workflow-and-style

SQC, SCN

27 Januar 2016

Contents

---

1 IMS R style guide

1.1 External Sources

- Hadley style guide
- Google style guide

If hadley and google contradict each other:

1. prio: hadley
2. prio: google

1.2 Naming Conventions

Folders: `meaningful name` or `meaningfulname` Files end with '.R': `meaningful-file.R`
Objects (Variables and Classes): `meaningful_variable` Functions: `meaningful_function` Packages: `mngflabrr` Trade of between: googleable, remindable, selfexplainable, short etc. link

1.3 Versioning

Adapted from http://semver.org/ and http://yihui.name/en/2013/06/r-package-versioning/.

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,

- MINOR version when you add functionality in a backwards-compatible manner that is worth writing a few lines (in the description file) and creating a GitHub Tag, and
- PATCH version when you make a commit to your development branch. (bugfix, feature or minor change)

1.4 Indentation

When indenting your code, use two spaces. Never use tabs or mix tabs and spaces. Spacing Place spaces around all binary operators (=, +, -, <-, etc.). Do not place a space before a comma, but always place one after a comma. Place a space before left parenthesis, except in a function call.

1.5 General Layout and Ordering

1. Copyright statement comment
2. Author comment
3. File description comment, including purpose of program, inputs, and outputs
4. source() and library() statements
5. Function definitions
6. Executed statements, if applicable (e.g., print, plot)

Unit tests should go in a separate file named `test_originalfunction(s).R`.

Install packages with `install.packages("package")`

Include non-standard packages using `libraray(package)`

1.6 Documentation

1. R packages are documented with vignette(s). For examples see here
2. Every project has a starUML document describing the software architecture of the project according to the 4+1 view model.

1.7 Backlog and User Stories

1. Work-In-Progress is documented on the Symphonical wall of the project in form of a User Story.

1.8 Documentation and project management

Use the following structure for packages

- /package/data/: RData (Output).
- /package/inst/extdata/: Data which have to be accessible by the user of the package.
- /package/tests/testthat/: All relevant rawdata for tests - keep the size small! source

For customer projekts we have

- /project/data/rawdata/yyyy/: subfolder for rawdata for year yyyy.
- /project/data/yyyy/: working directory includes all analysis for year yyyy.

1.8.1 Count Story Points - Script

Export Symphonical as JSON file. run the following script:

```
imsbasics::clc()
symphonical <- rjson::fromJSON(file = "swBn2e.json")
cards <- symphonical$board_cards
description <- c()
aufwand <- c()
for (i in 1:length(cards)) {
  description[i] <- cards[[i]]$description
  aufwand[i] <- strsplit(description[i], split = "Aufwand: ")[[1]][2]
  aufwand[i] <- strsplit(aufwand[i], split = " SP")[[1]][1]
  }
aufwand <- as.numeric(aufwand)
sum(aufwand, na.rm = TRUE)
```

---

2 R, Rtools and Rstudio

2.1 Install R and Rtools

For windows: Install latest R version from https://cran.r-project.org/bin/windows/base/ and install it to C:/R/R-x.x.x. Do not install it to the program folder because you will run into trouble with admin right.

Next install latest Rtools version to C:/Rtools. The file (It is not a package) can be downloaded from here: >https://cran.r-project.org/bin/windows/Rtools/> Be careful with path conventions. R needs "/"!

```
Be careful: Here, you have to add the system path "C:/R/R-x.x.x/bin;"
```

To reach the path see `Systemsteuerung\System und Sicherheit\System` => Umgebungsvariablen...

For further help see http://socserv.mcmaster.ca/jfox/Courses/R/ICPSR/R-install-instructions.html

2.2 Install RStudio

Install Editor from here Most likely you have to install further packages. Either go to Rstudio and use the console

```
> install.packages(c("devtools", "testthat", "ggplot2", "knitr", "lme4", "MBESS",
  "mi", "polycor", "readxl", "rgl", "rmarkdown", "sem", "sfsmisc"))
```

or use the built in package Editor => Install

```
Important note: Install packages always to C:/R/R-x.x.x/
```

It might also be useful to install MikTex to be able to Rmarkdown.

Finally, make sure, you installed the packages

- devtools used for downloading and installing code from github and
- testthat

When you want to automate testing open or create a project (including package) and type the following command in the R console:

```
> devtools::use_testthat()
```

This creates the test_that folders. When you change the project in RStudio, you start with a **clear environment** and a **clear shell**. For Details see Hadley "R packages", Ch. 7.

2.3 Install a package with auth_token

In RStudio use the following command in the R console

```
devtools::install_github("ims-fhs/repo", auth_token = "35b77ed8d745a9572409e6c153e20e36d45094ed")
```

3 Git and GitHub

3.1 External Sources and quick tour for impatient guys

- Hadley: How to set up Rstudio, git/SVN and github http://r-pkgs.had.co.nz/git.html
- Github help on git's clone, fetch, pull, merge https://help.github.com/articles/fetching-a-remote/
- Excellent git tutorial. **Read at least the the first three chapters**.

Use the following commands to track and understand what is going on. For explanations see below. However, these commands are save and you can't screw anything up!

```
> git status
> git branch -av
> git remote show origin
```

3.2 Existing Github accounts

- symbolrush (User)
- Christoph999 (User)
- ims-fhs (Organisation with rights to set folders as private)

Furthermore, we assume for the following that we have an existing github repository "https://github.com/ims-fhs/myfolder".

