will take only chunks with compete features. It may contain [some code refactoring](#_Appendix_C_1), in such a case, you will see it as soon as possible and if you are doing the change in the same files, you will have 1 complex merge with your little change that will enable conflict detection and resolution a lot easier.

## New Branch

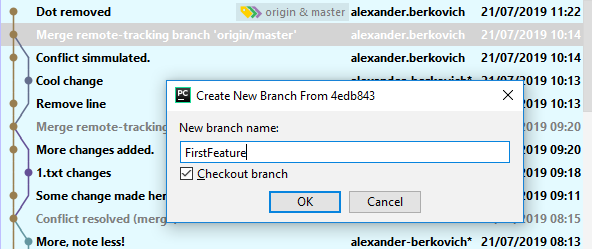
Note: When you do it, you should be aware on [what branch you’re currently working on](#_How_do_I). New branch will be created starting from your current branch. Typically, you should create new (feature) branches from master.

Note: You can create tag/branch from every commit (it doesn’t have to be last one in [commit history](#_Commit_History)).

Right-click on commit from which you want to start new branch -> New Branch…

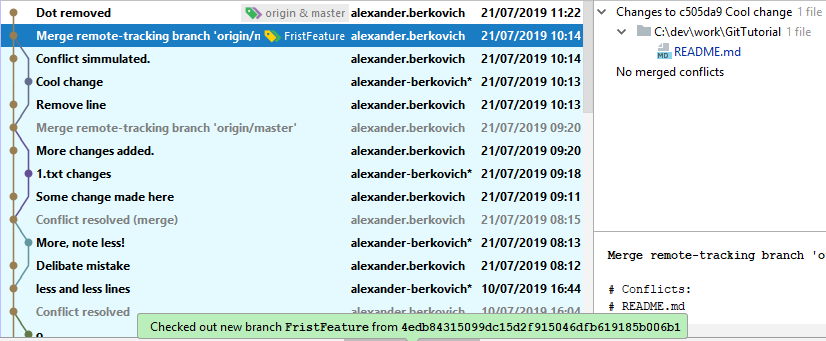


In the popup window write down name of the new branch.



Note: Checkbox “checkout branch” means whether you want to switch your [workspace](#_Stage/Indexing) to this branch. Usually, you want to do it.

When you will click ok, you will see:

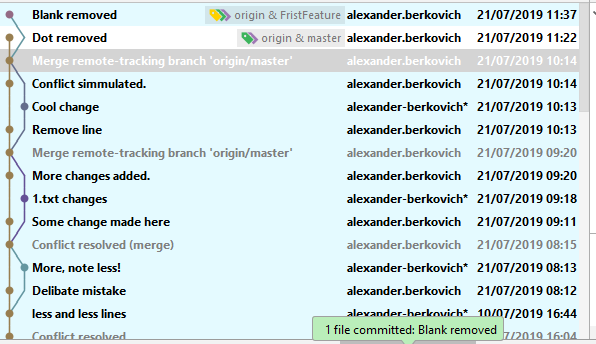


Note: If you use checkbox “checkout branch” you will see near left corner of the PyCharm the message “Checked out new branch *BranchName* from *id*”.

Note: While origin/master & master are in “Dot removed” commit, FirstFeature branch refers to “Merge remote-tracking branch ‘origin...” that is behind. It is perfectly fine.

Note:

* **FirstFeature branch is created only in your local repository.**
* **In order to create FirstFeature branch in remote repository, make change in some file and commit&push it.** This will create origin/FirstFeature branch as you can see in the commit history.



Note: Near “Dot removed” commit there 2 references now.

Note: “Origin&master means that 2 references points to this commit master and origin/master.

Note: “Origin&FirstFeature means that 2 references points to this commit FirstFeature and origin/FirstFeature.

Note: “Blank removed” commit’s (that I did on FirstFeature branch) parent is “Merge remote tracking branch “origin..” The latest is commit from which FirstFeature was started.

Now, you can continue to work on FirstFeature. You should be the only person that work on it, so no conflicts will occur.

For example, I make some additional commit (that changes test.docx file). The commit history is:



Note: “Additional commit on FirstFeatue branch” has parent “Blank removed (that we know is on FirstFeature branch).

Note: “Blank removed” commit has parent “Dot removed” that is on master and origin/master branch.

## Merging Branches

Suppose, we have following [commit history](#_Commit_History):



Note: “Additional commit on FirstFeatue branch” has parent “Blank removed (that we know is on FirstFeature branch).

Note: “Blank removed” commit has parent “Dot removed” that is on master and origin/master branch.

Let’s suppose out feature is ready and we want to merge it back to master. You should do the following steps:

Note: It is advised that you don’t have uncommitted changes.

1. Checkout the *destination* branch, that is the branch into which you want to merge. In our case, this is mater branch.

Note: “Checkout” means you want to switch your [workspace](#_Stage/Indexing) to *destination* branch. In other words, you want that your [workspace](#_Stage/Indexing) will contain file in the [workspace](#_Stage/Indexing).

Right-click on the project -> Git -> Repository -> Branches…



You will see the following popup window:



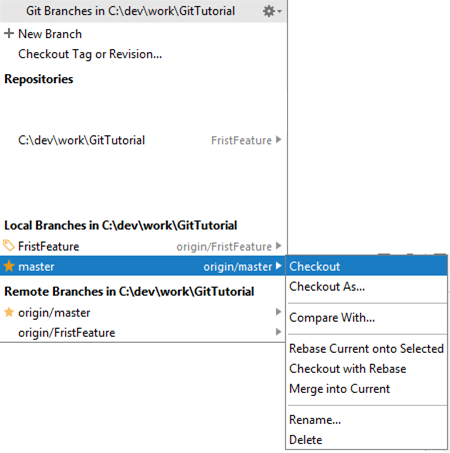
* “New Branch” – you can create new branch (see above).
* Checkout Tag or Revision…. – you can switch your workspace to state that was when you’ve created tag or even commit.
* Repositories – the list of Git Repositories in PyCharm. You can see that I’m working on FirstFeature branch in GitTutorial,
* Local Branches in…

List of local branches (branches that was created in local repository): FirstFeature and master. You can also see that FirstFeature is “connected” to origin/FirstFeature and master – to origin/master.

* Remote Branches in…

List of remote branches (branches that was created in remote repository) – origin/master and origin/FirstFeature.

**We will choose master (local branch). Click on checkout.**

****

**Now, your workspace reflects master branch.**

1. **Optionally, you can see what is difference between branches.**

Right-click on the project -> Git -> Repository -> Branches…

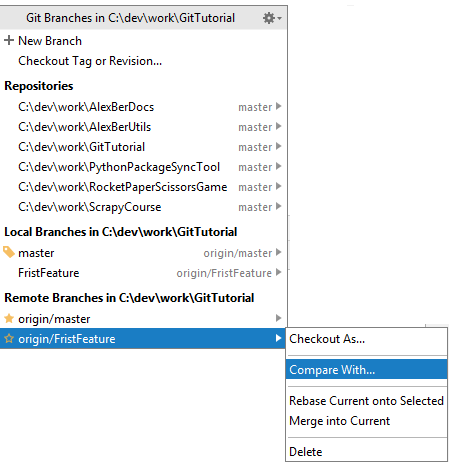


You will see the following popup window:

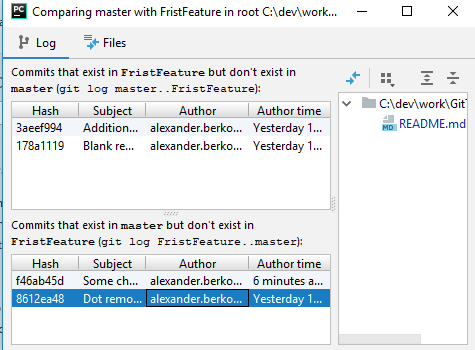


Note: In Repositories you see that C:\dev\work\GitTutorial reflects master branch.

**Now, chose “Compare with..”**

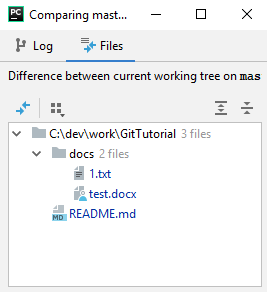


You will see the following window:

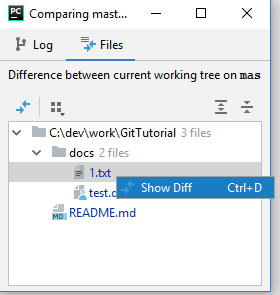


Here you can see commits that exists in one branch and not in another. You can select commit that you’re interesting in and see the file it contains. You can also see the change in the file.

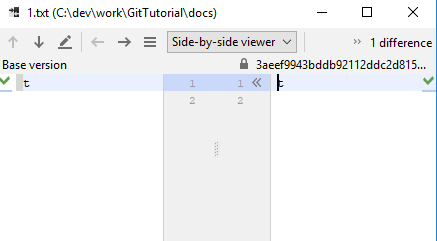
If you select Files tab, you will see the following window:



Now, you can select each file, right-click on it and chose “Show Diff”.



You will see the difference windows:



**3. Merging feature branch to the master.**

Right-click on the project -> Git -> Repository -> Branches…

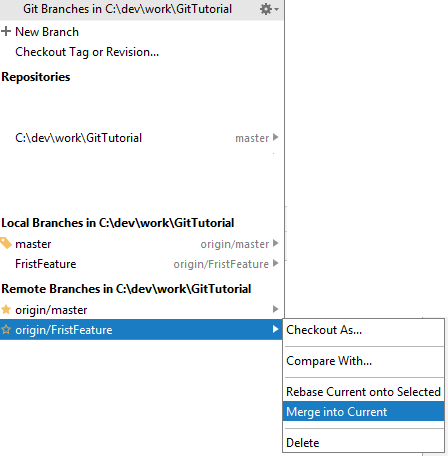


You will see the following popup window:



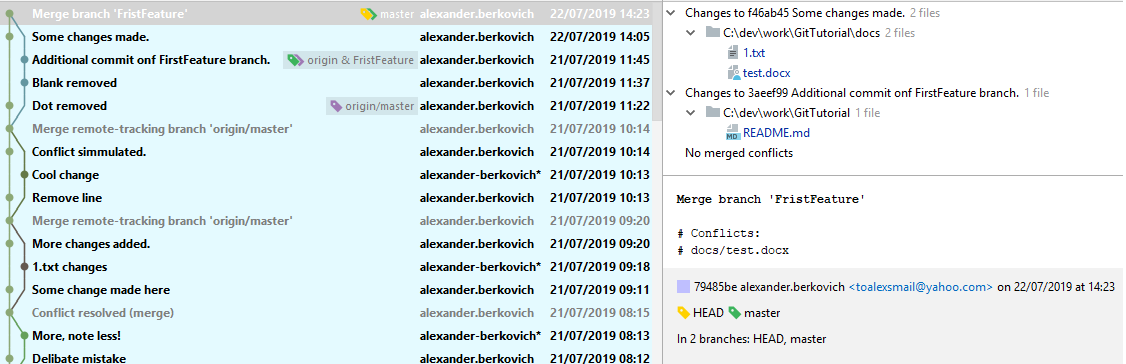
Note: In Repositories you see that C:\dev\work\GitTutorial reflects master branch.

