# Table of contents

Contents

[Table of contents 1](#_Toc14588131)

[Introduction 3](#_Toc14588132)

[Basic operations 5](#_Toc14588133)

[Add File 5](#_Toc14588134)

[Commit File 8](#_Toc14588135)

[Push (no conflicts) 11](#_Toc14588136)

[Pull (no conflicts) 16](#_Toc14588137)

[Merge 20](#_Toc14588138)

[*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.* 28](#_Toc14588139)

[*What if I’ve messed file with the files that are not present in the list?* 30](#_Toc14588140)

[*What if I want to delete my last commit?* 34](#_Toc14588141)

[*If you have conflict when you’re* ***pulling*** *from git, just resolve the conflict as described.* 35](#_Toc14588142)

[*If you have conflict when you’re* ***pushing*** *to git* 36](#_Toc14588143)

[*All conflicts fixed but you are still merging.* 40](#_Toc14588144)

[Theory 42](#_Toc14588145)

[Distributed Version Control 42](#_Toc14588146)

[Optimistic locking 44](#_Toc14588147)

[Branches 46](#_Toc14588148)

[Stash 48](#_Toc14588149)

[What if you’re in the middle of the work and you want to make “git pull”? 48](#_Toc14588150)

[What if you have complicated merge and you prefer to make it in more controlled way? 57](#_Toc14588151)

[Create New Project 57](#_Toc14588152)

[From Scratch (git init) 57](#_Toc14588153)

[From existing project (git clone) 57](#_Toc14588154)

[Why not master only? 57](#_Toc14588155)

[Feature branch 57](#_Toc14588156)

[Appendix A 58](#_Toc14588157)

[Appendix B 59](#_Toc14588158)

[Troubleshooting 59](#_Toc14588159)

[If you close merge resolution windows and when you want to commit your merge changes, you don’t see any file that change 59](#_Toc14588160)

[If you successfully resolve all merge conflicts, but you receive error message “You have not concluded your merge (MERGE\_HEAD exists)” 59](#_Toc14588161)

[If Pull failed, you should merge your changes first and then 60](#_Toc14588162)

[**Git Delete Last Commit** 60](#_Toc14588163)

[Advanced Troubleshooting 62](#_Toc14588164)

[If Pull failed, you should merge you changes first and then 62](#_Toc14588165)

[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. 62](#_Toc14588166)

[Appendix C 64](#_Toc14588167)

# Introduction

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

His 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.

There some choices and assumptions that was made deliberately and they not discussed in the document. 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 C](#_Appendix_C) 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)).

Then there is pretty elaborate section on [Merge](#_Merge) It described many day-to-day use-cases and pitfalls. It is essential to understand how you can resolve conflicts.

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.

Next section is about [Stash](#_Appendix_A).You should use it occasionally.

Next section describe 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).

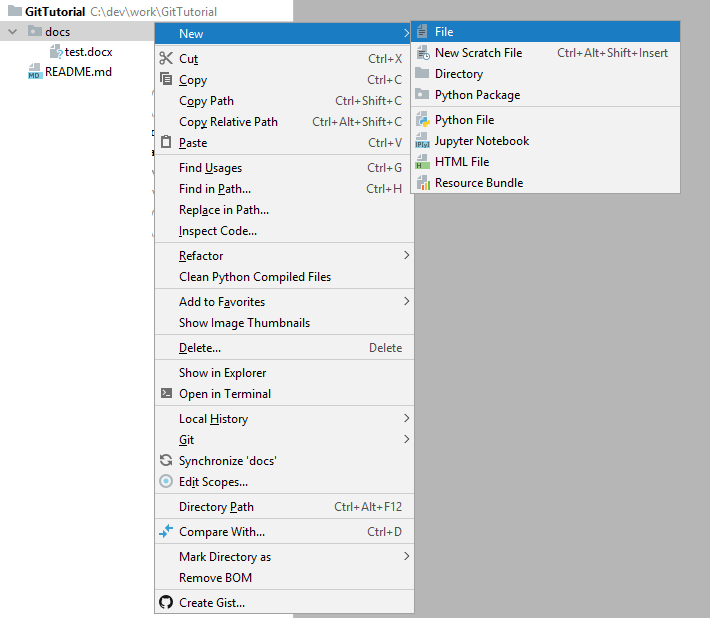
[Why not master only?](#_Why_not_master) section start the discussion of the best practices of working with Git.

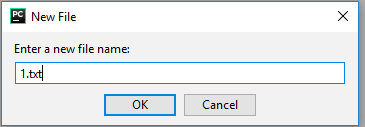
[Feature branch](#_Feature_branch) section describes one of the best practice that I recommend to do.

# 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).





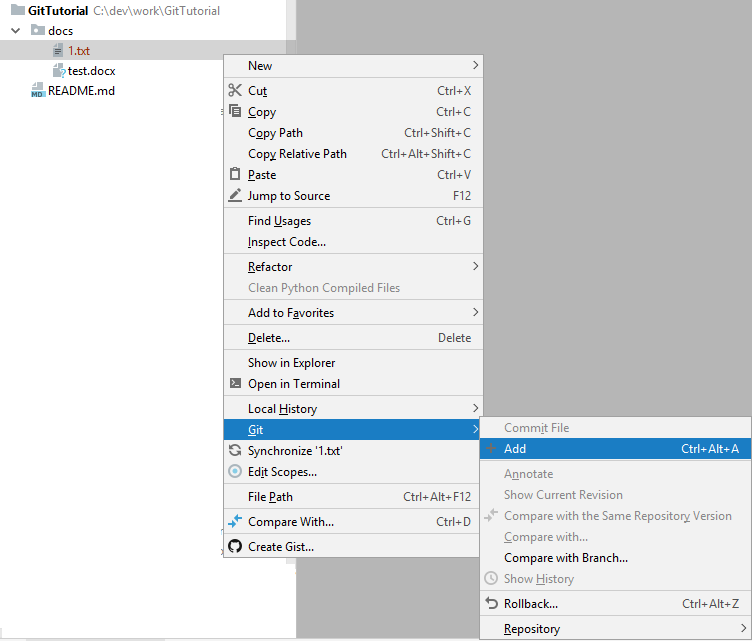
In such a case you just do nothing.

