GIT guidelines and policies

Git is a useful tool for managing the code with many developers. It allows people to work simultaneously on the same files by providing tools to merge their work (or indicate where there are conflicts).

The workflow is centered about creating commits - snapshots of the changes to the files (usually - source code). Each commit has to have a certain purpose reflected in the commit message (eg. “Add PID controller class”). The messages allow for identification of the commits, so a clear message should be the goal we strive for. Imagine yourself looking at what other people did - each change can be relatively complex, so it is important to structure the commits so that each commit accomplishes certain goal. State *what* it does, rather than *how it achieves it*. There are many guidelines on how to write “good” commit messages, so you can look them up online (e.g. [here, a guide on github.com](https://gist.github.com/robertpainsi/b632364184e70900af4ab688decf6f53)). While we won’t enforce any standard, it is good to keep these issues in mind as it: a) makes our lives easier later on; b) makes you more qualified for a job later (where your manager will grind you if you don’t stick to the guidelines).  
  
There is a good practice of structuring your work beforehand into committable chunks and following a rule: write, test, commit. So e.g. writing a function that does something specific could well be a valid commit.

How often you want to commit is up to you - use the chance to experiment a bit with it. The frequency of the commits does not impact other members of the team as much.

The more important issue is the one of branches. What branches allow you to do is to take your code and play with it in your own environment, creating your own version of the code.

Our main branch is master. This should have **working and tested** code only. It is so that if you want to add something, you can test against a last version to make sure that you don’t break anything else by accident. I.e. e.g. you take version from master, see what it does, go to your branch and compare the two behaviours. There should be some improvement, or at least no degeneration.

Remember that the master branch reflects a *consensus* about how we want things to look and work. I.e. you are absolutely free to experiment with whatever features you do, but not on the master branch*.* The reason for it is quite simple - there is no inherent difference between branches. All of them can be used, all of them might have the same files. But it will be impossible for us to know what branches are important, what are not, what experiments will turn out to work, what will fail.  
So we want to have some place to keep stuff that we agree on and we know works. Yeah - the master branch.

That’s why there should be **no commits to the master branch**. But what if the commit is a simple bug fix? Then it means it shouldn’t have been merged with the master branch in the first place. All the simple bugfixes should be caught on the other branch. Bigger bugfixes should be branched off first - after all the approach to fixing the bug might not work and we don’t want to roll-back commits, because it’s pain. It’s easier to just scrap a branch.

So how do you change the master? By using pull requests on the github page. Agreed, pull requests are pain - they are not as quick. But hey - why do you insist on having it on the master branch anyway? It is just a name, isn’t it? Yes, master is a branch like any other, it is just very, very convenient to have the *highest quality, working code we collectively have agreed on* in the master branch. This implies that whatever changes are made to the master branch are very likely to stick forever and your code should be compatible with them. (BTW, this is done with *$ git merge master* when you are working on your branch. NB you might need to *$ git checkout master* and *$ git pull origin master* first.)

So you make a pull request and what should happen next is that you have another person accept it. Ask them to have a look over it. We likely won’t do a full-fledged code review (although it might be a nice exercise) but it is a very simple sanity check. Grab someone and walk them through the code (github should highlight all the changes that you made). Don’t accept your own pull requests - it defeats the purpose of having something that the team agreed on. Remember - it does not matter if your code is on master one day later.

It is also a good idea for everyone to update their own branches after a pull request is accepted on the master. This saves you work later on, when you prepare your branch to get pulled to master. Also, you make sure that your features don’t break the cutting edge code that was just rolled out.

The burning issue is - when to branch. That is a good question that probably has many good answers. It largely depends on your taste, but as a general rule - each more complex feature or a task that you take (e.g. fix obstacle avoidance bug) might be enough to make a branch. When you think you might want to revert to a current state but you want to try out some things - make a branch. Usually it is tempting to comment-out big chunks of code, but as you make more and more changes it is hard to track what actually made the difference and what was the state when you first began (especially if your experiments span over more than one coding session).

Also - remember to keep the branch names relatively short but informative. E.g. using your name is not a great idea, because it doesn’t indicate what is inside, what changes you make. Before branching off make sure that there are no similar branches already as having names which look similar may cause unnecessary issues - “bug\_fix, bug\_fix\_2, simple\_bug\_fix” are not the best names - how do you differentiate between the three of them?

After the pull request is accepted and you don’t plan to develop on this branch further - consider closing it so that it doesn’t clutter the repository (you can do it from the github webpage).

$ git clone <https://github.com/MahbubIftekhar/RoboTour>

$ git checkout master

$ git checkout other\_branch

$ git checkout HEAD

$ git checkout -b new\_branch

$ git push --set-upstream origin new\_branch

$ git pull origin branch

$ git log

$ git status

$ git add files\_to\_add\_to\_commit

$ git commit -m “Add message to the commit”

$ git merge branch\_to\_merge\_from

$ git commit --amend -m “This will modify the previous commit”

$ git help some\_command #e.g. $ git help checkout

$ git diff other\_branch file

[Pretty cool tutorial with cheatsheets and stuff from Atlassian](https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud)