9 – Coordinating parallel development with Git

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

https://www.bccms.uni-bremen.de/cms/people/b-aradi/wissen-progr/python/2022

Outline

- **One repository, multiple branches**

  One developer

  Multiple developers with write access to the same repository

- **Multiple repositories, multiple branches**

  Multiple developers with read-only access to each-others repositories (typical scenario for distributed open source development)

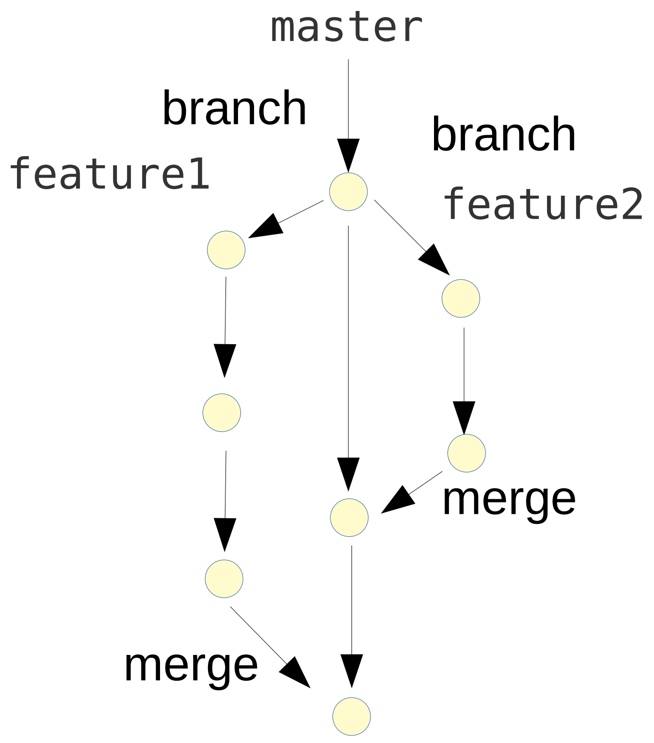
One repository, multiple branches

Branch & merge in one repository

Parallel development of features:

- Multiple independent features are explored at the same time
- A bug has to be fixed in an older version of the code (e.g. last release) without exposing unmature/unfinished new features

Typical workflow



- Features are implemented in **branches** (independent development histories)
- Branches start from the actual state of the main (master) project
- **Every** new feature / significant **change** gets its **own branch**
- If implementation finished, changes are added (merged) to main project
- **Conflicting changes** in parallel branches (e.g. same lines changed), must be manually **resolved** (during merge).

Branch & merge in one repository (#1)

Creating repository

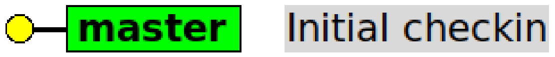
```
mkdir -p ~/gitdemo/one/hello
cd !$
git init
```

Last argument ($) of last command (!)

Create hello.py with following content:

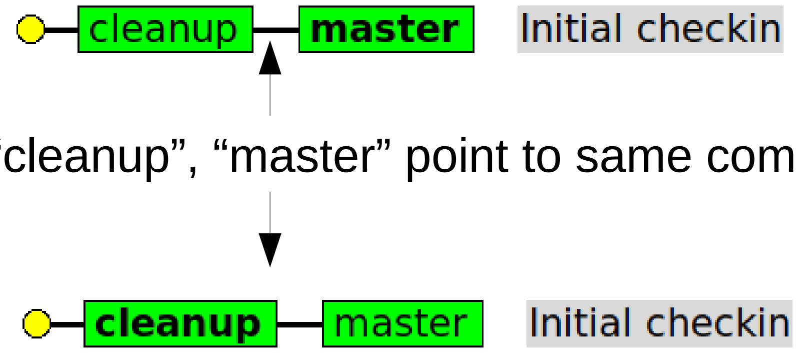
```
print("Hello!")
```

```
git add hello.py
git ci -m "Initial checkin"
```



Creating branch “cleanup”

```
git branch cleanup
```



Changing to branch “cleanup”

```
git co cleanup
   Switched to branch 'cleanup'
git branch
   * cleanup
     master
```

Shows all branches and marks actual one

Branch & merge in one repository (#2)

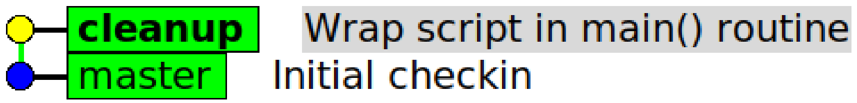
Developing on branch “cleanup”