Note about fork and branch: When talking about version control, we have to distinguish "fork a branch" and "Create a new branch": In contrast to "branch", "fork" is a pure github to github operation where everything gets copied to another user.

3.3 Shell: First steps

A (Linux) shell can be opened in Rstudio via Rstudio => Tools => Shell... Notation for the cd-command. You can find the help for any command via help command, e.g. `help cd`
There you will find
`cd [-L|-P] [dir]`.
Here, [] means an optional argument and | means "either or".
The "." notation refers to the working directory itself and the ".." notation refers to the working directory's parent directory. Sometimes you can also find

- `ls -r` or
- `ls - -reverse` These to commands are identical and the arguments just read `- -longargument = -abbreviatedarg`. The first variant is fast to write, the second one is easy (fast) to read and understand.

The following commands are possible

3.3.1 Overview: Shell commands

- Navigation `cd, pwd`
- Looking around `ls, less, file`
  - `q` = Quit!
- Manipulating files `cp, mv, rm, mkdir`
  - `rm name`: Remove file `name`.
  - `mv oldname newname`: Rename a file.
  - Wildcards * and ?, e.g. cp *.txt existingdirectory
- Commands `type, which, help, man`, e.g. `help cd`
- I/O redirection
- Expansions
- Permissions
- Jobcontrol
- Writing shell scripts

Link for further reading

3.3.2 Some useful commands for programming

- `set variable=VALUE`: Assign value to new `variable`.
  - `set _oldvariable=VALUE`: Assign value to existing `oldvariable`.
  - `set`: Display all existing variables.
  - `set prefix`: Display all existing variables ending with `prefix`.
  - `set variable=`: Delete `variable`.
- `ECHO %variable%`: Display value of existing variable.

3.4 Git: First steps

When you read the git tutorial, the following picture will become clear. You will learn about * staging * commiting * and what it all means...

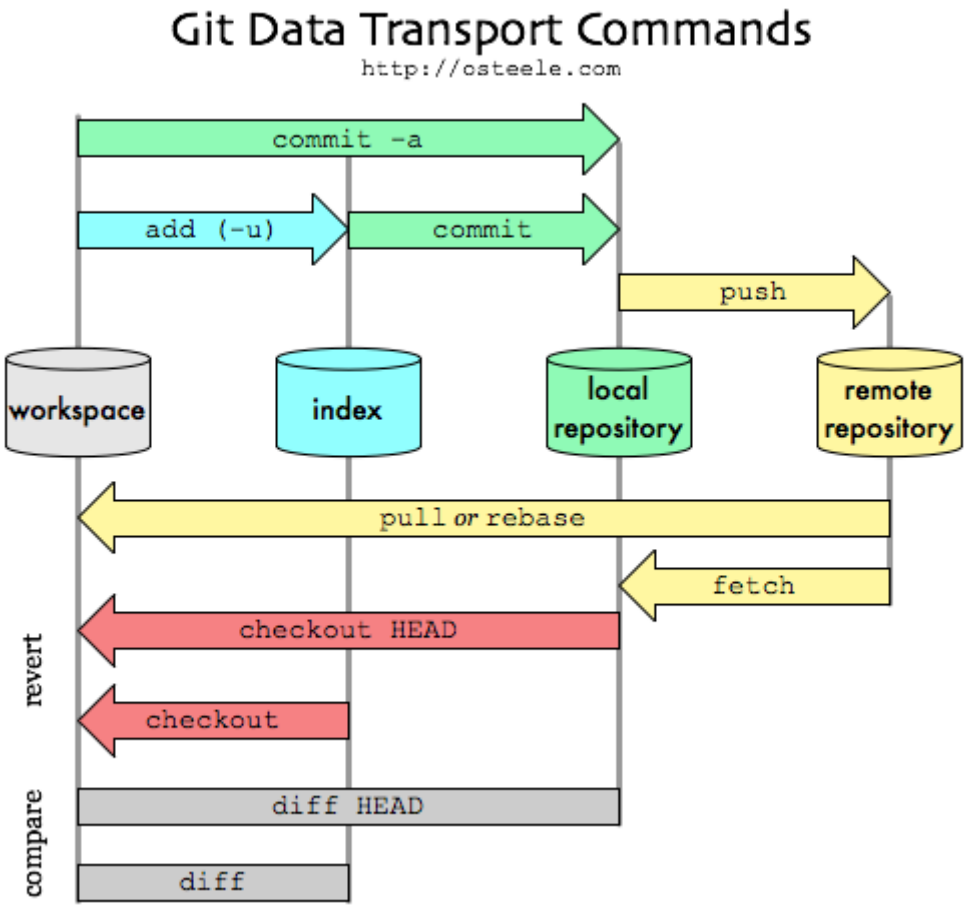

Figure 1: Git Data Transport

3.4.1 Git: Installation on Windows

Git for Windows is provided as installer package by the msysgit project. Download the latest package starting with "Git-", not a "msysgit-..." package (the latter are supposed to be used to build git yourself). Git for Windows comes with a UNIX environment as far as needed by git and also ships with a Bash shell for using the git command line tools.

```
> where git # Shows, where git.exe is located
> git version # Shows the version
> git config --global --edit # Here, you can see, which config file is used.
```

3.4.2 Git: First steps - The staging index

Git internally holds a thing called the index, which is a snapshot of your project files. After you have just created an empty repository, the index will be empty. You must manually stage the files from your working tree to the index using git add:

```
> git add somefile.txt
```

git add works recursively, so you can also add whole folders:

```
> git add somefolder
```

The same applies if you change a file in your working tree - you have to add this change to the index with git add:

```
> git add somefile.txt
```

It's important to realize that the index is a full snapshot of your project files - it is not just a list of the changed files.