**Choose feature branch you want to merge (FirstFeature) in our example, in the popup chose “Merge into current”.**

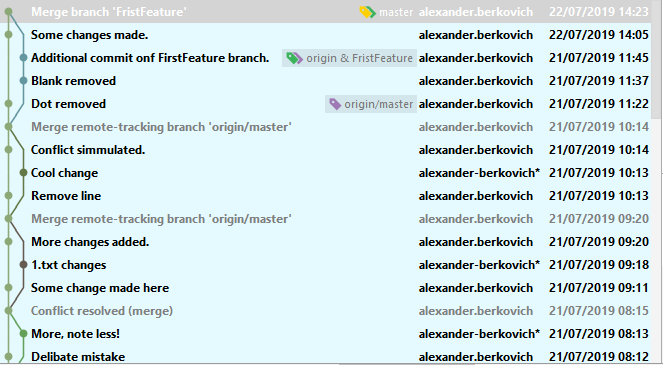


If for some files merge can’t be done automatically, you will see the regular merge window. Did [regular merge](#_If_you_have_1).

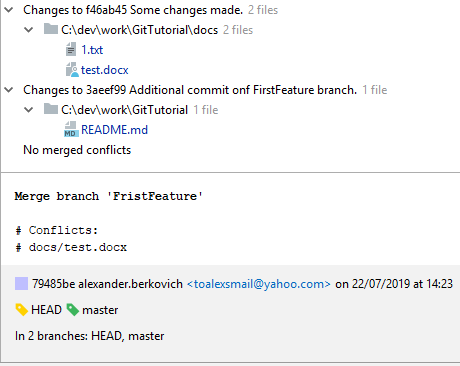
You will see your “merge commit” in the [commit history](#_Commit_History):



Zoom in to the left panel:



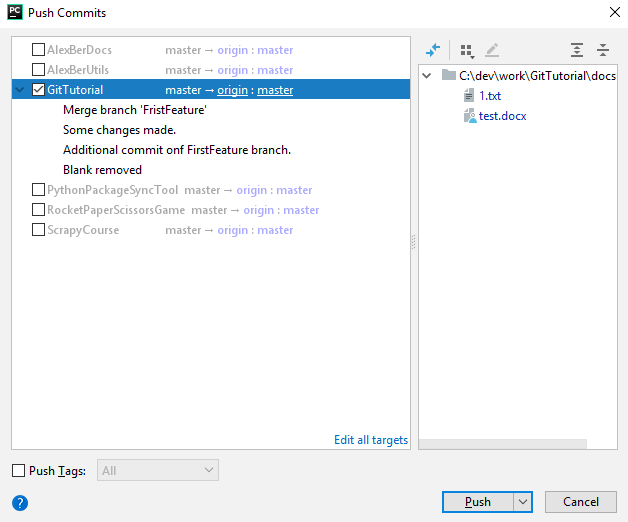
Zoom in to the right panel:



Note: Now master is ahead of origin/master.

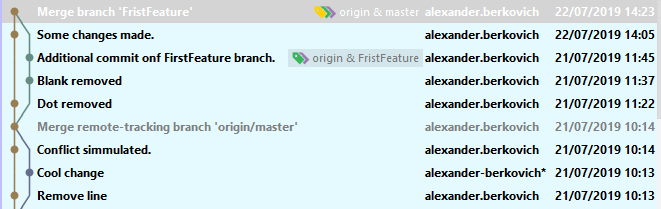
4. **Now, you should** [**push**](#_Push_(no_conflicts)) **your changes.**

Note: It is better to [**“git pull”**](#_Pull_(no_conflicts)) **first.**



Note: Here you see all commits made in your feature branch plus merge commit.

5.**Optionally, you can look on** [**commit history**](#_Commit_History)**:**



Note: Now, master and origin/master points to the same commit.

Note: If for whatever reason you want to postpone full conflict resolution you can use this [advanced](#_What_if_you) technique.

Note: [Appendix C](#_Appendix_C_1) contain code example that show how merging difference branches can go wrong, albeit you’re doing all according to the best practice. This example show why **it is important to merge master to the feature branch once in a while.** It is better to have such problem in the feature branch and not to break the master.

## Rename branch

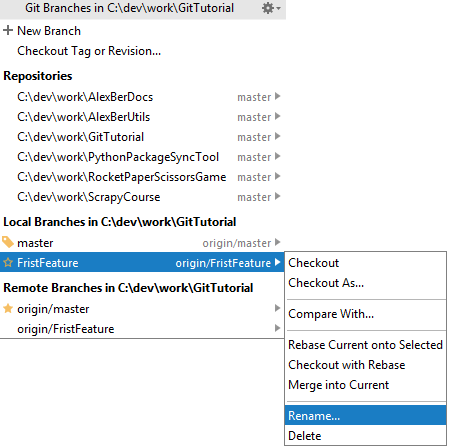
Right-click on the project -> Git -> Repository -> Branches…



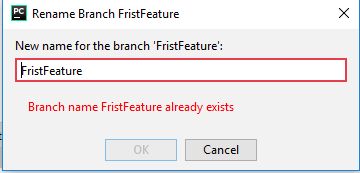
You will see the following popup window:



**Select the branch you want to rename. Click on Rename.**

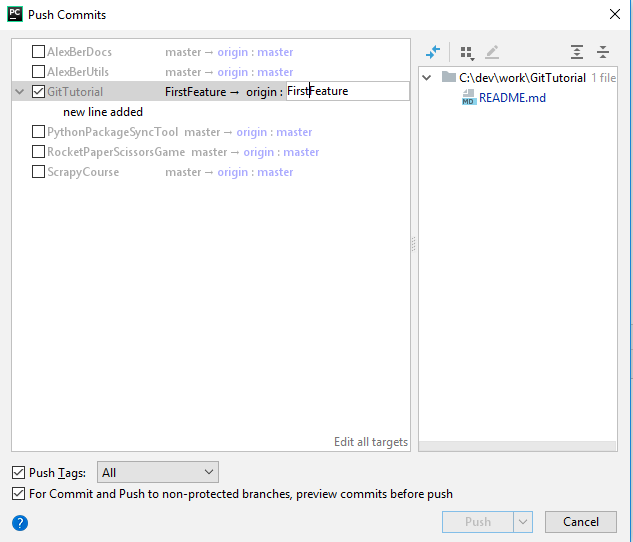
****

**In the popup window change the name:**



**Note: Probably, you will also want to change the name of the branch also in remote repository** (origin/FirstFeature)**.**

**In the** [**Push**](#_Push_(no_conflicts)) **commit window change the name of origin/FirstFeature**



## Delete branch

Note: You should checkout another branch, not the branch you about to delete. If you are in the branch that you about to delete, you willn’t see the option to delete it (only to rename).

Right-click on the project -> Git -> Repository -> Branches…

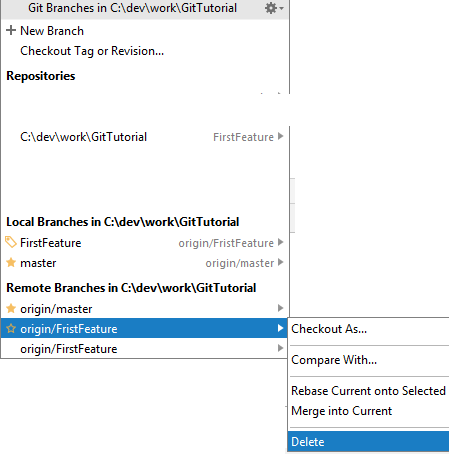


You will see the following popup window:

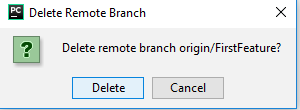


**Select the branch you want to delete. Click on Delete.**

**In general, you will want to delete also remote branch. Just chose in the list above.**

****

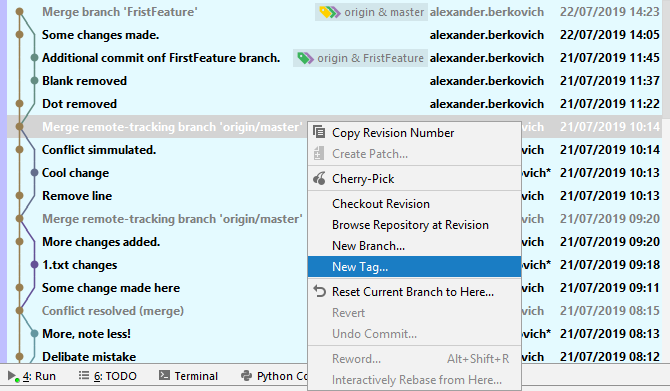
**Click on Delete button.**



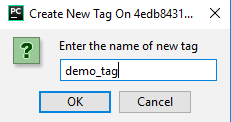
## New Tag

Note: You can create tag/branch from every commit (it doesn’t have to be last one in [commit history](#_Commit_History)).

Right-click on commit from which you want to start new branch -> New Tag…



Fill the tag name:

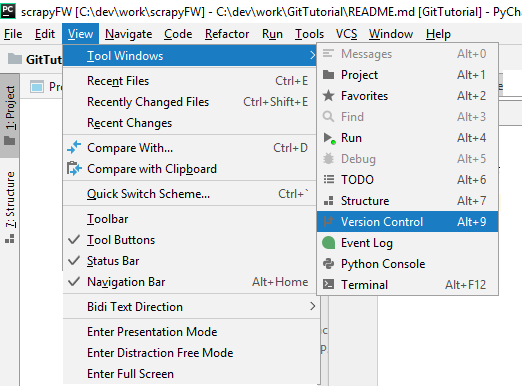


Note: In the title of the window you can see commit id.