Change content of hello.py →

```
def main():
    print("Hello!")

if __name__ == "__main__":
    main()
```

```
git add -u
git ci -m "Wrap script in main() routine"
```



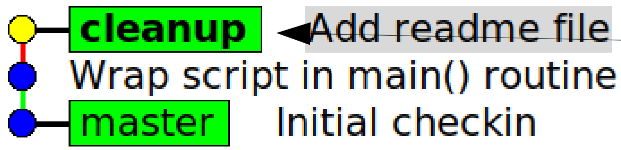
Pointer “cleanup” (actual branch) has been propagated, “master” remains unchanged

Create README.rst →

```
*****
Hello
*****


Trivial greeting projectto demonstrate
the usage of multiple git branches.
```

```
git add README.rst
git ci -m "Add readme file"
```

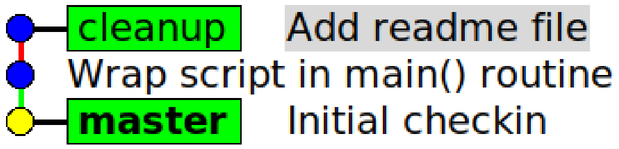


Branch name = **Named pointer** pointing to a given commit (usually last one) in a development line

Branch & merge in one repository (#3)

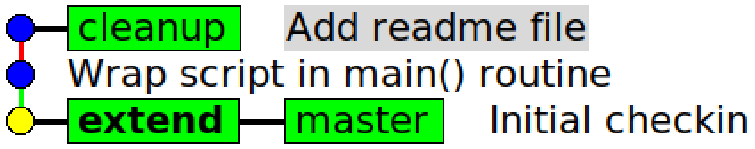
Changing back to “master” branch

```
git co master
   Switched to branch 'master'
```



Creating a new branch “extend” from the state of the project on “master”

```
git co -b extend
   Switched to a new branch 'extend'
```



→ Content of hello.py changed back to the state as in the master branch:

```
print("Hello!")
```

→ File README.rst does not exist
(it only exists in the cleanup branch, but not in master)

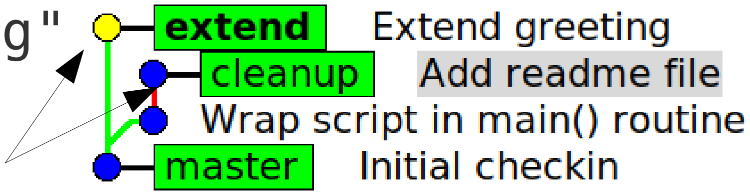
Branch & merge in one repository (#4)

Developing on branch “extend”

Change content of hello.py to:

```
print("Hello, World!")
```

```
git add -u
git ci -m "Extend greeting"
```

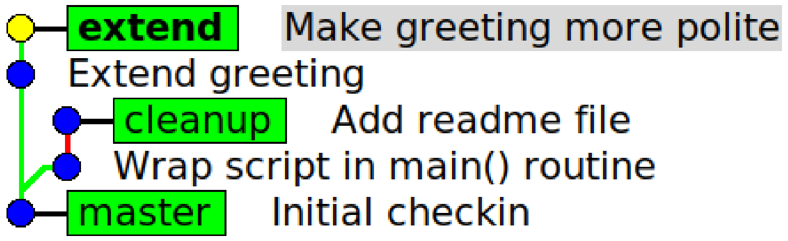


Branches “extend” and “cleanup” **diverge**

Change content of hello.py to:

```
print("Hello, World!")
print("How are you doing?")
```

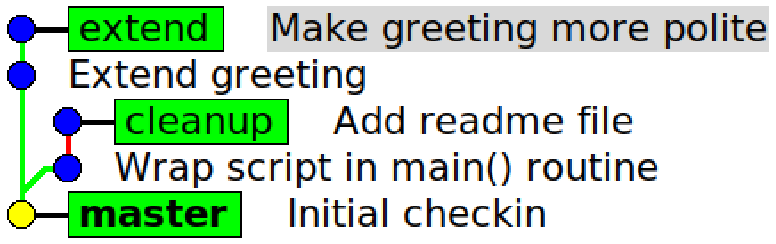
```
git add -u
git ci -m "Make greeting more polite"
```