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 (you better put some text there).
* Another use-case, when PyCharm is not added this file. See in commit section, how do you know.

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) automatically.

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

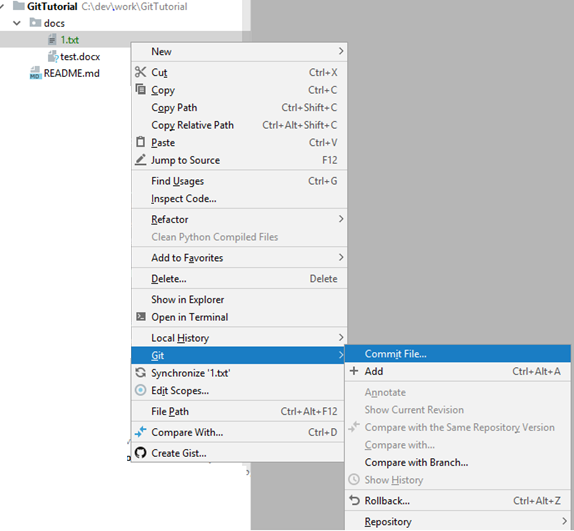


Note: PyCharm mark modified file by blue color.



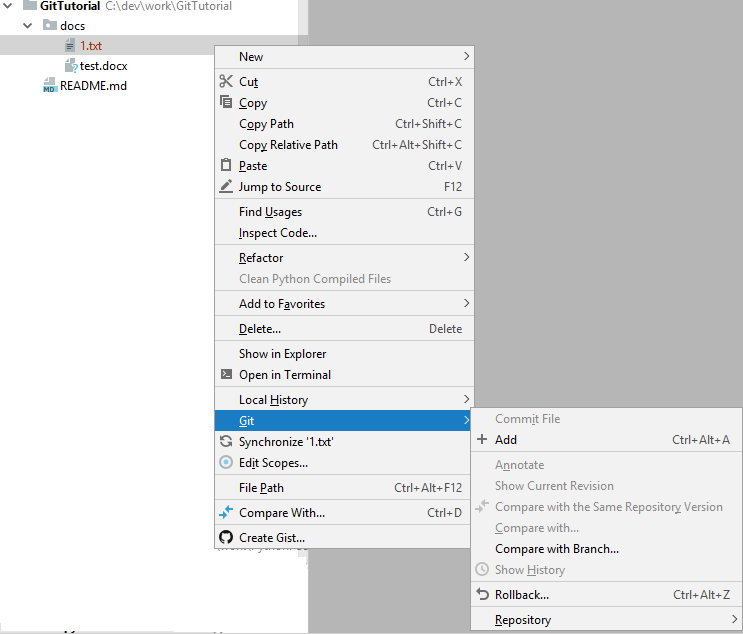
In order to commit file, you can

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



* How can you check that file you are about to commit was added (to index)? 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.

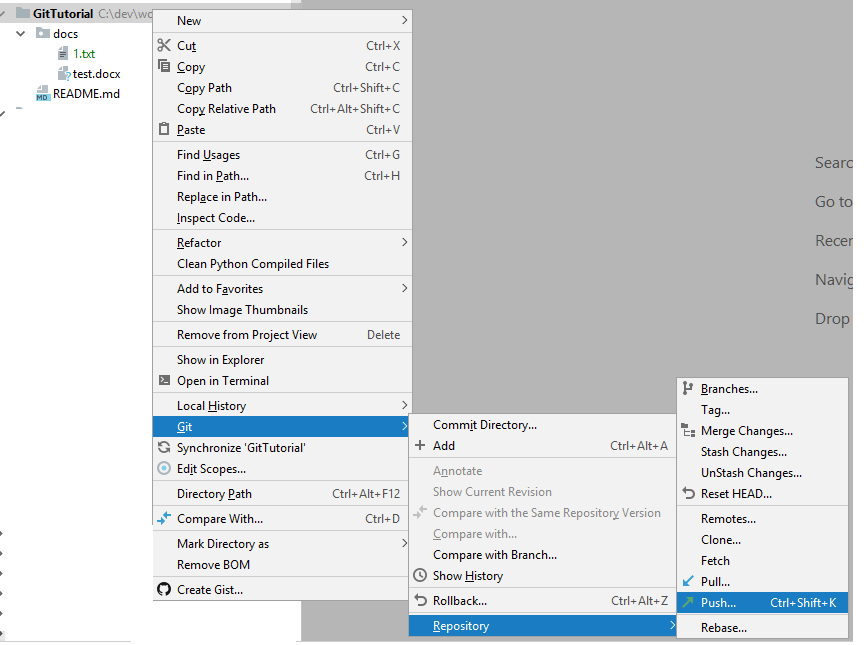
Note: Typically, you should do “git pull” first.

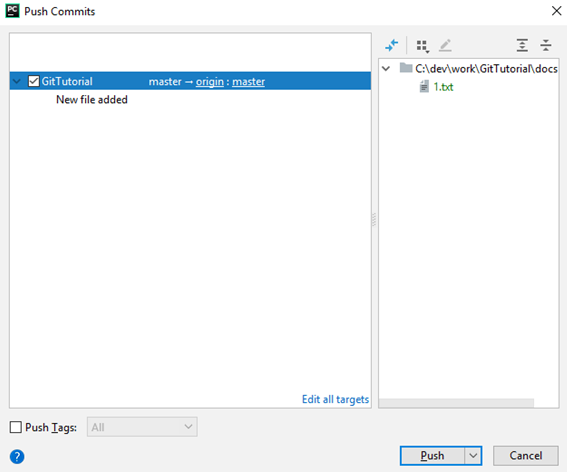
Note: Anyway, in this section I’m assuming that there are no conflicts in “git push”. Conflict resolution is walked through in “[Merge” section](#_Merge). See subsection “[If you have conflict when you’re pushing to git”](#_If_you_have).

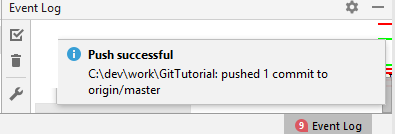
Right-click project -> Git -> Repository ->Push

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”.