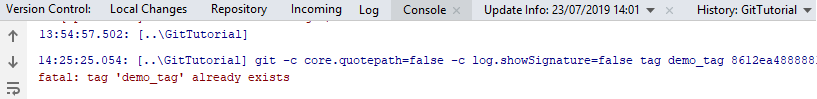
Note: Tag now is created only on local repository.

**Note: For some unknown reason, if you’re creating tag with existing name, you don’t see any notification that this operation fails.**

* You just see, that tag is not created on the commit (No **** appeared).
* If you want to verify that this operation has failed, you can go to Console tab. In the menu View->Tool Windows->Version Control.



Go to console tab



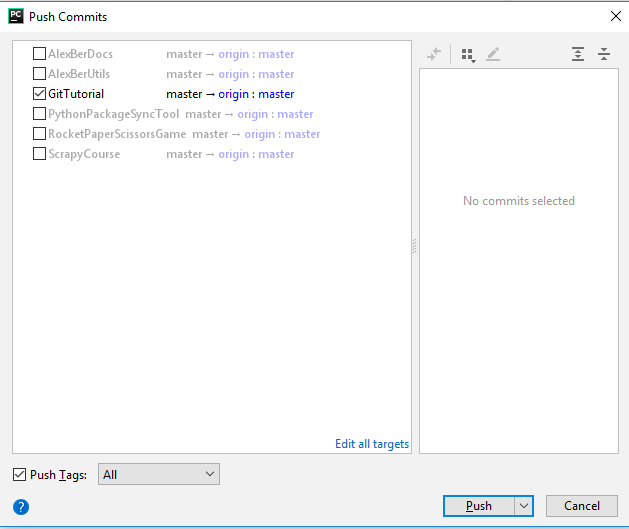
If in the last line you see message: ”fatal: tag ‘demo\_tag’ already exists” it means that you was unable to create tag.

**Now, you should** [**push**](#_Push_(no_conflicts)) **the tag to remote repository.** It will create tag on remote repository. Alternatively, you can execute in the cmd\Terminal

git push --tags

Note: It is better to [**“git pull”**](#_Pull_(no_conflicts)) **first.**

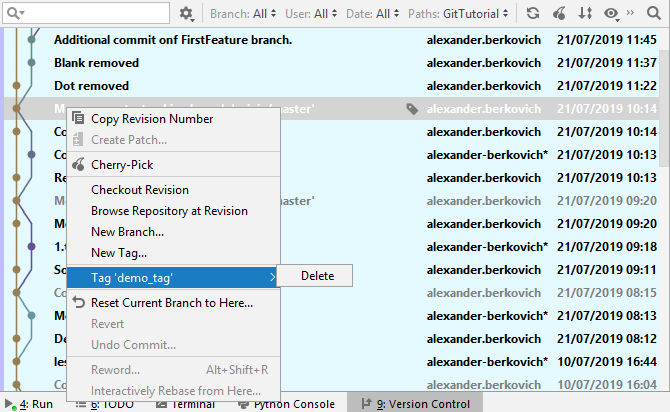
**In the Push Commits Window select check-box near “Push Tags”. Leave “All” without change.**



## Delete Tag

**Check commit that have tag on it ()**

**Right-click on it - chose Tag “tag\_name” - > Delete.**



In the right corner you will see



Note: In general, it is good idea also to delete tag on remote repository. Click on “Delete on Remote”. Alternatively, you can execute in the cmd\Terminal

git push --tags

Note: It is better to [**“git pull”**](#_Pull_(no_conflicts)) **first.**

# Stash

Stashing involves *recording the difference between the HEAD commit and the current state* of the working directory. Changes to the index can be stashed as well.

Note: Don’t be confuse with shelve.

## What if you’re in the middle of the work and you want to make “git pull”?

Let’s be more concrete. You’ve changed couple of the files and you’ve been asked to pull the changes (maybe, there was some bug fix that prevent to some of the functionality to work properly that you need).

* Possible solution will be to commit your incomplete work, pull the changes and resolve possible conflicts.

There are 2 problems with this solution:

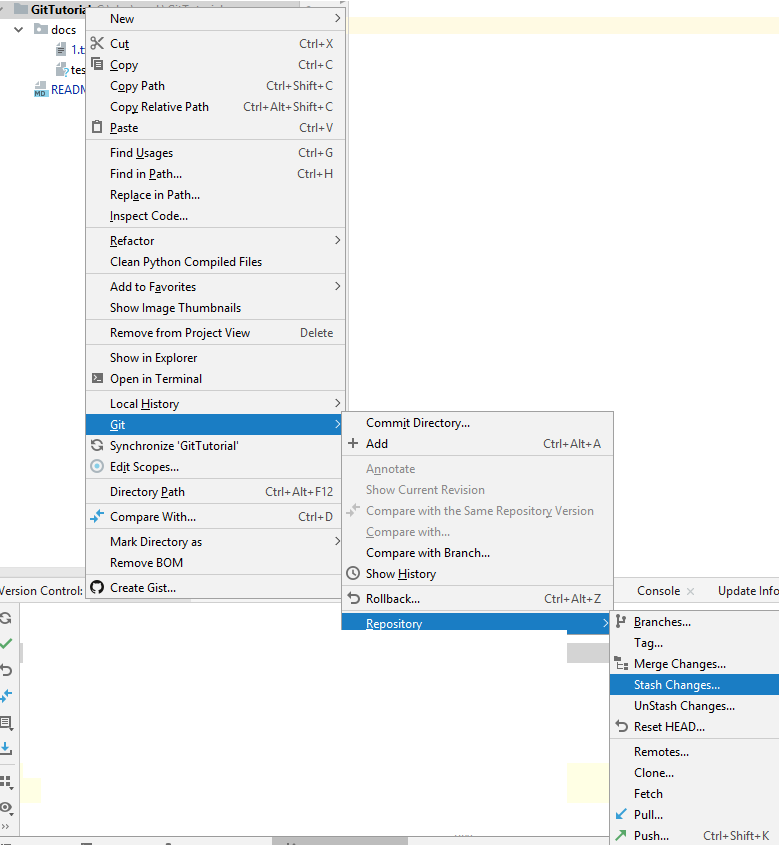
1. You will commit with incomplete work (the application may even not be able to compile)
2. Resolving merge conflict may be difficult (For example, if you modify 10 files and you are in the middle of refactoring).

* Another alternative is to manually copy your changes file aside, revert you changes in the PyCharm project, make [“git pull’](#_Pull_(no_conflicts)), potentially resolve merge conflict (which should be in usual complexity), redo you work (for example, by copy&paste relevant parts of the file from you aside directory) – this is essentially *manual* merge
* Stash make the (manual) process described above more manageable.

Let assume that you’ve changed 2 files 1.txt and REAMDE.md.

Now, you’ve been told to make [“git pull’](#_Pull_(no_conflicts)).

1. **Right-click on the project-> Git->Repository->Stash changes**…



You should see the following window:



In the message field fill something that will help you to identify this stash.

Note: If you want to stash local changes and bring the changes staged in the index to your working tree for examination and testing select check-box “keep index”.

1. **Click on “create Stash” button.**

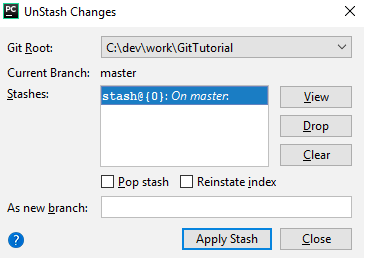
*Now you HEAD and master points to the same commit. You working tree is also as HEAD.*

1. **Now, make** [**“git pull’**](#_Pull_(no_conflicts)) **operation, resolving conflicts of necessary.**
2. **Confirm that you don’t have uncommitted code.**Commit/revert/stash you changes. Failing to do so may result with unStashing failure.
3. **Right-click on the project-> Git->Repository->UnStash changes..**.

Note:UnStash operation may fail if you have conflicts. This happens because conflicts are stored in the index where you can no longer apply the changes in their original state.

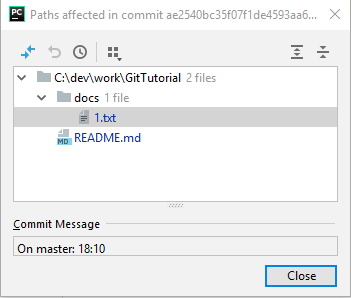
****

You should see the following window:

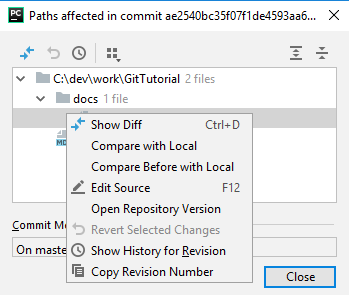


Let’s go through your options.

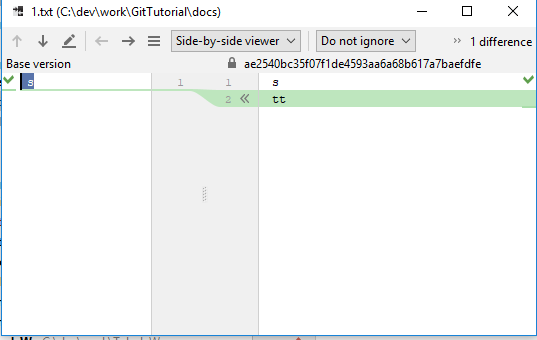
* “View” button will open new window where you can examine what files where stashed and to see the content of the files and it’s diff.



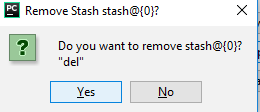
Right-click on the file, you will see the following menu:



Clicking on “Show Diff” will let you see the differene:



* Marking existing stash and clicking on “Drop” button will show you confirmation window:



Click yes, to remove the stash.