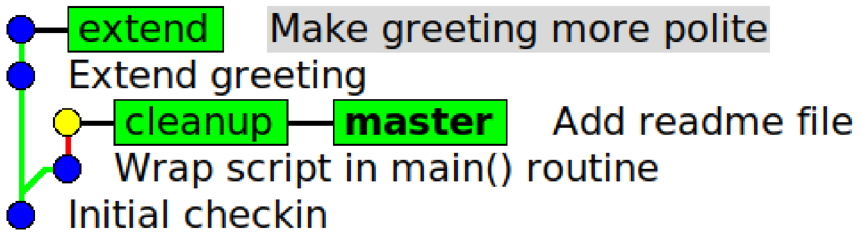
Branch & merge in one repository (#5)

Merging changes from first branch to main project

```
git co master
   Switched to branch 'master'
```

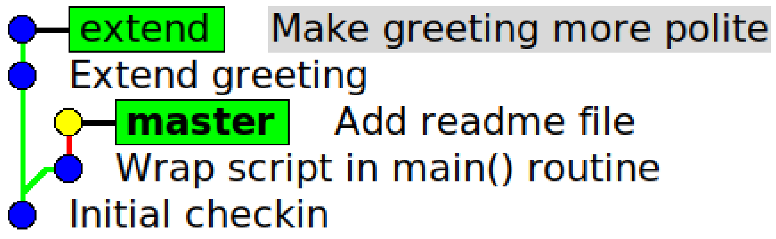


```
git merge cleanup
   Updating b97c415..d66bbe7
   Fast-forward
```



Commit pointed by “cleanup” can be reached from commit pointed by “master” by going only forward in time: Pointer “master” has been simply forwarded to point to “cleanup” (**fast forward**)

```
git branch -d cleanup
     Deleted branch cleanup
     (was d66bbe7).
```



Deleting unnecessary **pointer** (not the commit) “cleanup” (all commits until “cleanup” are contained in the history of the commit pointed by “master”)

Branch & merge in one repository (#6)

Merging changes from second branch to main project

```
git co master
git merge extend
   Auto-merging hello.py
   CONFLICT (content): Merge conflict in hello.py
   Automatic merge failed; fix conflicts and then commit the
   result.
```

Just to make sure we are on the master branch

- The **same lines** have been **changed** on master (due to merge of branch “cleanup”) and on branch “extend”
- Git can not apply both changes simultaneously
- Conflict must be solved manually
- Conflicts are marked in the file

hello.py

```
<<<<<<< HEAD

def main():
    print("Hello!")

if __name__ == "__main__":
    main()
=======
print("Hello, World!")
print("How are you doing?")
>>>>>>> extend
```

Branch & merge in one repository (#7)

Fix merge conflicts and commit merge

```
<<<<<<< HEAD

def main():
    print("Hello!")

if __name__ == "__main__":
    main()
=======
print("Hello, World!")
print("How are you doing?")
>>>>>>> extend
```

Conflicting change on current (master) branch

Conflicting change on branch being merged ("extend")

Manually resolved version

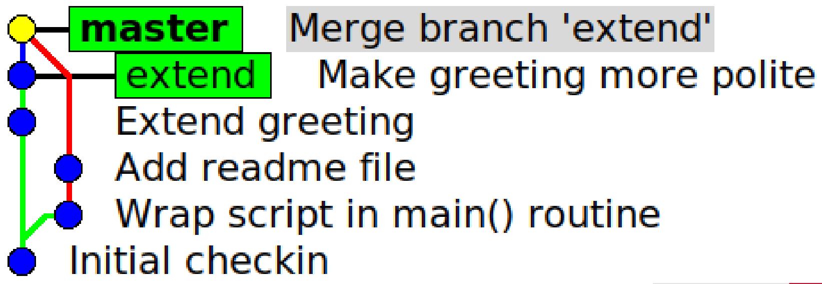
```
def main():
    print("Hello, World!")
    print("How are you doing?")

if __name__ == "__main__":
    main()
```

```
git add hello.py
git ci
```

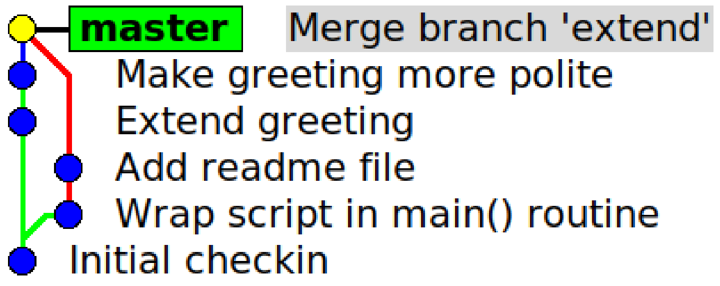
Tells git that conflict has been manually resolved