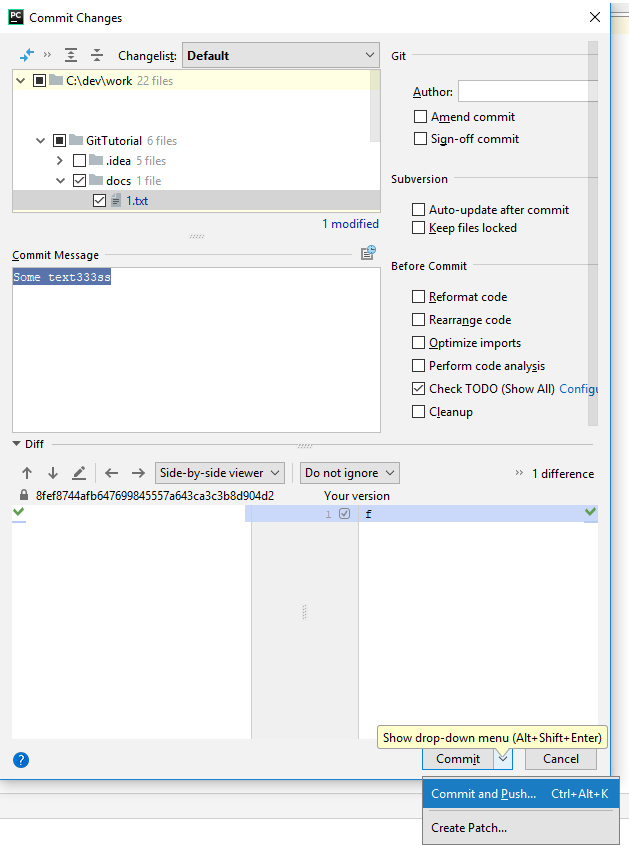


In the push commits window, you see all the Git project. In every Git project you see message of every commit that you’re about to push. On the right you can see from what files each commits consists of.

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 “v” and you will see sub-menu in which you can chose commiy&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)

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

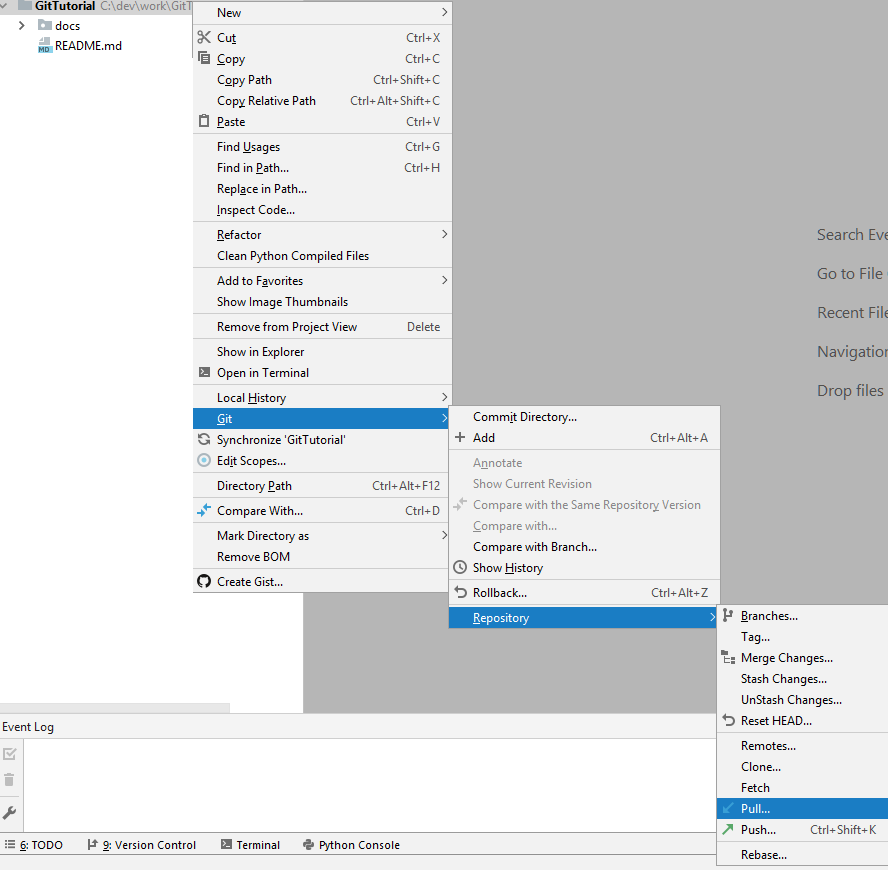
Important: It is better that **you will not have any uncommitted files before** you do “git pull”.

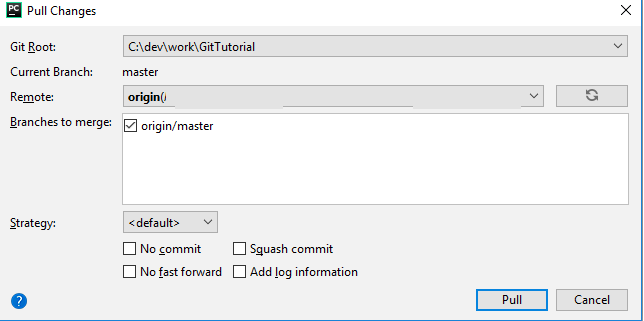
Note: If you make some changes, but didn’t finish you work and you want to pull the changes, you can stash (see [section on Stash](#_Stash)) your changes first, then make pull (resolve conflicts with your previous commits, if any), unstash (it will apply you changes on updated code-bases; you may require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

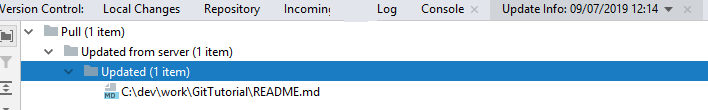
Righ-click on the project->Git>Reposiotry>Pull…

In the popup window click on “Pull” button.

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 resolve such a conflict. This use-case will be discussed below.