* Clicking on “Clear” button will show you confirmation window:

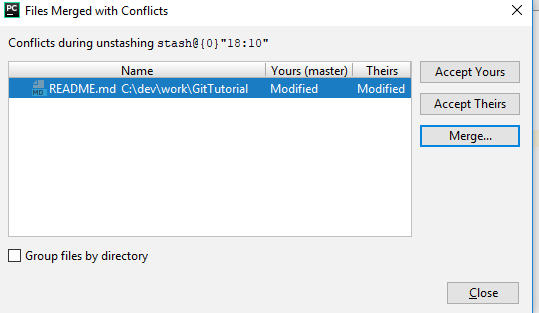


Click yes, **to remove all stashes.**

* Check-box “Pop stash” means that when we will press “apply stash” button it will apply the stash and remove selected stash from the list of stashes (like as we click on “Drop” button after “Apply Stash” button).
* Select check-box “Reinstate index” to apply stashed index modifications as well.
* “As new branch” field - If you want to create a new branch on the basis of the selected stash instead of applying it to the branch that is currently checked out, type the name of that branch

I will show the flow when you don’t change nothing, but **just click on “Apply Stash” button.** This should unStash your changes unless:

1. **If you have conflict with files that are commited you will see the merge window:**

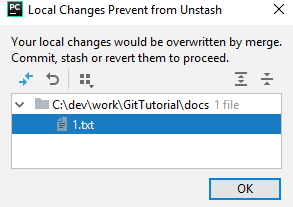


Make [usual merge process](#_If_you_have_1) for conflict resolution.

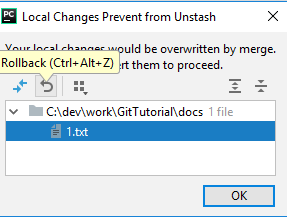
1. **You have some uncommitted change and you have conflict with one these files with unStashed filles.** Than you will see the following popup window on the right corner:

****

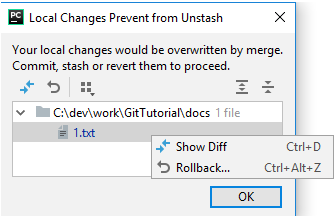
Clicking on the “View the” link will open the following window:



* **You can mark the file and click on “Rollback”.**

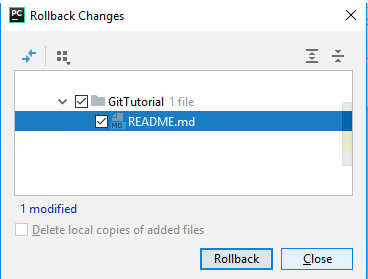


* Or **you can right click on the file**



To see the diff and then rollback.

**Clicking on “Rollback**” will open new window:



**Click on “Rollback” button.**

Note: if unStash fails besides rollback on selecting files you can 2 other options:

1. [Git reset HEAD](#_What_if_I’ve)
2. Commit/rollback/stash the changes and reapply the stash.
3. Create branch from stash and then merge it to the current branch.

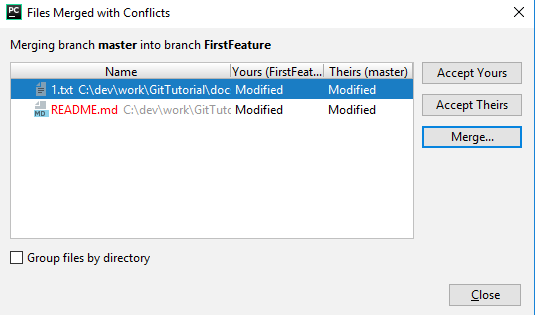
## What if you have complicated merge and you prefer to make it in more controlled way?

**Note: The technique that is described here is not considere best practice for the work with Git.**

Note: This technique contains some manual steps that deliberately hide information on what you’re doing from the Git (Git will think that the merge is over and you will “secretly” continue the merge in some point in future).

Note: This technique should be used only when you want to postpone full conflict resolution for some reason.

Let’s be concrete. We’re pulling changes from master to FeatureBranch and we have conflict to resolve. We have 2 files with conflicts.



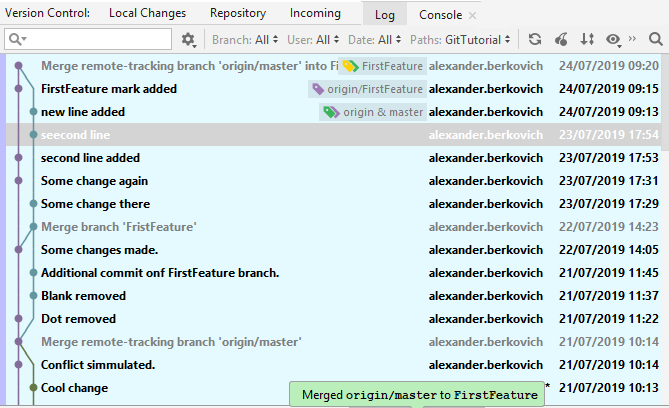
After examine the merge changes, I’ve figure out that README.MD has some nasty conflicts (maybe I should communicate with somebody in order to correctly resolved the issue and this person is currently unavailable). I want to defer this nasty merge.

After watching on 1.txt I realize that I can merge 1.txt.

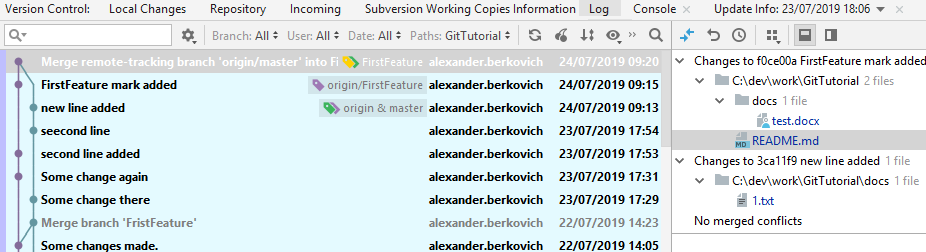
1. [**Resolve conflict**](#_Merge) **of 1.txt in regular way.**
2. **For README.md click on “Accept Theirs” button.**

Note: You temporarily lose you changes, but you will recreate them and put them to Stash in a while.

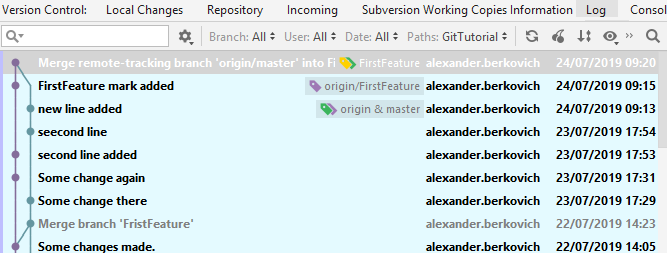
You should see merge commit in the Log:



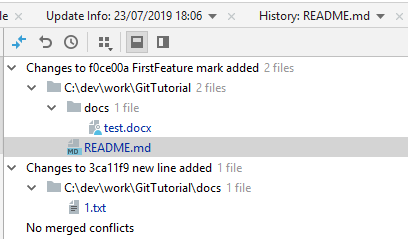
1. **Click on merge conflict (the top line).** In the right panel you should see changes to the files that where done during this commit. **Select all files that you have some nasty conflicts, README.md in our case, double-click on them.**



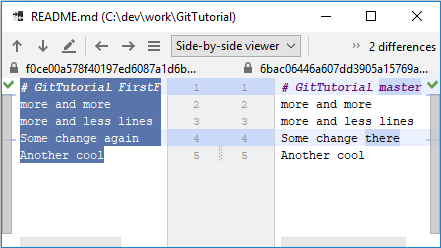
Let’s zoom in. Left Panel:



Right Panel:

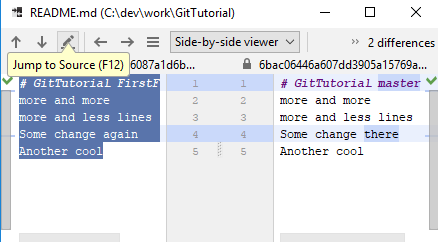


1. **In the opened popup, copy to buffer all text from the left panel.**

****

Note: Left panel has content of parent commit (last commit of the feature branch before merge) and right content has content (of the selected file) of the current selected commit (merge commit).

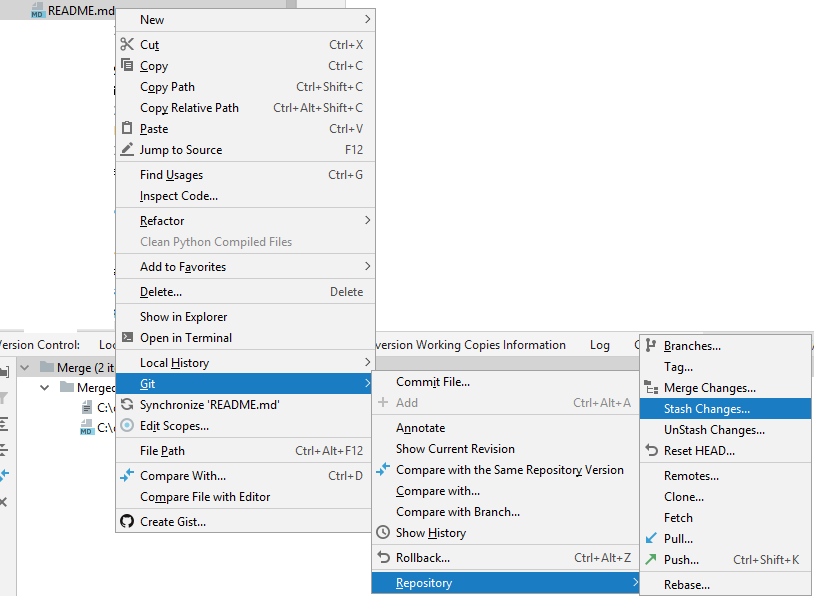
1. **Click on the third icon from left in the menu.**

****

**This will open README.md in the editor.**

1. Now, **paste the content of the buffer.  
     
   Note: You change the content of the file to reflect your changes and you overwrite theirs changes (from master branch).**
2. **Stash REAMDE.md**

**Right-click on REAMDE.md -> Git->Repository->Stash changes**…



You should see the following window:



In the message field fill something that will help you to identify this stash.