Commits merge (changes from branch + manual changes for conflict resolution)



Branch & merge in one repository (#8)

```
git branch -d extend
   Deleted branch extend (was 779ffb1).
```



```
master   Merge branch 'extend'
  Make greeting more polite
  Extend greeting
  Add readme file
  Wrap script in main() routine
Initial checkin
```

Deleting superfluos pointer, since commits on “extend” has been merged into master → they are part of the history of the commit where “master” points to (they can be reached from “master” by going only backwards in time)

Master branch contains all changes from both feature branches + all changes necessary to resolve the conflicts between them

Fast forward vs. explicit merge commit

Advantages of fast-forward merges

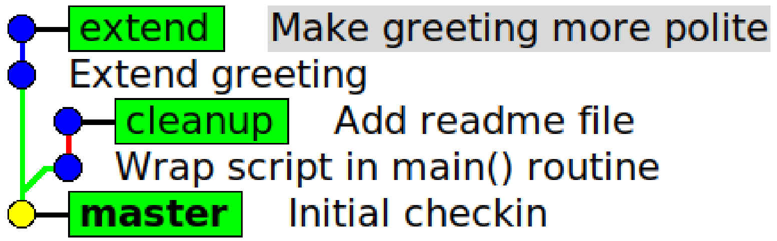
- No extra merge commits in the logs
- Keeps git-history linear (some projects prefer such history…)

Advantages of explicit merge commits

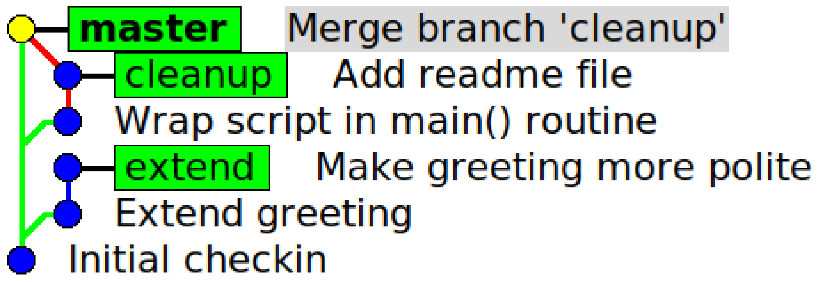
- It is clear, where the changes came from (feature branch)
- Feature can be easily removed (by removing/reverting) merge commit

Forcing merge commits

- The **--no-ff** option forces an explicit merge commit, even if fast forward were possible

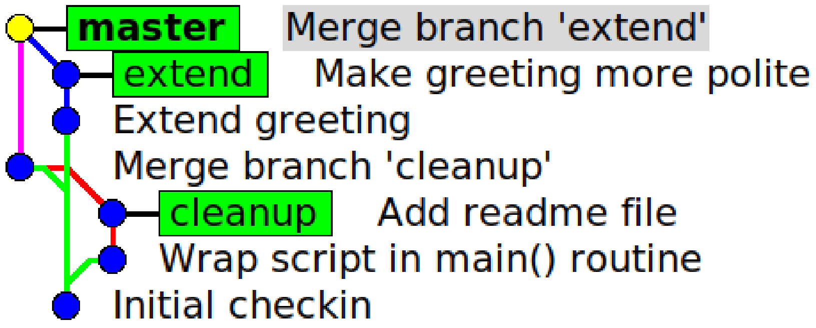

```
extend    Make greeting more polite
Extend greeting
  cleanup    Add readme file
  Wrap script in main() routine
master    Initial checkin
```

```
git merge --no-ff cleanup
   Merge made by the
   'recursive' strategy.
```


```
master    Merge branch 'cleanup'
  cleanup    Add readme file
  Wrap script in main() routine
  extend    Make greeting more polite
  Extend greeting
Initial checkin
```

Fix conflict in hello.py

```
git add hello.py
git ci
git merge --no-ff extend
   Auto-merging hello.py
   CONFLICT (content): Merge
   conflict in hello.py
```


```
master    Merge branch 'extend'
  extend    Make greeting more polite
  Extend greeting
Merge branch 'cleanup'
    cleanup    Add readme file
    Wrap script in main() routine
  Initial checkin
```

Fast forward not possible here, git automatically makes merge commit here, but option can still be set.