You can need to make merge in these cases:

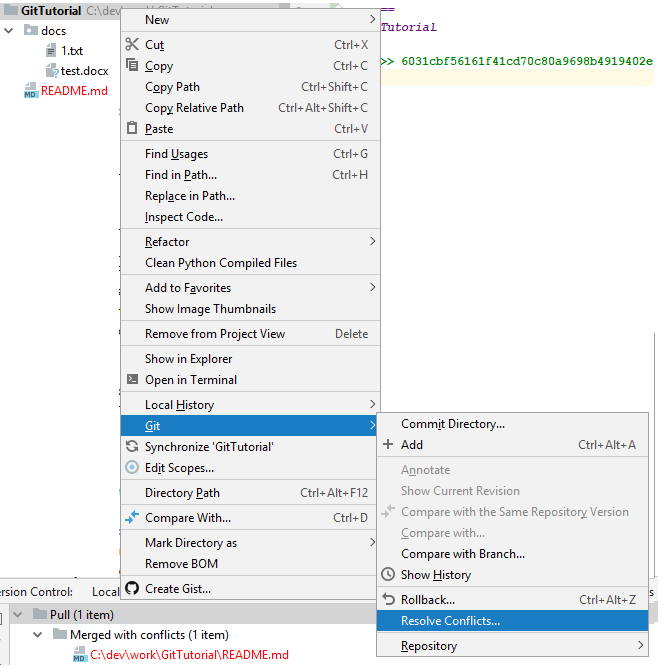
* On “git pull” – this is typical case.
* On “git push”. This can happen, but should be rare.
* When you specifically merge different branches (this will be discussed in separate section).

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

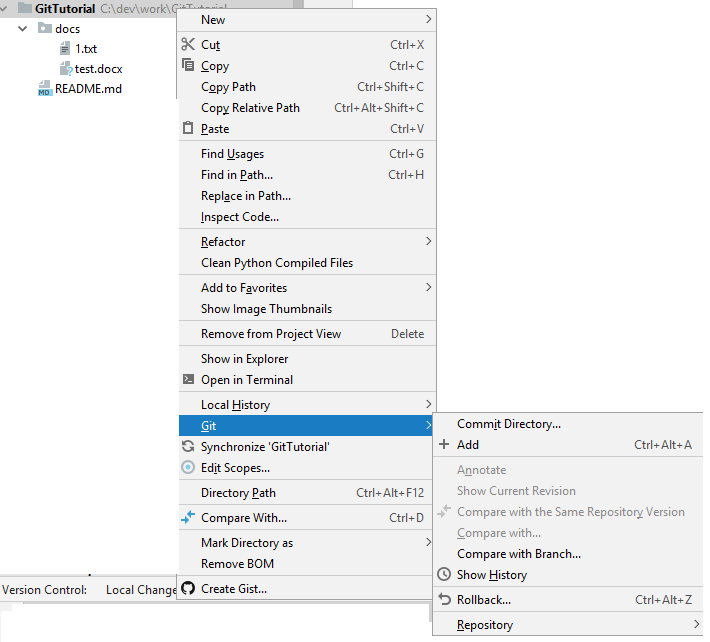
* If you add file to git, no conflict can occur on this operation.
* If you commit file, then on commit operation, no conflict can occur.

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

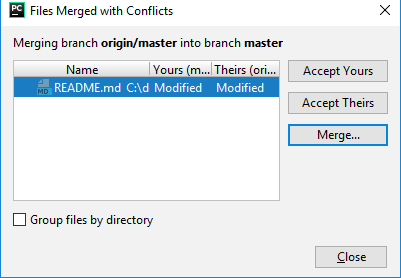
If not, right-click on the project->Git->Resolve conflicts…



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 here. It will be discussed later.

Note: Don’t do it blindly. You’re going to discard some other changes.

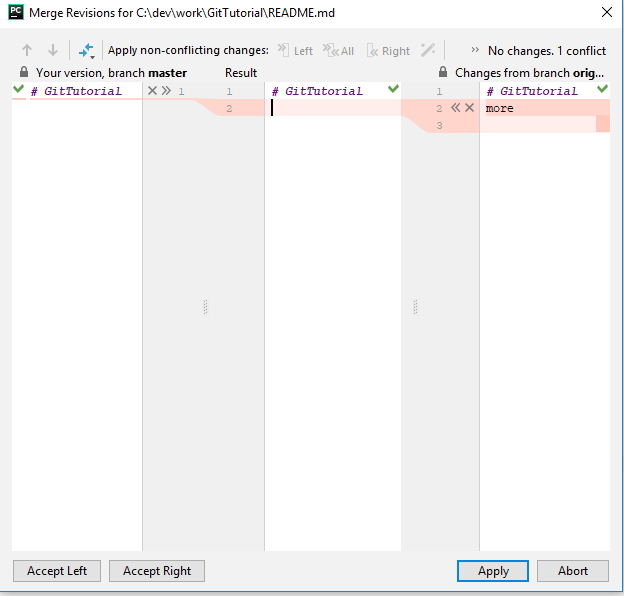
* click on "Accept Theirs” button – this will discard all your changes, and you will receive all “theirs” changes.

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:



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 adviced.
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 here. It will be discussed later.

Note: Don’t do it blindly. You’re going to 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).

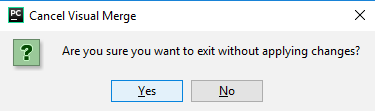
* “Abort” button will abort merging, See details below.
* “Apply” button will save you changes and close the merge window.

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

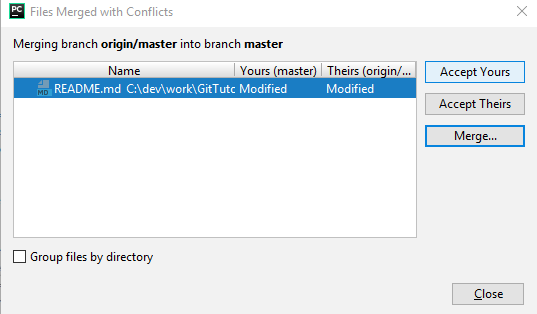
Note: If for all files in the merge you click on "Accept Yours” button, you will have no files to commit. So, PyCharm will not let you commit your merge. This is caveat from above. I will discuss it more below.