If you want to list all the files **currently being tracked** under the branch master, you could use this command:

```
> git ls-tree -r master - -name-only
```

Use `git status` to see untracked files.

3.4.3 Git: Set up the editor of your choice

The git executable will be called with a fixed set of 7 arguments: `path old-file old-hex old-mode new-file new-hex new-mode` As most diff tools will require a different order (and only some) of the arguments, you will have to specify a wrapper script, which in turn calls the real diff tool. Additionally, you have to configure the external diff tool via "git config":

1. Create a wrapper script "git-diff-wrapper.sh" which contains something like. See appendix. As you can see, only the second ("old-file") and fifth ("new-file") arguments will be passed to the diff tool.
2. Type

   ```
   > git config --global diff.external <path_to_wrapper_script>
   ```

at the command prompt, replacing with the path to "git-diff-wrapper.sh", so your ~/.gitconfig contains `external = <path_to_wrapper_script>`

Be sure to use the correct syntax to specify the paths to the wrapper script and diff tool, i.e. use forward slashed instead of backslashes.

Mind the trailing "cat"! (I suppose the '| cat' is needed only for some programs which may not return a proper or consistent return status. You might want to try without the trailing cat if your diff tool has explicit return status).

For further details see here

3.4.4 Git: Useful commands

All commands can be prompted to the console. The `> git...` input is ommitted in the list

Remark:

```
Only if the file is part of a package or project in Rstudio,
"everything" can be done from the git-ribbon in Rstudio!
```

- `config` : Configure git.
  - `config - -list`: List all configurations.
  - `config - -global - -list`: List all global configurations.
  - `config varname`: Display the value of git variable varname.
  - `config - -global alias.myshortname Realgitcommand`: Introduce shortcut for "Real git command", e.g. `config - -global alias.ci "commit -v"`. Be careful: If the command uses a white space "..." (Windows) or '...' (Linux) is needed. Excample for a complex command (working on windows!): `git config - -global alias.hist "log - -pretty=format:'%h %ad | %s%d [%an]' - -graph - -date=short"`.
  - `config - -global - -unset varname`: Delete global `varname`, e.g. `alias.myshortname` shortcut. Does not work, if several `varname` with the same name exist.
  - `config - -global - -unset-all varname`: Delete all global `varname`, e.g. `alias.myshortname` shortcuts
  - `git config - -get-regexp alias`: List all defined alias names.
  - `config - -global - -replace-all core.editor "'pathtoexe/exename`: Set global editor for windows. Be careful with spaces. `--replace-all` only necessary in case of wrongly set multiple editors.
  - `config --global --edit`: View your config file
- `init`: Initialize git repository. This command creates basically a .git directory with subdirectories for objects, refs/heads, refs/tags, and template files. An initial HEAD file that references the HEAD of the master branch is also created. Therefore, right after `git init`, the master branch is not visible after `git branch -av`. This only happens, after the first commit. Before, you're said to be "on an unborn branch", at this point.
  If the repo already exists, the history will not be deleted. The primary reason for rerunning git init is to pick up newly added templates (or to move the repository to another place if - -separate-git-dir is given).
  The default name for the starting branch is **master**.
- `status`: Get the status, displays
  - Tracked **AND** Untracked (=new!) files
  - Modified files
  - Branches

- `diff` Show changes between the working tree and the index or a tree, changes between the index and a tree, changes between two trees, changes between two blob objects, or changes between two files on disk.
  - `diff remotename/branchname:remote/path/file1.txt local/path/file1.txt`: View the differences going from the remote file to the local file.
  - `diff HEAD:local/path/file1.txt remotename/branchname:remote/path/file1.txt`: View the differences in the other direction
  - `diff ref1:path/to/file1 ref2:path/to/file2`: Diff any two files anywhere using this notation. As usual, ref1 and ref2 could be branch names, remotename/branchname, commit SHAs, etc.
- `difftool` is a frontend to git diff and accepts the same options and arguments.
- `mv oldname newname`: Rename file `oldname`.
- `clone`: Clones a repository into a newly created directory, creates remote-tracking branches for each branch in the cloned repository, and creates and checks out an initial branch that is forked from the cloned repository's currently active branch. E.g. `git clone https://github.com/ims-fhs/myfolder`.
  The default shortname for the remote server is then **origin**.
  - `clone - -depth n - -branch branchname repo directory`: Clone history up to nth commit, only `branchname` from `repo` and `folder`.
- `branch newbranch`: Create new branch "newbranch". Note: A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. The only reason nearly every repository has one is that the git init command creates it by default and most people don't bother to change it. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, it moves forward automatically.
  - `branch -d branch`: Delete branch.
  - `branch -v`: Display branches including description.
  - `branch -r`: Display remote branches.
  - `branch -l`: Display local branches.
  - `branch -av`: Display all branches including remotes and description of each.
  - `branch - -merged`: Display all merged branches.
  - `branch -u remoteserver/branch localbranch` (-u = - -set-upstream): Corresponds to `git config branch.localbranch.remote remoteserver` + `git config branch.localbranch.merge refs/heads/localbranch`
  - `branch -m oldname newname`: Rename a branch while pointed to any branch. If you want to rename the current branch, you can simply do `git branch -m newname`
- `add file`: Add file to tracking, e.g. `git add myfile.txt` (Only useful after you start with commitments.)
  If `git add`is not successful, **look at your .gitignore file**.
  - `add .`: Looks at the working tree and adds all those paths to the staged changes if they are either changed or are new and not ignored, it does not stage any 'rm' actions.
  - `add -u`: Looks at all the already tracked files and stages the changes to those files if they are different or if they have been removed. It does not add any new files, it only stages changes to already tracked files.
  - `add -A` is equivalent to `git add .` + `git add -u`.
  - `add folder/*`: Add folder to git.
- `checkout branch`: Switching to specific branche. Update files in the working tree to match the version in the index or the specified tree. If no paths are given, git checkout will also update HEAD to set the specified branch as the current branch.
  To prepare for working on branch, switch to it by updating the index and the files in the working tree, and by pointing HEAD at the branch. Local modifications to the files in the working tree are kept, so that they can be committed to the branch. This command doesn't make any changes to the history.