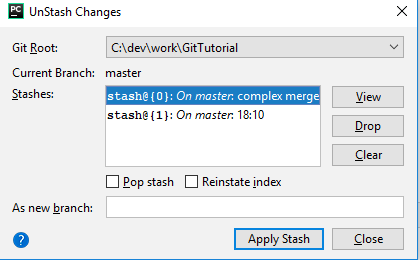
1. **Click on “create Stash” button.**

*Now you HEAD and master points to the same commit. You working tree is also as HEAD.*

1. **Now, you can continue to work in regular way (commit, push, etc).**Note: README.md includes only theirs changes. You effectively lost all your changes.
2. **Confirm that you don’t have uncommitted code.**
3. **Right-click on the project-> Git->Repository->UnStash changes..**.



You should see the following window:



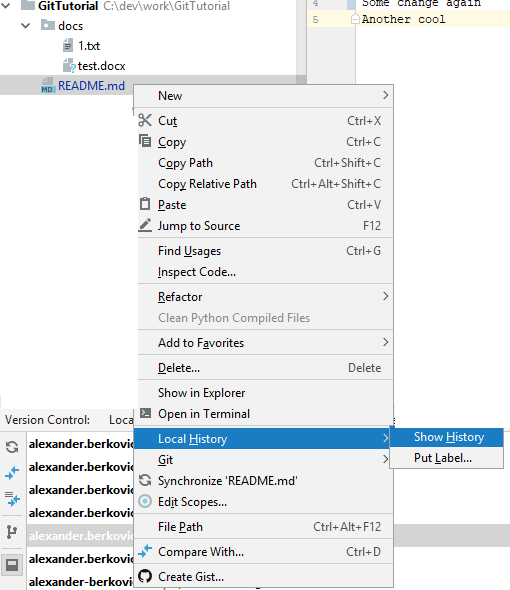
1. **Select the stash you’ve created and apply it.**

Note: UnStash operation should succeed. As far as Git concern REAMDE.MD contain changes that you have done (the merge conflict is part of your feature branch; Git doesn’t care that the content of REAMDE.MD you take from theirs (master)). When you stash you changes you pasted the code from the buffer. As far as Git concern, you just *manualy* rewrite the whole file (with content that is happen to be taken from theirs (master)), but Git don’t know this). So, git UnStash will reapply you change – namely will again rewrite the whole file. Again, as far as Git concern there is no conflict here and no merge should be done. You should do merge manually.

Note: You can see the files that you’ve stashed by clicking on view button. They also should be colored blue in PyCharm.

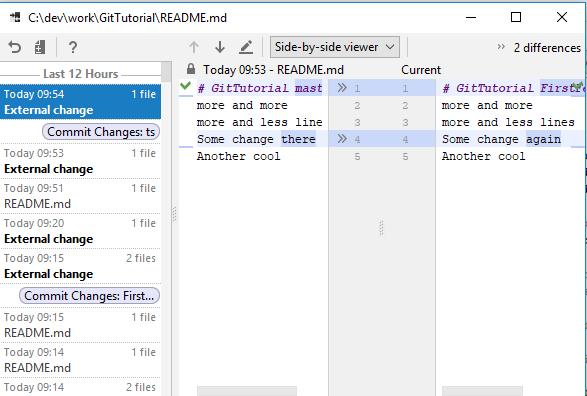
1. **For every file that you’ve stashed (**REAMDE.MD in our case)

**Right-click on the file -> Local History -> Show History.**



1. **Select top change.**

You will see following window.

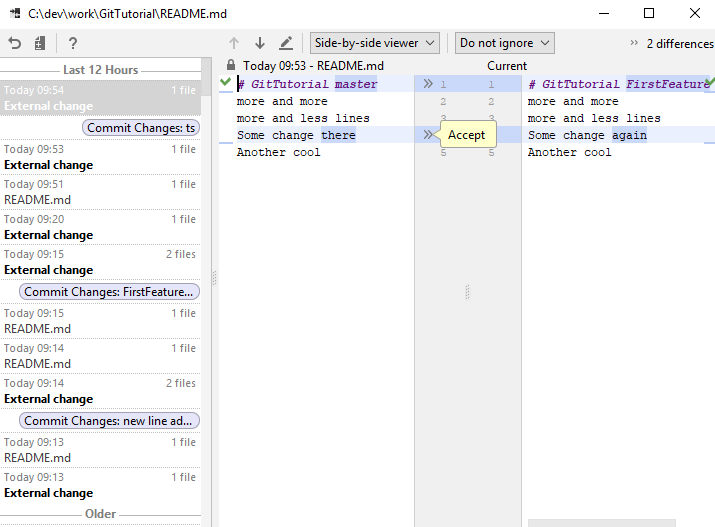


Note: Now, README.md includes only theirs changes (Current…, right panel).

In the left panel (Today…) you see the unstashed changes (theirs, from master).

In the right panel (Current…) you see merged changes (yours, from feature branch).

Clicking on  you can take changes from master and apply them on feature branch.



1. **Now** [**commit**](#_Commit_File) **and** [**push**](#_Push_(no_conflicts)) **your changes.**Note: As far Git concert this is regular commit. It will not be recognized as merge commit.

# Create New Project

Usually, you have some main repository and you want to contribute to it. In such scenario see section [“from existing project (git clone)”](#_From_existing_project_1)

If you have some files and you want to start manage them on new Git repository it is better to use [“From Scratch (git init)”](#_From_Scratch_(git) section.

Note: You can still use [“from existing project (git clone)”](#_From_existing_project_1) , but you will need to do some extra steps.

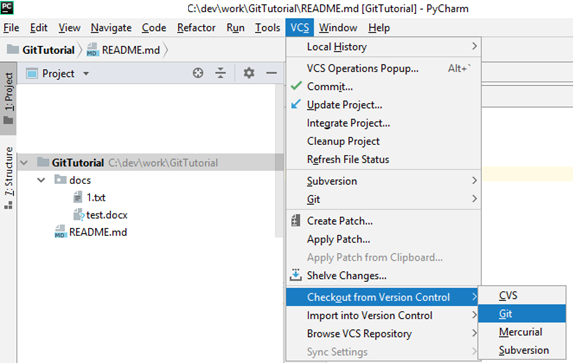
Section [“from Scratch using git commands (git init)”](#_From_Scratch_using) is optional.

* It is variation of [“From Scratch (git init)”](#_From_Scratch_(git), it uses cmd/terminal.
* This way is more quick.
* You have opportunity to see what PyCharm are doing under the hood.

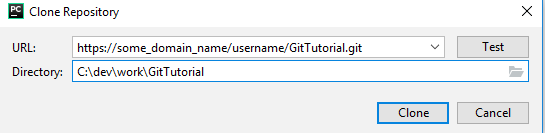
## From existing project (git clone)

You should know somehow full URL for the remote repository that you want to clone. It should be something like <https://some_domain_name/username/GitTutorial.git> where GitTutorial is the name of the repository to clone, Some\_domain\_name is domain name where you host your repository (GitHub, for example), uersname is user name on remote repository.

1. **On the menu click on VCS->Checkout from Version Control->Git.**

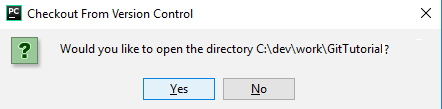


You will see this popup window. In the URL fill the URL of remote repository.



1. **Optionally, you can click on Test button to check whether link is valid.**
2. **Click on Clone to clone the repository.**

You will see following popup window:

****

1. **Click on Yes button.**

You will see the following window:



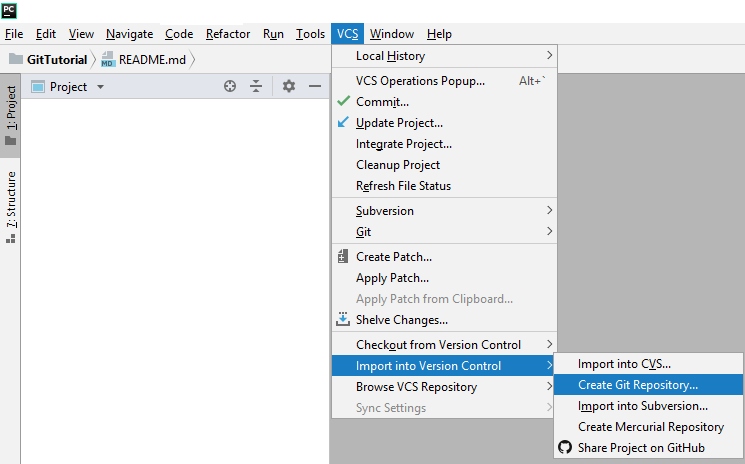
This windows asks you how you want to import Git repository as PyCharm project. Whether you want to create separate window with this PyCharm project (first option) or you want to append this project to currently opened window. The checkbox Add to currently opened projects appears when you open a new project in a (current) window, where another project is already opened.

1. **Click on ok.**

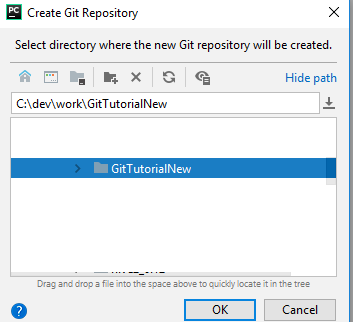
## From Scratch (git init)

Let suppose that you want to create new repository, let say GitTutorialNew.

1. **You should create empty remote repository**, something like <https://some_domain_name/username/GitTutorialNew.git> Some\_domain\_name is domain name where you host your repository (GitHub, for example), uersname is user name on remote repository.
2. Create empty directory GitTutorialNew.
3. On the menu **select VCS->Import into Version Control->Create Git Repository…**



**In the opened popup select path** to GitTutorialNew directory. New PyCharm project (and Git repository) will be created there.



1. **File->Open**

****

You will see the popup window.