## *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, 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: This is equivalent to writing the following command in CMD/Terminal

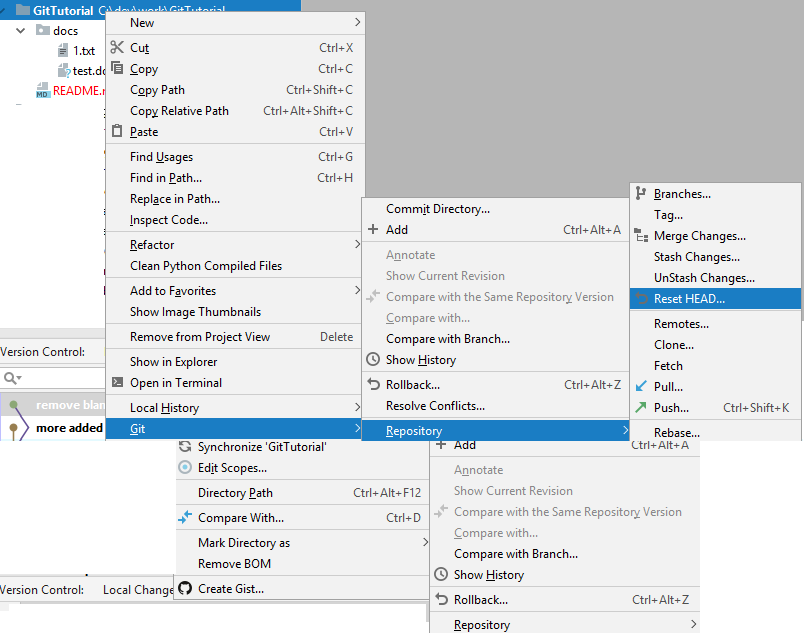
git merge --abort

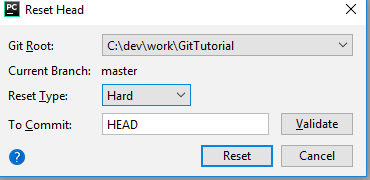
## *What if I’ve messed file with the files that are not present in the list?*

In such case you should “RESET head”.

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

“Reset Head” window should be opened.

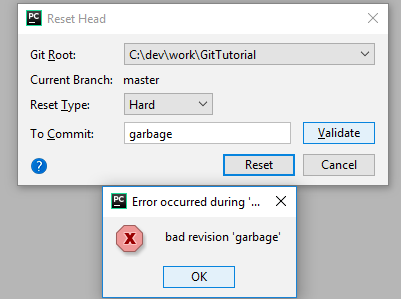




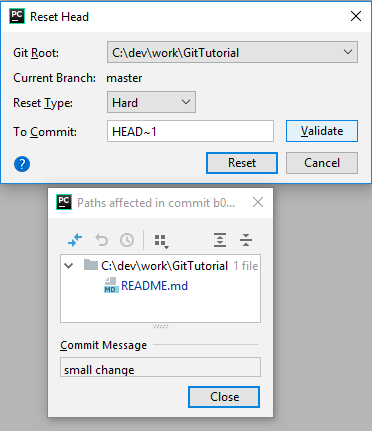
* Reset Type “Hard” means that any change that was made locally will be lost. This is typical case.
* Reset Type “Soft“ means that no changes to files will be done. If you want to “undo” merge operation, this is not very helpful.
* Reset Type “Mixed” means that any change (with exception of files that 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 vaild 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).

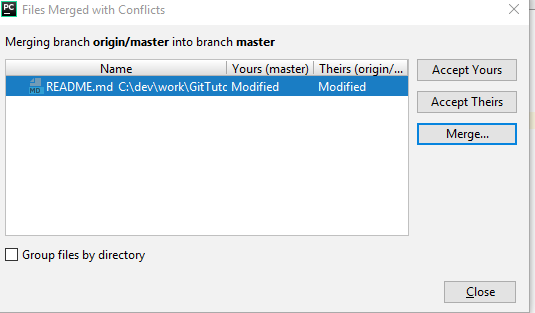
## *What if I want to delete my last commit?*

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”.

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



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

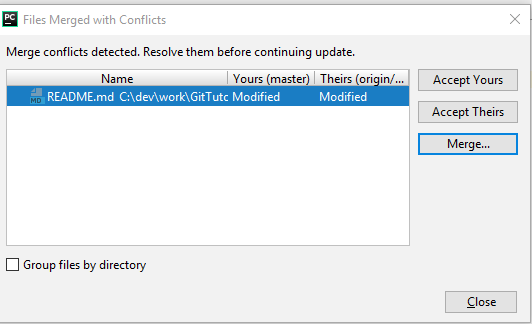
Note: See also [*Troubleshooting*](#_If_Pull_failed,_1) section.

## *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”.

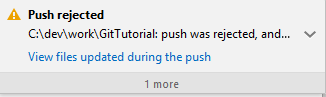
Note: Typically, you should do “git pull” first.

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.

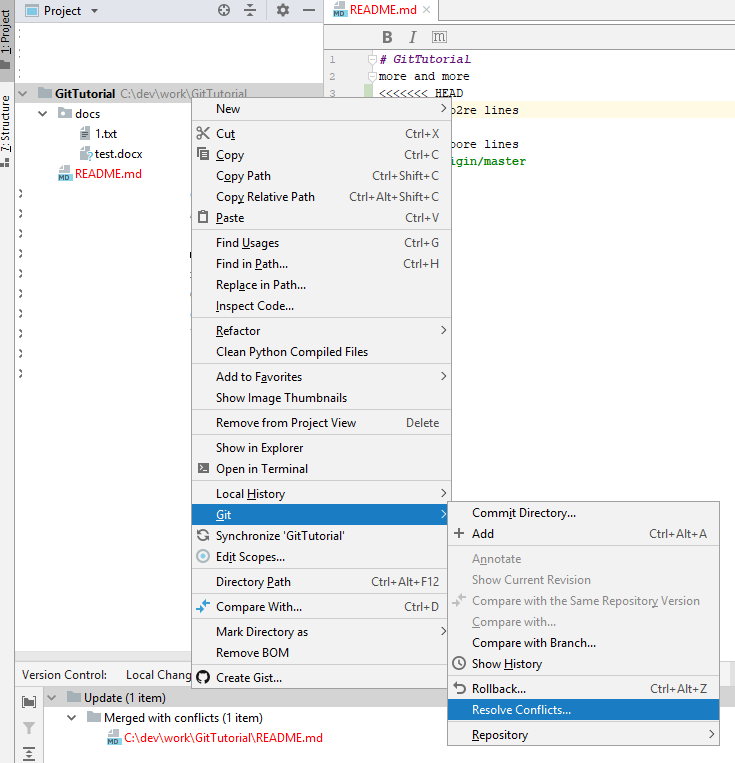


Now, PyCharm pulled the changes to our workspace.