  - `checkout -b localbranch`: create localbranch and switch to it at the same time (= `git branch name` + `git checkout name`).
  - `checkout -b newbranch remoteserver/remotebranch`: Checkout and create a copy of `remoteserver/remotebranch`. Furthermore, the branch is being set up to track the remote branch, which usually means the remoteserver/remotebranch branch. This is of interest if you you want to join a project. Thus, checkout also works remotely.
    In case of a "**fatal** git checkout: updating paths is incompatible with switching branches...": This occurs when you are trying to checkout a remote branch that your local git repo is not aware of yet See `git remote show origin`. If the remote branch is under "New remote branches" and not "Tracked remote branches" then you need to `fetch` them first.
  - `checkout -b newbranch HEAD`: If you have a **detached HEAD** you can use this command to assign newbranch to the detached HEAD.
  - `checkout shorthash`: checkout old version. See `git log` for possible hash keys. As result you end up with a **detached HEAD**. This is on purpose, as you can play around without consequences. Read the note after command is executed! With a subsequent `git checkout lastbranch` you go back as if nothing had happend. **This is even true, when you modified and commit the old version. All changes are lost!** In case you meesed things up (e.g. `git checkout notlastbranch`) and you don't want to keep changes see `git reset - -hard HEAD`.
  - `checkout @{-1}`: Checkout previous branch. @{-1} is always a way to refer to the last branch you were on.

- `fetch remoteserver remotebranch`: Update remote branch, that is refresh "what others do". Usual case `fetch origin remotebranch`; Get all data from branch on the origin (Normally the remoteserver has shortname origin, see clone!). No merging!
- `merge branch`: Merge branche into the one which is actice (HEAD points onto it. See `git branch -av`): If you changed the same part of the same file differently in two branches you're merging together, Git won't be able to merge them cleanly and an error results. Use "status" to get the unmerged files. In this case the version in HEAD (your master branch, if you have checked out that when you ran your merge command) is the top part of that block (everything above the =======)
- `commit`: Commit changes to current branch (If you just started to work, it will be called `master`)
  - `commit -a -m "text"`: Stage everything (-a, could be also done with a preceding add files) and attach text (Comment) to the commit. See also log.
  - `commit - -amend`: This command takes your staging area and uses it for the commit. You can edit the message as always, but it overwrites your previous commit. If you've made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you'll change is your commit message. The same commit-message editor fires up, but it already contains the message of your previous commit. Do **not use - -amend option after you pushed the last commit**. In this case you have to merge the two.
- `log`: Show the whole commit history.
  - `log -2`: Show last two commits.
  - `log - -pretty=format "%h - %an, %ad:  %s:"`: Nice format, shows hash - authorname, date: commitcomment.
  - `log - -oneline - -decorate`: Show branches including pointers.
  - `log - -follow name`: Continue listing the history of a file `name`. beyond renames (works only for a single file).
- `tag`: list all tags. A Tag is a specific point in history, e.g. the publication of the code (Release V1.0).
  - `tag -a version -m text`: Create an annotated tag with tagging version of type v0.1.0 and message text = "my text" or the commit checksum (something like "67a465f").
  - `tag -a version HASH -m text`: Create an annotated tag as above for old commit HASH.
  - `tag -l "xy"`: Search for tags starting with xy.