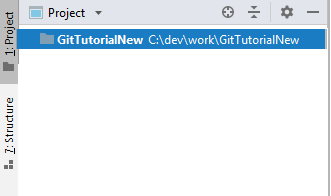


1. **Check GitTutorialNew.**
2. **Click on OK button.**
3. You will see the following window:

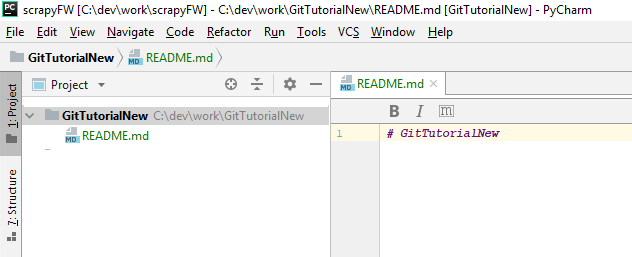


1. This windows asks you how you want to import Git repository as PyCharm project. Whether you want to create separate window with this PyCharm project (first option) or you want to append this project to currently opened window. The checkbox Add to currently opened projects appears when you open a new project in a (current) window, where another project is already opened.
2. **Click on ok.**

Note: you should see empty project in PyCharm.

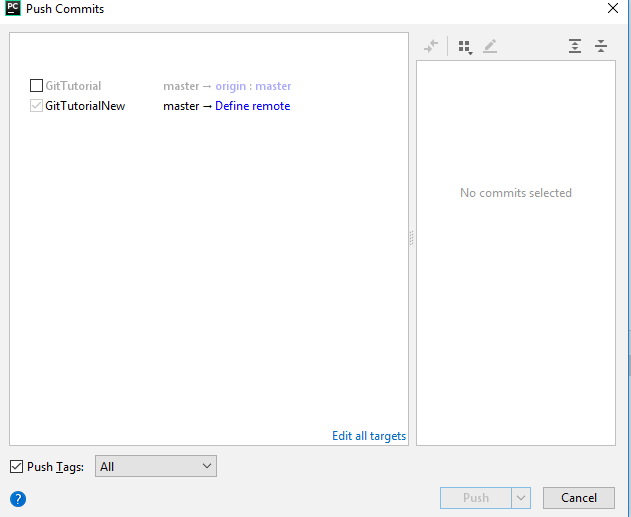


1. **Create new file README.md (or any other file). Put some text their (it should be non-empty).**

****

Note: This file is highlighted as red. This means this file was “git added” and now it was modifed.

1. [**Commit**](#_Commit_File) **file.**Note: Be aware that PyCharm will want to commit README.md and (hidden) .idea directory.
2. Now, there is 2 ways to make Push.
3. [**Push**](#_Push_(no_conflicts)) **Commits.**

****

1. **Click on “Define remote” link**.

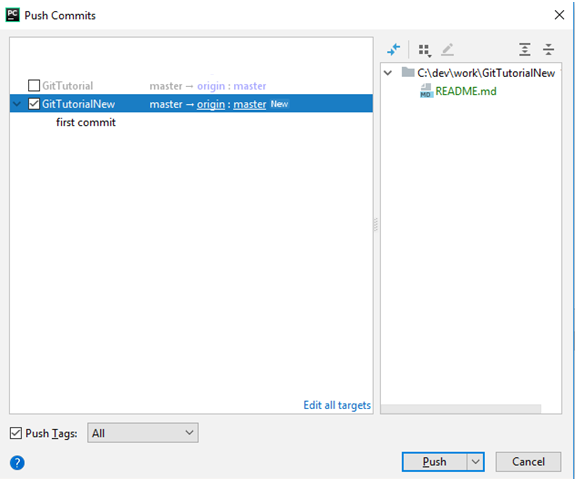
Note: if alias to remote link is defined you will see something like master -> origin : master.

You will see the popup windows below.

1. **Fill URL to your remote repository.**

****

1. **Click on OK button.**
2. Now Push Commits Window should change to



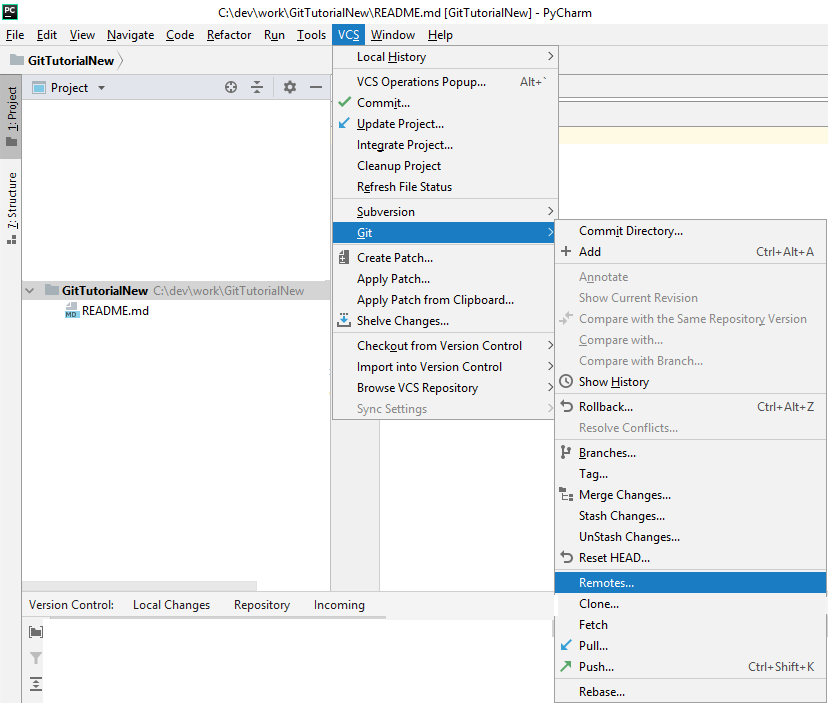
Note: Near GitTutorialNew there is mapping master -> origin : master. There is also New label near it.

1. **Now click on Push button.**

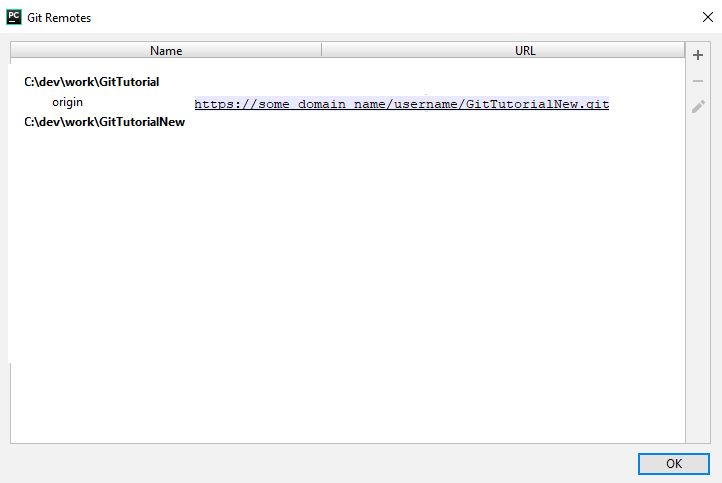
Note: You can define or change alter remote origin explicitly.

* **On the menu click on VCS->Git->Remotes…**

.

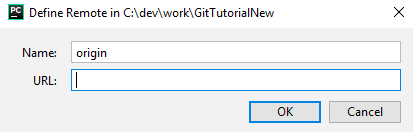


You will see following popup windows:



* **Select C:\dev\work\GitTutorialNew line and click on + (or on  if you want to change the link to remote repository).**

You will see the following popup windows.



* **Fill URL to your remote repository.**

****

* **Click on OK button.**
* **Click on OK button.**
* Important: In order that this change will take a place**, you should** [**pull**](#_Pull_(no_conflicts)) from this Git repository before [push](#_Push_(no_conflicts)).

## From Scratch using git commands (git init)

Let suppose that you want to create new repository, let say GitTutorialNew.

1. **You should create empty remote repository**, something like <https://some_domain_name/username/GitTutorialNew.git> Some\_domain\_name is domain name where you host your repository (GitHub, for example), uersname is user name on remote repository.
2. **On the cmd/Terminal type**

* **Create directory, let say GitTutorialNew,** where you want your local repository (and PyCharm project) will be located.
* **Go to this directory**, type something like

cd C:\dev\work\GitTutorialNew

* **Type the following commands:**

git init

echo "# GitTutorialNew" >> README.md

git add README.md

git commit -m "first commit"

git remote add origin https://some\_domain\_name/username/GitTutorialNew.git

git push -u origin master

Explanation:  
  
Note: I will point to appropriate step in the [previous](#_From_Scratch_(git) section for reference.

* “Git init” creates Git Repository (p. 3). This is really “.git” directory created inside GitTutorialNew directory.
* Second line just creates some file (p.10).

* “Git add” and “Git commit” (p.11).
* “Git add” adds file created in previous step to Git manage, now [Git ”is aware](#_Stage/Indexing)” that this file exists in your workspace.
* [“Git commit”](#_Commit_File) save this file in your local repository.

Note: “-m” is shorthand from message. Message “first commit” will be associated with the commit that contain only file README.md

Note: First run of “Git commit” also creates your local master branch (“git init“ doesn’t do it). This is called root-commit.

* “Git remote add origin” (see note).
* This command defines alias “origin” for remote repository.

Note: The name “origin” is just a conventional name, but it is widely used by everybody. This documents assume it without mentioning.

* This defines that alias origin will be resolved to actual link https://some\_domain\_name/username/GitTutorialNew.git

1. **File->Open**

****

You will see the popup window.



1. **Check GitTutorialNew.**
2. **Click on OK button.**
3. You will see the following window:



1. This windows asks you how you want to import Git repository as PyCharm project. Whether you want to create separate window with this PyCharm project (first option) or you want to append this project to currently opened window. The checkbox Add to currently opened projects appears when you open a new project in a (current) window, where another project is already opened.
2. **Click on ok.**

# Appendix A

This section contains **rough** “translation” of familiar terms from SVN to Git.

This may be useful if you know SVN and start to use Git. Note, that in any place in the document, except introduction, there is no assumption on SVN knowledge.

* Trunk in SVN is roughly corresponds to master on Git.
* Update (without conflicts) in SVN – to pull.
* Commit (without conflicts) in SVN to (add file)+commit+push.
* Checkout of entire repository in SVN is clone (there is also another way).
* Branch on SVN is chain of commits. Merge of 2 branches is simply created new commit that copies changes from another branch. This is “hard branches”.  
  Branches in Git are not “hard branches”, *they are only pointers on commits*.
* Commit in Git can have more than one parent.