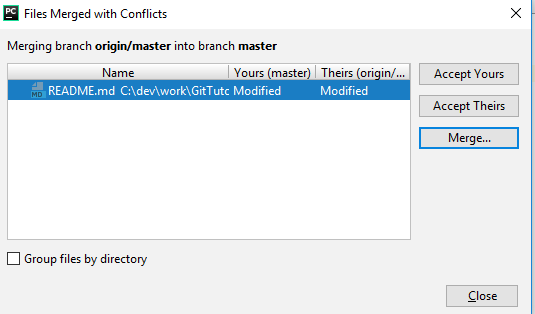
We should manully open the window and start merging process.

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.*

How do you do that merge wasn’t concluding?

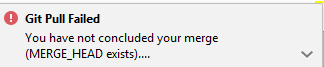
In theory, you can open cmd/terminal and type

git status

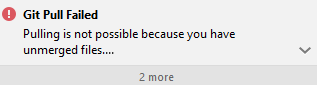
If you see something like



In practice, it would be when some action that you want to do with git will fail. For example, when you make “git pull” 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.

# 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 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.

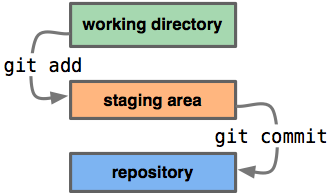
**Git uses optimistic locking**. Git assumes that the conflict should be rare, so while the cost of each single merge is high, the overall cost of conflict resolution is low. In practice, you should make active steps to reduce the probability of conflicts, they will be discussed in the chapters [Why 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.
2. 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.
3. 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. With its parental commits, the full branch structure can be constructed. It takes only few seconds to create and delete 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. 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.

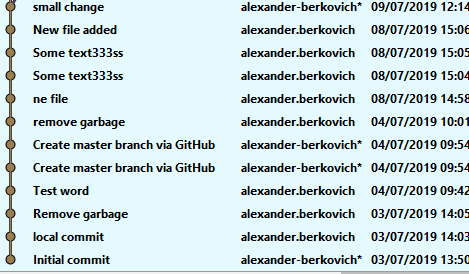
**origin**: the default name that git gives to your main remote repo. You have local repo, and you most likely push out to some remote repo that you and all your coworkers push to. That remote repo is almost always called origin, but it doesn't have to be.

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.

### Commit History

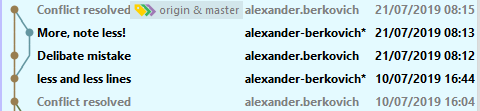
This is example of the commit history view:



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

Note: Typically, **commits are immutable**. **You can’t modify commit’s 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.

* 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.

# 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:

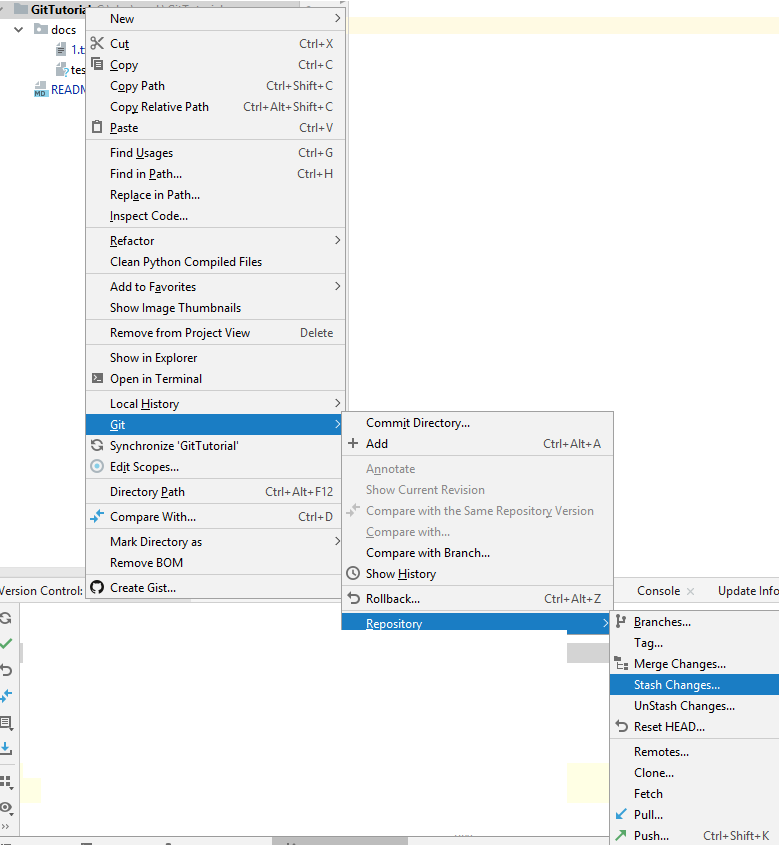
1. You will commit with incomplete work (the application may even to compile)
2. Resolving merge conflict may be difficult (For example, if you modify 10 files and you are in the middle of refactoring).

* Another alternative is to manually copy your changes file aside, revert you changes in the PyCharm project, make “git pull”, 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”.

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.**

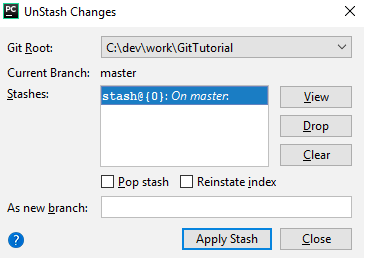
*Now you HEAD and master points to the same commit. You working tree is also as HEAD. If you*

1. **Now, make “git pull” 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.