- `remote`: Needed when using and managing remote repoitories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Be careful with fork (Github).
  - `remote -v`: Use argument verbose (-v), which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote
  - `remote show remoteserver`: Show information about remoteserver including upstream settings (if set).
  - `remote add remoteserver server`: Add remoteserver of server to tracking. E.g. `git remote add origin https://github.com/ims-fhs/myfolder`. Only visible after a push!
  - `git remote set-url`: Change your remote's URL from SSH to HTTPS with the it remote set-url command. E.g.
    `git remote set-url origin https://github.com/USERNAME/OTHERREPOSITORY.git` And change it back:
    `git remote set-url origin git@github.com:USERNAME/OTHERREPOSITORY.git`
  - `remote rename oldremoteserver newremoteserver`: Rename oldremoteserver to newremoteserver.
  - `remote rm remoteserver`: Remove remoteserver.
  - `remote show remoteserver`: Show details from remoteserver. Good to check wether remoteserver exists.
- `push remoteserver localbranch:remotebranch`: Pushing localbranche to the repository remotename (Normally called origin) in remotebranch. If only one branch is specified, it is assumed that localbranch = remotebranch.
  - `push remote-name :remotebranch`: Pay **attention:** deletes** the remotebranch. Use `push -u remoteserver remotebranch` instead (See below). E.g. `git push -u origin remotebranch`.
  - `push -u remoteserver remotebranch`: Necessary if you want git pull to know what to do. "Upstream" refers to the main repo that other people will be pulling from, e.g. your GitHub repo. The -u option automatically sets that upstream for you, linking your repo to a central one. That way, in the future, Git "knows" where you want to push to and where you want to pull from, so you can use git pull or git push without arguments. A little bit down, this article explains and demonstrates this concept. "Tracking" is essentially a link between a local and remote branch. When working on a local branch that tracks some other branch, you can git pull and git push without any extra arguments and git will know what to do. However, git push will by default push all branches that have the same name on the remote. To limit this behavior to just the current branch, set this configuration option: git config - -global push.default tracking
  - `push - -all remoteserver`: Push (and pull!) all the branches to the remoteserver by default, including the newly created ones. Combination -u - -all possible.
  - `push - -tags remoteserver`: Push (and pull!) all tags to remoteserver.
  - `push - -mirror`: Create identical copy of current folder. **Not tested** "push - -mirror = push - -all + push - -tags".
- `pull remoteserver remotebranch`: Begin tracking file by doing this from the master branch. "pull = fetch + merge"
  - `pull remoteserver remotebranch:localbranch`: **Pay Attention** because you fetch to localbranch (which is ok) but the merge is to the present branch which might not be ok, if remotebranch does not exist. Then you have to run first git checkout -b remotebranch first.
  - `pull -u remoteserver`: Equivalent to - -update-head-ok and different from push -u! By default git fetch refuses to update the head which corresponds to the current branch. This flag disables the check. This is purely for the internal use for git pull to communicate with git fetch, and unless you are implementing your own Porcelain you are not supposed to use it.
- `rebase branch`: Similar to merge - kind of rewrites history, which can be summed up in a single line: **Do not rebase commits that exist in other peoples' repository.** If you follow that guideline,

you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family. When you rebase stuff, you're abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with git rebase and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours. Again, read here.

- `revert`: This command creates a new commit that undoes the changes from a previous commit. This command adds new history to the project (it doesn't modify existing history). Therefore, it is a "safe" way to undo a single commit— it does not "revert" back to the previous state of a project by removing all subsequent commits. In Git, this is actually called a reset, not a revert.
- `reset` **is an exteremly dangerous method**. It can be used if you messed things up and it is ok when you loose **all uncommited changes**. Therefore, you shoud check your last commit before you reset as this commit is where you end up with
  - `reset - -hard HEAD` Permanently undo uncommitted changes in case of a detached HEAD.
- `clean -f`: Remove (= **delete**) untracked files.
  - `clean -f -n`: Don't actually remove anything, just show what would be done.
  - `clean -fd`: Force removing Remove untracked directories in addition to untracked files.
  - `clean -x`: **WARNING** also removes all ignored files!
- `mv oldname newname`: Rename a file. Shorthand for `mv oldname newname` + `git add newname` + `git rm oldname`. For religion see here.
- `rm -r - -cached folder`: remove `folder` from tracking including subfolders (-r = recursive) but keep the files in the folder (- -cached).
- `stash`: Use `git stash` when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the HEAD commit. The latest stash you created is stored in refs/stash; older stashes are found in the reflog of this reference and can be named using the usual reflog syntax (e.g. stash@\protect\T1\textbraceleft0} is the most recently created stash, stash@\protect\T1\textbraceleft1} is the one before it).
  - `stash list`: List the modifications stashed away
  - `stash show`: Inspected modification.
  - `stash apply`: Restore modifications (potentially on top of a different commit)

Concerning fork and branch: All branches on GitHub will be copied in a fork. (Obviously, this doesn't include branches that were never pushed to GitHub in the first place.) But a **fork is a GitHub-to-GitHub operation**; nothing is copied to your PC. It's not quite the same as a Git clone. See the manual for git-clone(1) if you wonder "what's copied when I clone a project?". You cannot always make a branch or pull an existing branch and push back to it, because you are not registered as a collaborator for that specific project. Forking is nothing more than a clone on the GitHub server side:

- without the possibility to directly push back
- with fork queue feature added to manage the merge request

You keep a fork in sync with the original project by:

- adding the original project as a remote
- fetching regularly from that original project
- rebase your current development on top of the branch of interest you got updated from that fetch.

The rebase allows you to make sure your changes are straightforward (no merge conflict to handle), making your pulling request that more easy when you want the maintainer of the original project to include your

patches in his project.
The goal is really to allow collaboration even though direct participation is not always possible. See here for more information, escpessially for the difference between **upstream** and **origin**

For further reading see here and here.

3.5 Advanced Git: Hooks - the pre-commit hook

For example files see appendix. The process is as follows:

- Copy the text in the appendix into a file and save it as `pre-commit` in /.git/hooks/ **without any ending**.
- Depending on the operating system and your path to Rscript. As first line use
  - Windows: #!C:/R/R-3.2.2/bin/x64/Rscript or
  - Mac OSX: #!/Library/Frameworks/R.framework/Versions/3.2/Resources/bin/Rscript
    and make the file executable:

    ```
    chmod +x filename.sh
    ```

    including prefix if existing.
- As described in the R-file, the version increases automatically.
- If you don't want to increase the version run

  ```
  SET inc=FALSE
  git commit -a -m "Your message"
  ```

If you increase the minor or major version, the patch is set to 0.

For a complete list see here.

3.6 GitHub: Create new branch from master

Do it on GitHub

---

4 Getting started with git and github - Useful cases

Order of cases?