More advanced (you can skip it, especially in your first read):

* Commit history saved in different way than in SVN. I will point only on 1 of them:
* On SVN history is stored on your central repository. Each operation on it, requires call to it. On SVN each local repository has full copy of the history. Each operation on it are local.
* Update (with conflicts) in SVN – pull + resolve/merge conflicts+commit. Commit is used to conclude merge.
* Commit (with conflicts) in SVN (typically due SVN commit which fails as “out of date”, svn update resolve the issue, svn update typically it works without conflicts, you can commit after this) – (commit can’t case conflicts, you are working locally) git push + resolve conflicts.

# Appendix B

## Troubleshooting

### If you close merge resolution windows and when you want to commit your merge changes, you don’t see any file that change

Type in the terminal/cmd

git commit

See next section for more details.

### If you successfully resolve all merge conflicts, but you receive error message “You have not concluded your merge (MERGE\_HEAD exists)”

Type in the terminal/cmd

git commit

or to abort merge

git merge --abort

Note: If you should absolutely sure that all conflicts where resolved, you can try also.

del /q /s /f .git\MERGE\*

Or you can type

rm -fr .git/MERGE\*

See <https://stackoverflow.com/a/21829654>

### How do I know what branch is currently checkout?

There is 3 ways to do it.

1. Right-click on the project -> Git -> Repository -> Branches…

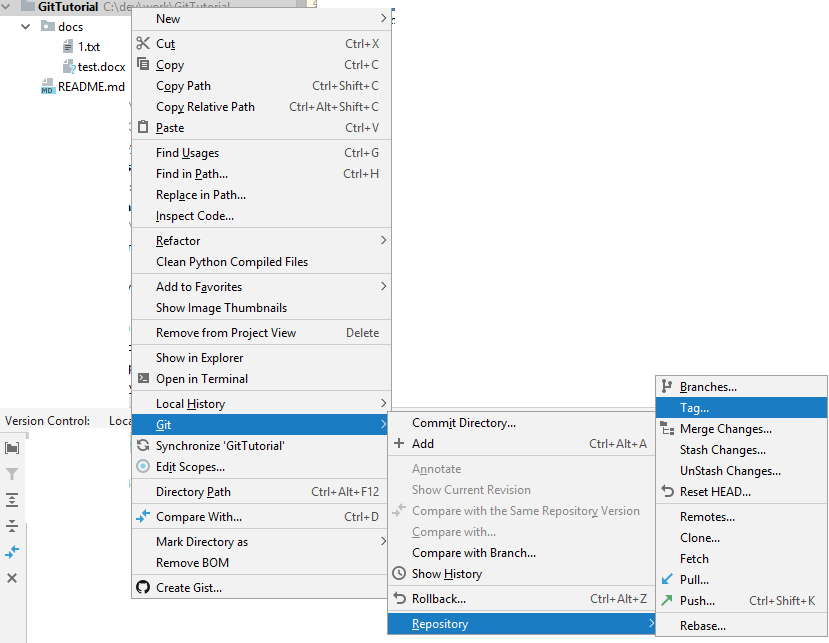


You will see the following popup window:

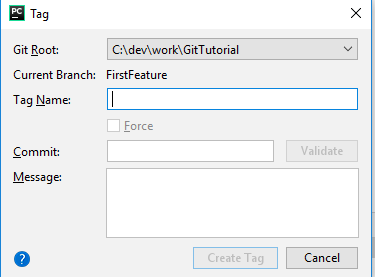


Look on “Local Branches in..” section. Marker  is placed near currently checkout branch (FirstFeature in this example).

1. Right-click on the project -> Git -> Repository -> Tags..



You will see the following popup window:



Read the line “Current Branch”.

c) Open cmd\Terminal

git branch

You will see something like this



Asterisk near the branch name show the current branch name.

### If Pull failed, you should merge your changes first and then

Type in the terminal/cmd

git commit

Close opened editor, ignore it.

You should have additional commit “Merge branch ‘master’ of”.

And push your changes in regular way.

Note: See also [Troubleshouthing](#_If_Pull_failed,) version below for another option. In this option you have auto-generate message for merge commit.

Note: For additional merge problem, See section on [*Merge*](#_Merge).

### **Git Delete Last Commit**

See <https://superuser.com/a/319587>

Once in a while late at night when I ran out of coffee, I commit stuff that I shouldn't have. Then I spend the next 10 - 15 minutes googling how to remove the last commit I made. So after third time I wanted to make a record of it so I can refer to it later.

If you have committed junk but not pushed,

git reset --hard HEAD~1

HEAD~1 is a shorthand for the commit before head. Alternatively, you can refer to the SHA-1 of the hash you want to reset to. Note that when using --hard any changes to tracked files in the working tree since the commit before head are lost.

If you don't want to wipe out the work you have done, you can use --soft option that will delete the commit but it will leave all your changed files "Changes to be committed", as git status would put it.

Now if you already pushed and someone pulled which is usually my case, you can't use git reset. You can however do a git revert,

git revert HEAD

This will create a new commit that reverses everything introduced by the accidental commit.

### Delete all local branches which were delete on the server

git remote prune origin

With this simple command you delete all local branches which were deleted on the server.

Note: With --dry-run option, report what branches will be pruned, but do not actually prune them.

### List all tags

See <https://stackoverflow.com/q/1028649>

Lists it out of order relative to all of the other tags  
  
git tag -l

To see [commit history](#_Commit_History):

git log --oneline --decorate --graph

### Remove remote tag/branch

See <https://stackoverflow.com/a/21300116>

git push origin :old\_branch

Note: Usually, you will want also to remove branch/tag also locally

git tag --delete old\_branch

git branch --delete old\_branch

### Create remote tag/branch from local tag/branch

Note: Here it assumed that new exists in local repository (as tag or branch).

See <https://stackoverflow.com/a/21300116>

git push origin new

### Rename tag

See <https://stackoverflow.com/a/5719854>

Here is how I rename a tag old to new:

git tag new old

git tag -d old

git push origin :refs/tags/old

git push --tags

The colon in the push command removes the tag from the remote repository. If you don't do this, Git will create the old tag on your machine when you pull.

Finally, make sure that the other users remove the deleted tag. Please tell them (co-workers) to run the following command:

git pull --prune --tags

### Removed hidden files from Git (such as .idea)

See <https://stackoverflow.com/questions/32401387/git-add-adding-ignored-files>   
  
Note: I assume that ”.idea” directory is not commited or pushed. However, it is [staged](#_Stage/Indexing).

In cmd/Terminal type:

git rm -r -f --cached .idea

## Advanced Troubleshooting

This subsection some dangerous command that should be used only if you know what are you doing.

### If Pull failed, you should merge you changes first and then

Make some little change to any file, commit your changes (you should have additional commit “Merge branch ‘master’ of”), commit&push your changes and undo you change.

Note: For additional merge problem, See section on [*Merge*](#_Merge).

Note: See also [Troubleshouthing](#_If_Pull_failed,_1) version below for another option. In this option you have auto-generate message for merge commit.

### If you want to move remote branch (for example origin/master), move your local branch to the desired point (Reset current branch to Here…) and then type.

git push --force origin master

Note: This operation should be extremely rare. Do this with care.

Note: Change master to your local current branch name if needed.

Note: If you do this on long lived branch (such as master) you should say to everybody to pull from this repository (to reflect your change in their local repository)

### How to remove 'id\_rsa' or 'id\_dsa' files that was pushed by mistake

Prerequisites

* The [Java Runtime Environment](https://www.java.com/en/download/manual.jsp) (**Java 8** or above)

That's it - the Scala library and all other dependencies are folded into the [downloadable jar](https://rtyley.github.io/bfg-repo-cleaner/#download).

Note: **You should have JRE Installed (java in your system environment path) and you should** [**download jar**](https://rtyley.github.io/bfg-repo-cleaner/#download) **first.**

See <https://rtyley.github.io/bfg-repo-cleaner/> for complete details. You can also strip-blobs-bigger-than 100M. See the link for another example of what files can be removed.

First clone a fresh copy of your repo, using the --mirror flag:

In the cmd/Terminal:

**git clone --mirror git://example.com/some-big-repo.git**

This is a bare repo, which means your normal files won't be visible, but it is a full copy of the Git database of your repository, and at this point you should make a backup of it to ensure you don't lose anything.

**Delete all files named 'id\_rsa' or 'id\_dsa'.**

In the cmd/Terminal:

**java -jar bfg.jar --delete-files id\_{dsa,rsa} my-repo.git**

The BFG will update your commits and all branches and tags so they are clean, but it doesn't physically delete the unwanted stuff. Examine the repo to make sure your history has been updated, and then use the standard [git gc](http://git-scm.com/docs/git-gc) command to strip out the unwanted dirty data, which Git will now recognize as surplus to requirements:

$ cd some-big-repo.git

$ git reflog expire --expire=now --all && git gc --prune=now --aggressive

Finally, once you're happy with the updated state of your repo, push it back up (note that because your clone command used the *--mirror* flag, this push will update ***all*** refs on your remote server):

$ git push

At this point, you're ready for everyone to ditch their old copies of the repo and do fresh clones of the nice, new pristine data. **It's best to delete all old clones, as they'll have dirty history that you *don't* want to risk pushing back into your newly cleaned repo.**

### How to cleanup garbage in remote git repo

See <https://stackoverflow.com/a/37253227>

Note: **the method discards last commit from the repo** - so you probably want to have its backup.

This is how to see unreachable commits locally, where we don't take into account reachability via reflog:

In cmd/Terminal

git fsck --unreachable --no-reflog

Locally, unreachable commits can be cleaned with:

In cmd/Terminal

git reflog expire --expire-unreachable="now" --all

git prune --expire="now" -v

git gc --aggressive --prune="now"

We cannot however run any of these commands remotely.

If you want to clean up unreachable commits in **remote repo**

**First of all commit&push something that you will discard in a few minutes.**

In cmd/Terminal

git reflog expire --expire-unreachable="30m" --all

git prune --expire="30m" -v

git gc --aggressive --prune="30m"

Note: that the time param to expire / expire-unreachable / prune sets the expiration cut-off point measuring from now back. So e.g. "now" means expire / prune everything, and "30m" means except for changes in last 30 minutes.