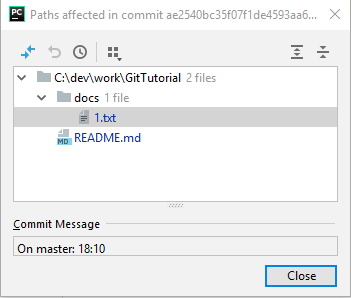
****

You should see the following window:

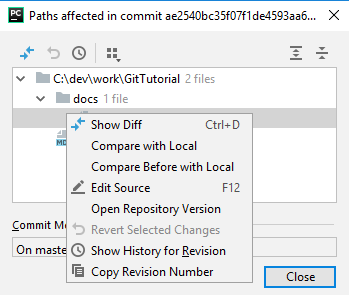


Let’s go through your options.

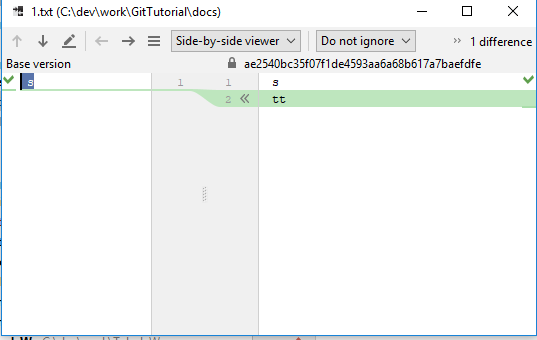
* “View” button will open new window where you can examine what files where stashed and to see the content of the files and it’s diff.



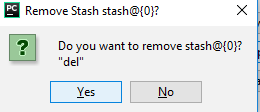
Right-click on the file, you will see the following menu:



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:

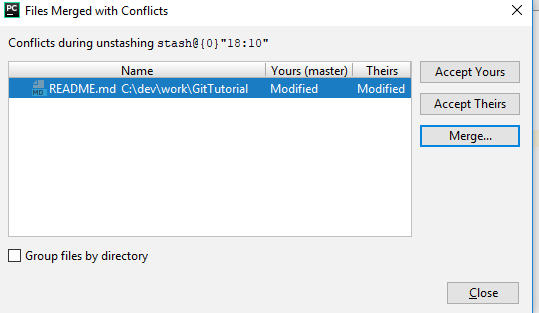


Click yes, **to remove all stashes.**

* 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:**

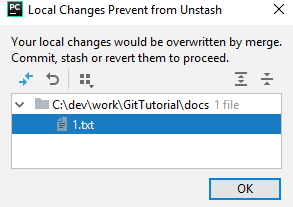


Make [usual merge process](#_If_you_have_1) for conflict resolution.

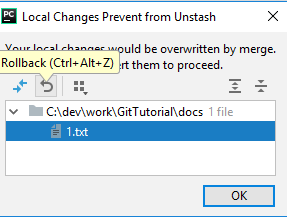
1. **You have some uncommitted change and you have conflict with one these files with unstashed filles.** Than you will see the following popup window on the right corner:

****

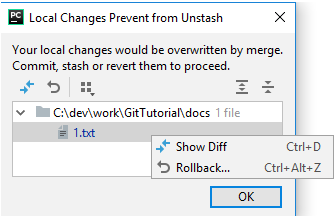
Clicking on the “View the” link will open the following window:



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

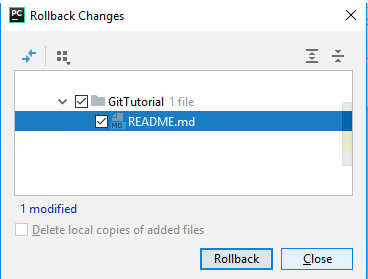


* Or **you can right click on the file**



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?

# Create New Project

## From Scratch (git init)

## From existing project (git clone)

# 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.

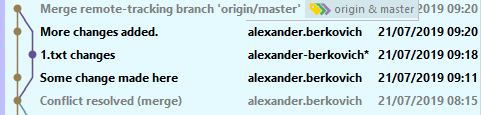
1. I committed some change on README.MD file.



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 Im commit&push the changes.

The commit history looks like this



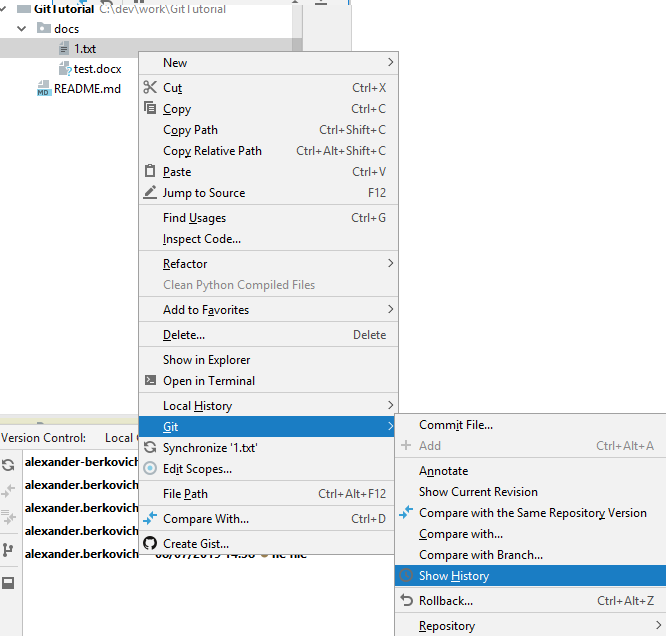
* My first commit was “Some change made here”. Its parent is “Conflict resolved (merge).
* My second commit was “More changes added”. Its parent is “Some change made here”
* But **what is “Merge remote-tracking branch ‘origin/master” commit (greyed-out)?** I wasn’t asked to do any merge (because there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

1. Git created *merge commit* automatically.
2. Our workspace contains something we’re not aware for.
3. What if we do have merge conflict?

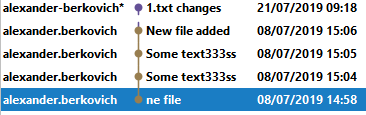
So, let’s examine all issues one-by-one.