Multiple repositories, multiple branches

Branch & merge in two repositories

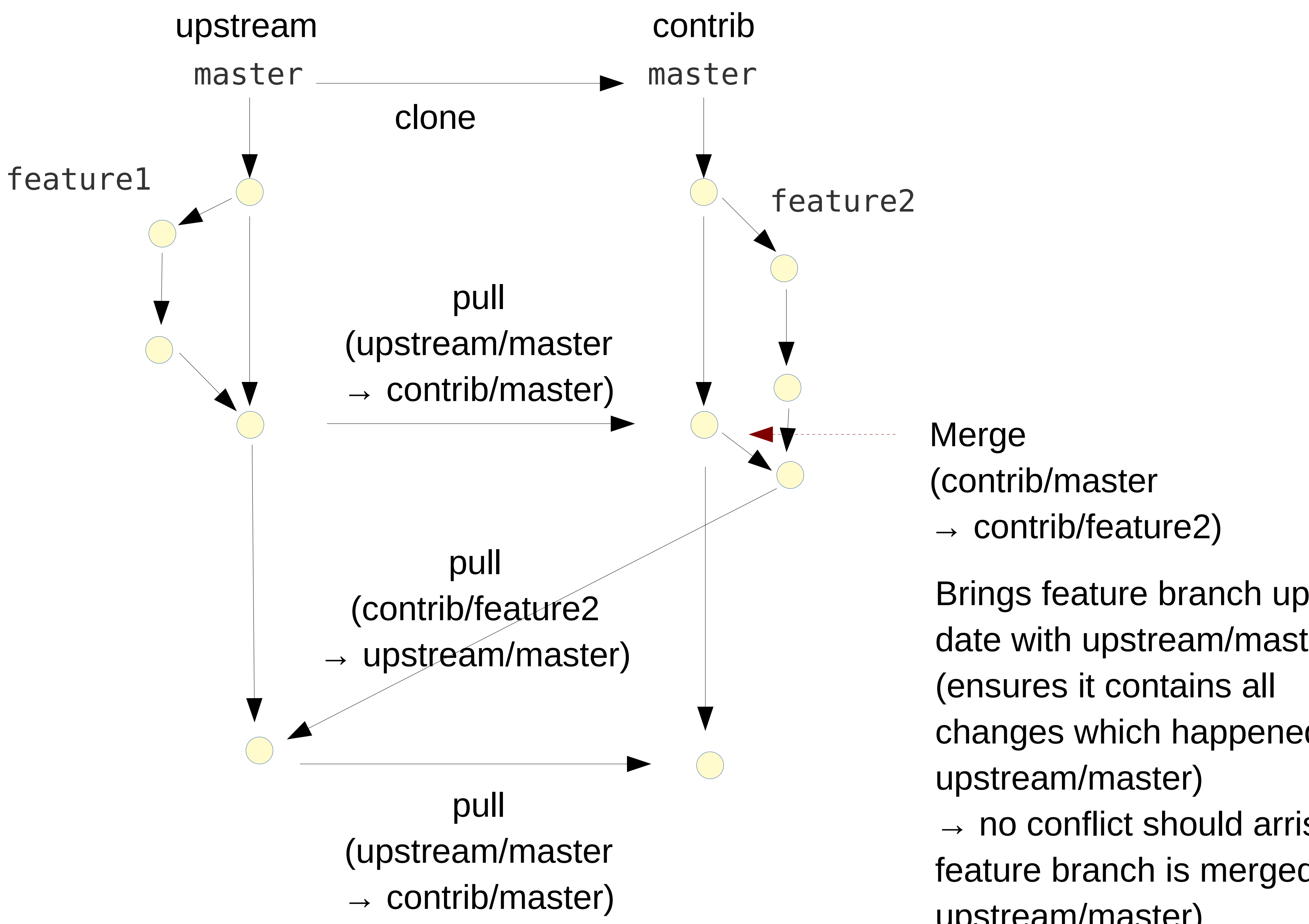
Typical scenario (e.g. open source projects)

- Program is developed by multiple developers simultaneously
- There is one “official” (upstream) version of the project with main developer(s) (developer(s) in charge) and several contributors.
- Parties have only read-only access to each others repositories