In cmd/Terminal

git reset --hard HEAD~1

git push -f

**Note:** git reset --hard HEAD~1 **discards last commit (you should create such commit beforehand).**

**Note:** git reset --hard HEAD~1 and git push --f will trigger git gc in remote repository.

### Update Git

See <https://stackoverflow.com/a/48924212>

git update-git-for-windows

### SSL certificate problem: self signed certificate

If you have following error message:

fatal: unable to access 'https://somepath/ProjectName.git/': SSL certificate problem: self signed certificate

You have two options (see <https://stackoverflow.com/a/9008394> ):

1. Open cmd/terminal and type

git config http.sslVerify false

This will disable SSL certificate checking the repository in which you type this command only.

1. Open cmd/terminal and type

git config --global http.sslVerify false

This will disable SSL certificate checking globally.

Note: Seems the --global option IS needed when a repo is NOT yet checked out (can't set options for a repo that doesn't exist yet locally). One can always turn it back on after.

# Appendix C

This appendix contain example on how you can mess-up your code using Git merge.

This is simplified real-world example.

1. Let suppose, we have master that looks like this.



This is exact content of all files:

app.py

**import** sys  
  
**def** main():  
 args = sys.argv[1:]  
 msg = args[0]  
 print(**f"Hello, {msg}!"**)  
  
**if** \_\_name\_\_ == **'**\_\_main\_\_**'**:   
 main()

Basically, this CLI-tool expects to receive the name of the person to greet as parameter, say Alice, and it prints “Hello, Alice!”

1. Now, suppose you’ve started feature branch CoolFeature.   
     
   To make it fun, let suppose you want to check whether the name that was passed matched the username and if it muches you want to add 3 exclamation marks before the name and after the name. The exact details here are not important.  
     
   Before to implement new feature, you decided to make some refactoring first.

You’ve changed app.py and utils.py as following:



app.py

**from** utils **import** calc\_message  
  
**def** main():  
 msg = calc\_message()  
 print(**f"Hello, {msg}!"**)  
  
  
**if** \_\_name\_\_ == **'**\_\_main\_\_**'**:  
 main()

utils.py

**import** sys **as** \_sys  
  
**def** calc\_message():  
 args = \_sys.argv[1:]  
 msg = args[0]  
 **return** msg

Note: You’ve decided to put calc\_message() in *other* file.

These changes are committed&pushed in your feature branch.

1. Meanwhile, bug was reported in production. If no parameter is supplied to your CLI-tool, the program is crashing.  
     
   In order to fix the bug, somebody else in your team are starting HotFixBranch.   
     
   He/She ended up only with 1 file changed.

app.py

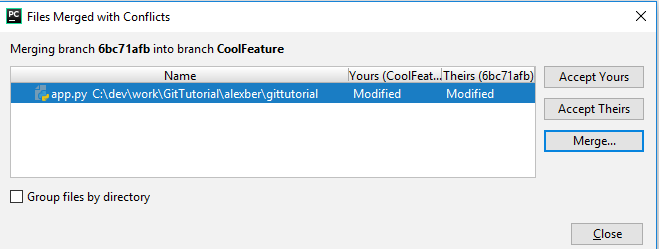
**import** sys **as** \_sys  
  
**def** \_calc\_message():  
 args = \_sys.argv[1:]  
 msg = args[0] **if** args **else "World"  
 return** msg  
  
**def** main():  
 msg = \_calc\_message()  
 print(**f"Hello, {msg}!"**)

**if** \_\_name\_\_ == **'**\_\_main\_\_**'**:  
 main()

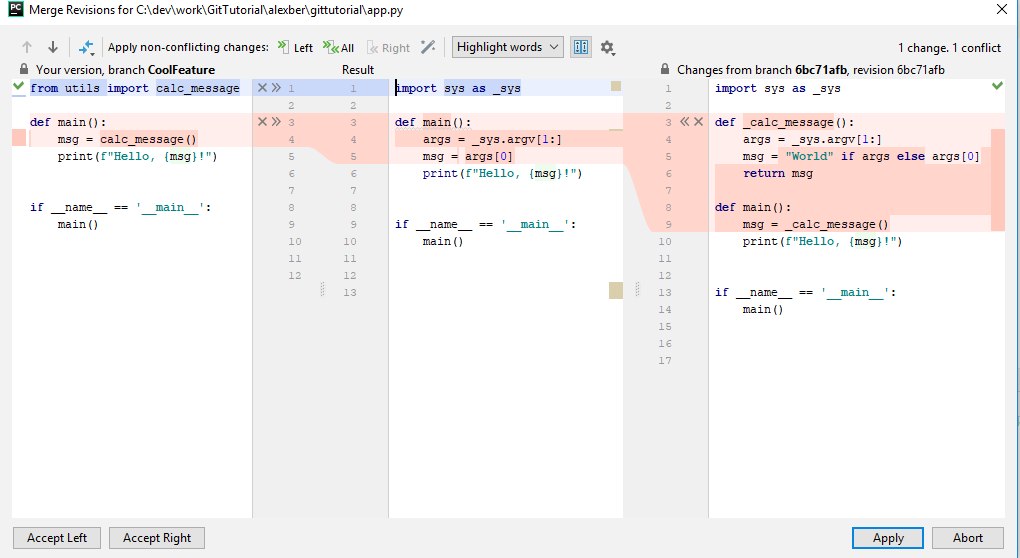
After finishing all test, he/she commit&push the change to HotFixBranch.

He/she merges this change to the master.

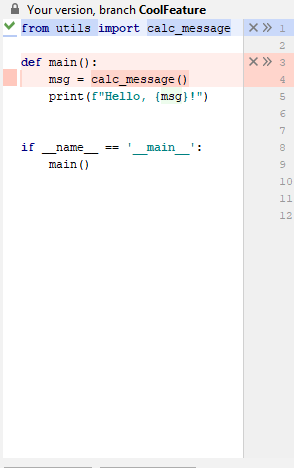
1. You receive the following merge window:



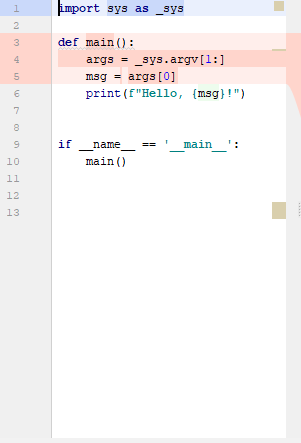
Now, you click on Merge window and you see the following:



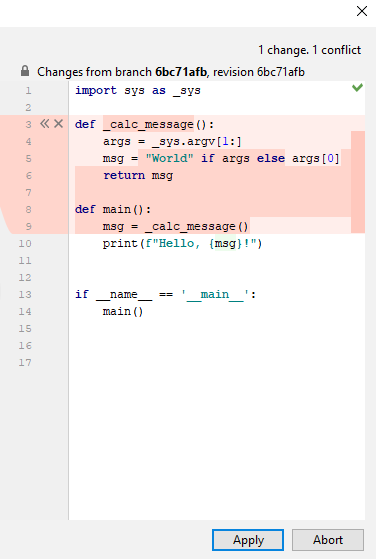
Let’s zoom in to the left panel:



Middle panel:



Right panel:



**And now, you have a problem.**

The problem here, that **you’ll end up with code duplication**. You will have calc\_message() function in both app.py and utils.py. Moreover, the code is different utils.py will still contain the bug.

The real problem, that **you should remember that you have some related code to this merge**.

If you do remember this, you have couple of different way to handle this:

* Make correct merge for this file. Make separate change in app.py and amend merge-commits.
* Overwrite your changes. Look on diff and reapply your code on top of it making manual merge.
* Make cherry-picking.

And again, the main problem that usually, you’re not aware that you have a problem in the first place.

# Appendix D

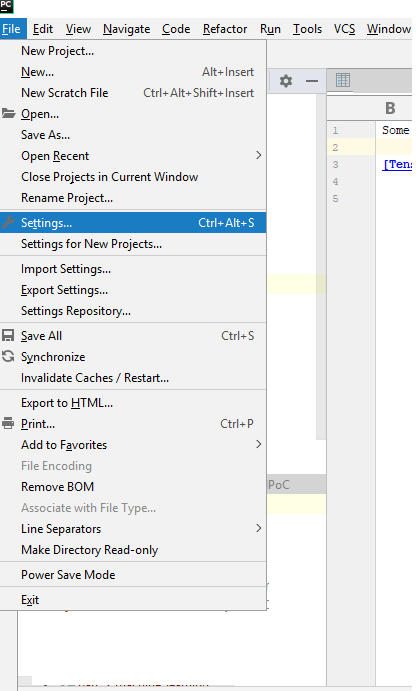
This appendix contains step to reproduce my local environment.

1. Install git client.

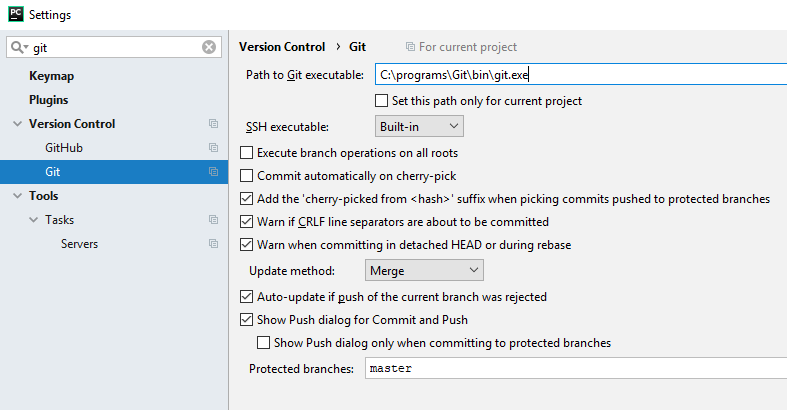
* Go to <https://git-scm.com/downloads> and download appropriate one. I use 2.22.0.windows.1 (latest for the day this document was written).
* Install it.
* You can see Appendix B->Advance Troubleshooting->Update git section if you need to update it.
* Add path to git.exe to your Path environment variable (or it’s equivalent on another OS).

1. PyCharm 2018.2.4 (Communnity edition) is used as Git UI in this document.

* See <https://www.jetbrains.com/pycharm/download/previous.html>
* In Pycharn open Settings. File->Settings



Ensure that you have following settings:



Note:

* Path to Git executable may differ on your machine.
* Ensure that “Update method“ is Merge
* Ensure that check-box “Warn when committing in detached HEAD or during rebase is checked.
* (you can also do git init, and git remote add origin, create local master, and git push -u origin master)