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 original one and merge commits).
2. I want to re-iterate, you did commit&push, but Git on behave of you did **“git pull”.**In order to prove it will examine the history of 1.txt file.

Rig



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” 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
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
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 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.

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. Potentially, it will contain some code refactoring ??????????, you will see this as soon as possible and if you are doing the change in the same files, you will have 1 complex merge with your little changes and not when you will have many changes in the same files at the end.

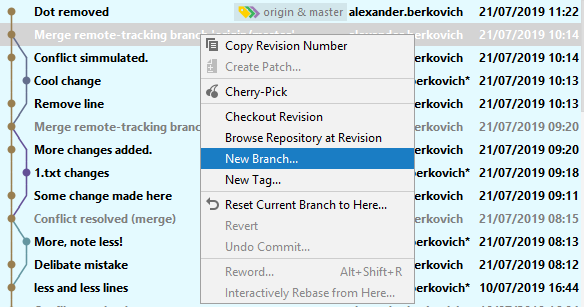
# Feature branch

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.

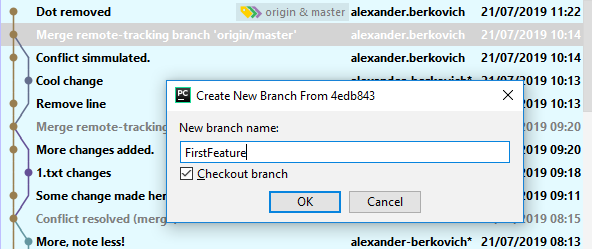
## New Branch

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…

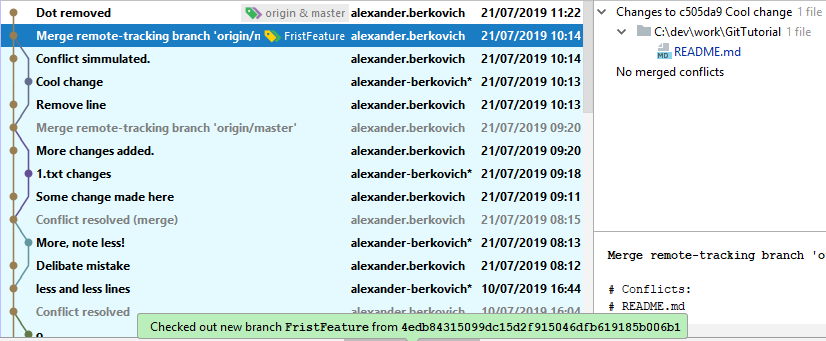


In the popup window write down name of the new branch.



Note: Checkbox “checkout branch” means whether you want to switch your workspace to this branch. Usually, you do want it.

When you will click ok, you will see:

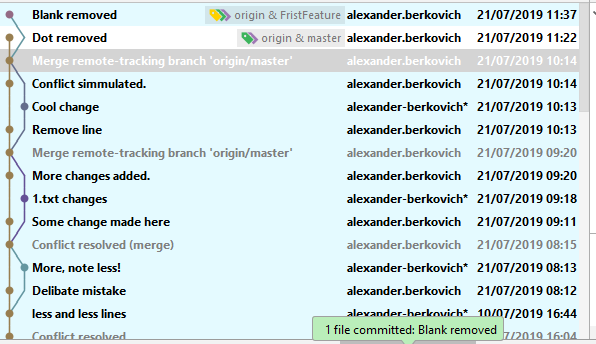


Note: If you use checkbox you will see near left corner of the PyCharm the message “Checked out new branch *BranchName* from *id*”.

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.

# 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>

### 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**

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.

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

## 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.

* 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)

* Update git. See <https://stackoverflow.com/a/48924212>

git update-git-for-windows

* 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

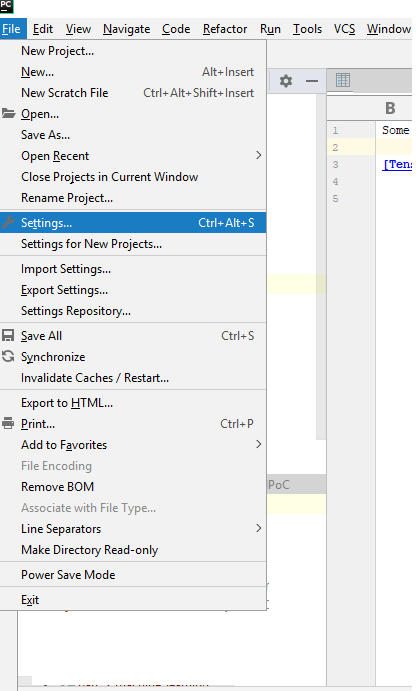
This appendix contains step to reproduce my local environment.

1. Install git client.

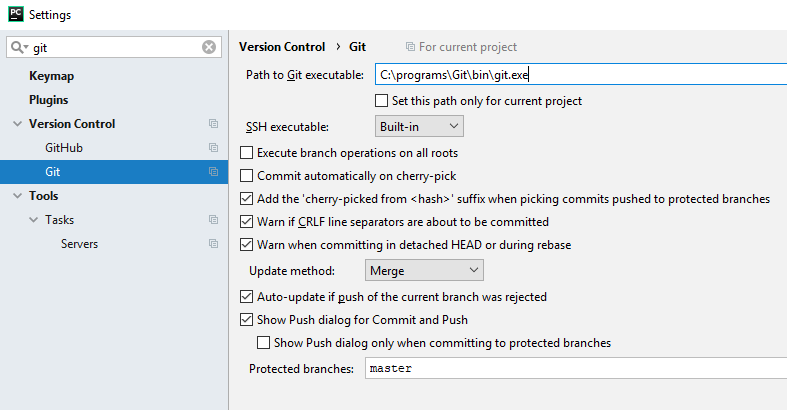
* Go to <https://git-scm.com/downloads> and download appropriate one. I use 2.22.0.windows.1 (latest for the day this document was written).
* Install it.
* You can see Appendix B->Advance Troubleshooting->Update git section if you need to update it.
* Add path to git.exe to your Path environment variable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