4.1 Case 1: You want to join a project

Assume you have an existing github repo https://github.com/ims-fhs/myfolder. and you want to join working on it. Do the following steps:

- `shell > cd working directory` (Go the place for the new folder)
- `shell > git clone https://github.com/ims-fhs/myfolder` (clone repo)

- `shell > git push -u origin master` Eventually: Restart RStudio if the Arrows in the Git Section of RStudio are greyed out.
- If you have already existing files, just copy them in the new folder by hand. Otherwise create a new file... All changes will then be visible in git!
- If you have reached a certain status you want to share, do the following two steps:
  - Rstudio => Git -> Commit
  - Rstudio => Git -> Push (Only works, if you are a collaborator and when previous `push -u` was successful)

4.2 Case 2: You want somebody else to join a project

Assume you worked on "myproject" project or package or folder and you want others to join the project

- Rstudio => Create new project or package or file. If you create a new project or package, you can set up the default to create a **git repository**. If the default "create git repo" was not set, do
  - shell `> cd working directory` (The place for the new folder)
  - shell `> git init` and
  - Rstudio => Tools => Project options => Git/SVN ... Use Git!
  - Rstudio => Tools => Global options => Git/SVN .. Use Git!
- Create github repo https://github.com/ims-fhs/myproject. and add your partner as collaborator (Github repo => Settings => Collaborators and teams.
- shell `> git remote add origin https://github.com/ims-fhs/myproject`
- shell `> git pull origin master` (pull files from origin (defined above!) to master)
- shell `> git push -u origin master`

version control in Rstudio can be only accessed within an project. If you have a folder which is not specified as project or package use the Rstudio project dialog... File => New project => Existing directory => specify parent folder of file

4.3 Case 3: Rename remote branch (Just a better name...)

If you want to rename a branch "both locally and remote" you only have to rename the branche remotely, which means

```
> git push remoteserver remotesrever/oldbranch:refs/heads/newbranch
> git push origin :oldbranch.
```

4.4 Case 4: Commit the work you did on the console

You work on a project that is not manipulated within R Studio. Therefore you cannot use R Studio's integrated Git Support to commit the work you did. Bad for you: You have to get along with the console.`??????????????`

4.4.1 Case 4: You're clever and think to pull the remote repository before you work on the project

1. Before you start working on your project: Open the console, go to the working directory your project is stored.

```
> cd to-working-directory-of-your-project
> git init
> git pull origin master
```

2. Change the files you want to change.
3. Commit them.

```
> git commit -a -m "text"
```

4. Push your changes back to the remote repository.

```
> git push -u origin master
```

4.4.2 Case 4: You forgot to first pull the project, nobody worked on the remote repo

1. You already made your changes when you read this document
2. Commit them.

```
> git commit -a -m "text"
```

3. Push your changes back to the remote repository.

```
> git push -u origin master
```

4.4.3 Case 4: You forgot to first pull the project, somebody worked on the remote repo

1. You already made your changes when you read this document
2. Commit them. > git commit -a -m “text”
3. You cannot push them back because you’re in `detachedHEAD` status > git status

returns you this unhappy message.

4. You have to checkout the remote repository into a new branch > git checkout -b newbranch
5. Now you can merge the newbranch from step 3. with your local master > git checkout master > git merge newbranch
6. Now you can delete the branch from step 4 and push them back to the remote repo > git branch -d newbranch > git push -u origin master

4.5 Case 5: Commit the work you did

You changed a file and want to commit your work. Within R Studio go to Git => Diff. Select the file you want to commit. The lower part of the editor shows your changes:

- red: Everything you deleted
- green: Everything you added

4.5.1 You want to commit everything you changed

Insert a comment to the window topright that explains the changes you made. Press commit.

4.5.2 You want to commit only part of what you changed

- Select the part you don't want to commit, with `shift` held down. Press `unstage selection`.
- Repeat until only the parts you want to commit are shown in the `staged` window.

4.6 Case 6: You changed something you didn't want to change

You made false changes to a file and want to correct them before you commit.
Within R Studio go to Git => Diff. Select the file you want to repair. The lower part of the editor shows your changes:

- red: Everything you deleted
- green: Everything you added
- Select the parts you changed by fault, with `shift` held down. Press `unstage selection`.
- Repeat until only the changes you want to be preserved are shown in the `staged` window.
- Change to the `unstaged` window
- Select the parts you changed by fault, with `shift` held down. Press `Discard selection`.

4.7 Case 7: You have an existing remote branch which you don't want to change but you need to work on it

```
> git checkout -b newbranch remoteserver/remotebranch
```

. . . Wirk on it and commit changes. . .

```
> git push -u origin newbranch
```

Afterwards, you can push from Rstudio again. . .

4.8 Case 8: You want to start work on a package based on your own "work in progress (wip)" branch

We assume you start with master (HEAD points on master). To create an identical copy of master run

```
> git checkout -b wip
```

. . . Start to work on the branch. . .

```
> git commit
> git push -u origin wip
```

At some point deleting the local branch makes perhaps sence:

```
> git branch -d wip
```

4.9 Case 9: Undo a commit and redo

This is most often done when you remembered what you just committed is incomplete, or you misspelled your commit message1, or both.

```
> git commit -m "Something terribly misguided"
```

Leaves working tree as it was before git commit.

```
> git reset - -soft HEAD~
```

Make corrections to working tree files.

```
> git add ...
```

git add whatever changes you want to include in your new commit.

```
> git commit -c ORIG_HEAD
```

Commit the changes, reusing the old commit message. reset copied the old head to .git/ORIG_HEAD; commit with -c ORIG_HEAD will open an editor, which initially contains the log message from the old commit and allows you to edit it. If you do not need to edit the message, you could use the -C option instead.

4.10 Case 10: You worked on something and want to pass it to github

Create a folder in Github. Then open a shell and add the remote repo

```
> git remote add origin https://github.com/account/remotefolder/
```

Think about what to add to gitignore before you push. Do you want /data to be tracked on the server?

Push your master branch to the remote repo:

```
> git push -u origin master
```

See also git push -u - -all!
Check what you have done, if you like:

```
> git remote show origin
```

Reference

4.11 Case 11: Remove directory DIR from remote repository after adding them to .gitignore

All the data git uses info is stored in .git/ , so removing it is fine. Of course, make sure that your working copy is in the exact state that you want it, because everything else will be lost. /.git folder is hidden so make sure you turn on Show hidden files, folders and disks option.

- Step 1. Add the folder path to your directories's root /.gitignore file.
  `path2DIR/DIR/`
  In case DIR is subfolder of your root directory, you only need to add `DIR/`
- Step 2. Remove the folder from your local git tracking, but keep it on your disk.
  `git rm -r - -cached path2DIR/DIR/`
- Step 3. Commit and push your changes to your git repo.

```
> git rm -r - -cached path2DIR/DIR/
> git commit -m "Message"
> git push origin master
```

You can't delete the file from your history without rewriting the history of your repository - you shouldn't do this if anyone else is working with your repository, or you're using it from multiple computers. If you still want to do that, you can use git filter-branch to rewrite the history - there is a helpful guide to that here.

4.12 Case 12: You face a staging area with too many files which you never expect to be part of your project

This means that `git status` shows untracked file on a higher level. It could happen as follows: You had a repo and you cloned it. You ran `git clone url/to/repo.git` from your directory say /home/repos/my. Now after the clone, you try `git status` and it shows untracked files from /home/repos.

So how is this possible?

1. You must have had a .git folder (probably an artifact of previous failed clone, etc) in /home/repos (or even /home etc.)
2. Since you did git clone url/to/repo.git, git would have created a folder repo at /home/repos/my/repo. But you did the git status from /home/repos/my. So git goes to parent folders looking for .git and found it (as per 1) and hence shows untracked files.

So see if 2) is your case. Try changing directory to the folder that git has created. Usually you should do a clone as git clone url/to/repo.git. (. at end) if you are already in the folder where you want the repo to be.

4.13 Case 13: When do I use revert, reset, checkout

If a commit has been made somewhere in the project's history, and you later decide that the commit is wrong and should not have been done, then git revert is the tool for the job. It will undo the changes introduced by the bad commit, recording the "undo" in the history.

If you have modified a file in your working tree, but haven't committed the change, then you can use git checkout to checkout a fresh-from-repository copy of the file.

If you have made a commit, but haven't shared it with anyone else and you decide you don't want it, then you can use git reset to rewrite the history so that it looks as though you never made that commit.

4.14 Case 14: You completely messed up git for unknown reasons but a working older version is on github.

Rename your messed up folder. Then you create a completely(!) new folder with subfolders - renamaing the old folder won't help...
Then copy the necessary bare files, e.g. .png, .R, ... to the new folders.

Create a new package (A project based on existing folders should also work???), initialize git (`git init`) and commit you changes "First commit after rebuilding". Then proceed as in 4.6 using a newbranch(!). History is then clear and you are save as you last changes are in the previous commit and the history is in newbranch. Think about what happend and merge master (your last changes) into newbranch. In the end you need 'git push -u origin newbranch. You can the merge master and newbranch later on when you fee save.

4.15 Case 15: You want run "git add -A ." in each submodule.

```
git submodule foreach - -recursive git add -A .
```

And then you could create a commit in every submodule with:

```
git submodule foreach - -recursive "git commit -m 'Committing in a submodule'"
```

(If you don't have other submodules nested inside those submodules, the - -recursive option is unnecessary.) This is not recommended: You should carefully change into each submodule in turn, and consider how you want to update them, treating each as a standalone repository. Then only commit these new submodule versions in the main project when you have tested that the project as a whole works with those new versions of each submodule.

Use git status to see untracked files. Empty folders are only visible, when you start to commit files in these new folders.

5 Debugging

Some useful hints from here

- It's a great idea to adopt the scientific method. Generate hypotheses, design experiments to test them, and record your results. This may seem like a lot of work, but a systematic approach will end up saving you time. I often waste a lot of time relying on my intuition to solve a bug ("oh, it must be an off-by-one error, so I'll just subtract 1 here"), when I would have been better off taking a systematic approach.
- **Fix it and test it:** Once you've found the bug, you need to figure out how to fix it and to check that the fix actually worked. Again, it's very useful to have automated tests in place. Not only does this help to ensure that you've actually fixed the bug, it also helps to ensure you haven't introduced any new bugs in the process. In the absence of automated tests, make sure to carefully record the correct output, and check against the inputs that previously failed.

---

6 Appendix: Some useful hooks

Source files are from [here]https://gist.github.com/rmflight/8863882

6.1 Pre-commit hook (Tested)

```
#!C:/R/R-3.2.2/bin/x64/Rscript

# License: CC0 (just be nice and point others to where you got this)
# Author: Robert M Flight <rflight79@gmail.com>, github.com/rmflight
#
# This is a pre-commit hook that checks that there are files to be committed,
# and if there are, increments the package version in the DESCRIPTION file.
#
# To install it, simply copy this into the ".git/hooks/pre-commit" file of your
# git repo, change /path/2/Rscript, and make it executable. Note that
# /path/2/Rscript is the same as your /path/2/R/bin/R, or may be in /usr/bin/Rscript
# depending on your installation. This has been tested on both Linux and Windows
# installations.
#
# In instances where you do NOT want the version incremented, add the environment
# variable inc=FALSE to your git call e.g. "inc=FALSE git commit -m "commit message".
# This is useful when you change the major version number for example.

inc <- TRUE # default

# get the environment variable and modify if necessary
tmpEnv <- as.logical(Sys.getenv("inc"))
if (!is.na(tmpEnv)) {
  inc <- tmpEnv
}

# check that there are files that will be committed, don't want to increment
# version if there won't be a commit
fileDiff <- system("git diff HEAD - -name-only", intern = TRUE)

if ((length(fileDiff) > 0) && inc) {

  currDir <- getwd() # this should be the top level directory of the git repo
  currDCF <- read.dcf("DESCRIPTION")
  currVersion <- currDCF[1,"Version"]
  splitVersion <- strsplit(currVersion, ".", fixed = TRUE)[[1]]
  nVer <- length(splitVersion)
  currEndVersion <- as.integer(splitVersion[nVer])
  newEndVersion <- as.character(currEndVersion + 1)
  splitVersion[nVer] <- newEndVersion
  newVersion <- paste(splitVersion, collapse = ".")
  currDCF[1,"Version"] <- newVersion
  currDCF[1, "Date"] <- strftime(as.POSIXlt(Sys.Date()), "%Y-%m-%d")
  write.dcf(currDCF, "DESCRIPTION")
  system("git add DESCRIPTION")
  cat("Incremented package version and added to commit!\n")
}
```

6.2 Post-commit hook (Not tested)

```
#!C:/R/R-3.2.2/bin/x64/Rscript
```

```
# License: CC0 (just be nice and point others to where you got this)
# Author: Robert M Flight <rflight79@gmail.com>, github.com/rmflight
#
# This is a post-commit hook that after a successful commit subsequently
# increments the package version in DESCRIPTION and commits that. Analogous to
# the pre-commit at https://gist.github.com/rmflight/8863882, but useful if you
# only have good reasons for not doing it on the pre-commit.
#
# To install it, simply copy this into the ".git/hooks/post-commit" file of your
# git repo, change /path/2/Rscript, and make it executable. Note that
# /path/2/Rscript is the same as your /path/2/R/bin/R, or may be in /usr/bin/Rscript
# depending on your installation. This has been tested on both Linux and Windows
# installations.
#
# In instances where you do NOT want the version incremented, add the environment
# variable inc=FALSE to your git call e.g. "inc=FALSE git commit -m "commit message"".
# This is useful when you change the major version number for example.

inc <- TRUE # default

# get the environment variable and modify if necessary
tmpEnv <- as.logical(Sys.getenv("inc"))
if (!is.na(tmpEnv)){
  inc <- tmpEnv
}

if (inc){

  currDir <- getwd() # this should be the top level directory of the git repo
  currDCF <- read.dcf("DESCRIPTION")
  currVersion <- currDCF[1,"Version"]
  splitVersion <- strsplit(currVersion, ".", fixed=TRUE)[[1]]
  nVer <- length(splitVersion)
  currEndVersion <- as.integer(splitVersion[nVer])
  newEndVersion <- as.character(currEndVersion + 1)
  splitVersion[nVer] <- newEndVersion
  newVersion <- paste(splitVersion, collapse=".")
  currDCF[1,"Version"] <- newVersion
  currDCF[1, "Date"] <- strftime(as.POSIXlt(Sys.Date()), "%Y-%m-%d")
  write.dcf(currDCF, "DESCRIPTION")
  system("git add DESCRIPTION")
  system('inc=FALSE git commit -m "increment package version"') # inc=FALSE is
  #required, otherwise we end up in an infinite loop
  cat("Incremented package version and committed!\n")
}
```

6.3 git diff wrapper and the .gitconfig file (Windows)

After installing kdiff3, you need the following wrapper to get access to the external gui:

```
#!/bin/sh
# diff is called by git with 7 parameters:
```

```
# path old-file old-hex old-mode new-file new-hex new-mode

"C:/Program Files/KDiff3/kdiff3" "$2" "$5" | cat # Use kdiff3 with 2 arguments
```

which you save as `git-diff-wrapper.sh`.

Adapt your git config file

```
[filter "lfs"]
    clean = git-lfs clean %f
    smudge = git-lfs smudge %f
    required = true
[user]
    name = Christoph Strauss
    email = christoph.strauss@fhsg.ch
[credential]
    helper = wincred # Use helper to avoid typing pwd all the time
[alias]
    hist = log --pretty=format:'%h - %an, %ad: %s' --graph --date=short # Nicly formatted log
[core]
    editor = 'C:/Program Files/KDiff3/kdiff3' # Set core editor to kdiff3
[diff]
    tool = kdiff3
    guitool = kdiff3
    external = 'C:/Users/sqc/git-diff-wrapper.sh' # Use external wrapper for 7 arguments
[difftool "kdiff3"]
    cmd = 'C:/Program Files/KDiff3/kdiff3' $LOCAL $REMOTE # Use kdiff3
    keepBackup = false
    trustExitCode = false
```

Most up to date auth_token = “3bdb5ca1735725c6dd270e487eca41fea1c75f20”