Typical workflow

- Every developer regularly synchronizes **master** to keep it identical to **upstream/master**
- Each developer implements features in **feature branches** derived from his/her master branch
- The master branch of the contributors is never modified directly, only by synchronization with upstream master
- If feature finished, **main developer pulls contributors feature branch** and merges it into upstream master

Branch & merge in two repositories



Brings feature branch up-to-date with upstream/master (ensures it contains all changes which happened on upstream/master)
→ no conflict should arrise if feature branch is merged into upstream/master)

- Workflow works very well also with **large nr. of contributors**

Branch & merge in two repositories (#1)

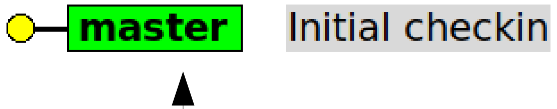
Main developer: create “official” reporitory

```
mkdir -p ~/gitdemo/multi/upstream/hello
cd !$
git init
```

Create hello.py with following content:

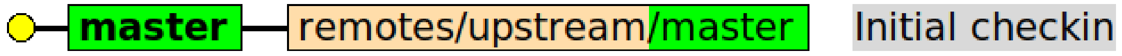
```
print("Hello!")
```

```
git add hello.py
git ci -m "Initial checkin"
```



Contributor: clone “official” repository

```
mkdir -p ~/gitdemo/multi/contrib1
cd !$
git clone -o upstream ~/gitdemo/multi/upstream/hello
   Cloning into 'hello'...
```



How we refer to cloned repository (default: origin)

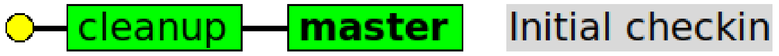
Points to the position of master in current repo

Points to the position of master in upstream repo

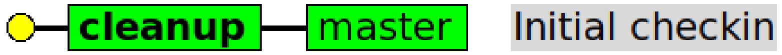
Branch & merge in two repositories (#2)

Main developer: develop feature in feature branch and merge into master

```
git branch cleanup
```



```
git co -b cleanup
```



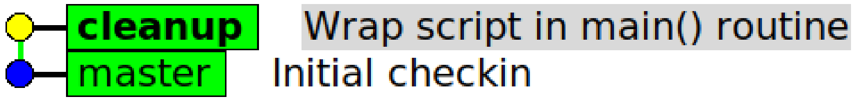
```
   Switched to a new branch 'cleanup'
```

Change content of hello.py to:

```
def main():
    print("Hello!")

if __name__ == "__main__":
    main()
```

```
git add -u
git ci -m "Wrap script in main() routine"
```



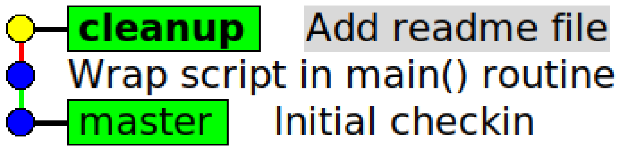
Branch & merge in two repositories (#3)

Create README.rst
with following content:

```
*****
Hello
*****

Trivial greeting project to demonstrate
the usage of multiple git branches.
```

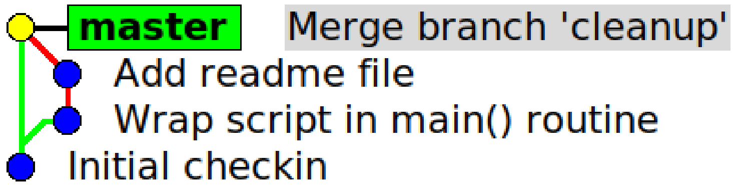
```
git add README.rst
git ci -m "Add readme file"
```



```
cleanup  Add readme file
Wrap script in main() routine
master   Initial checkin
```

```
git co master
git merge --no-ff cleanup
git branch -d cleanup
```

Optional, in case you want
to avoid fast-forward

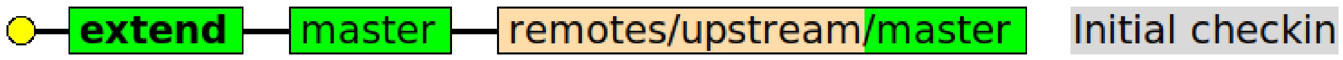


```
master  Merge branch 'cleanup'
  Add readme file
  Wrap script in main() routine
Initial checkin
```

Branch & merge in two repositories (#4)

Contributor: develop feature in a branch

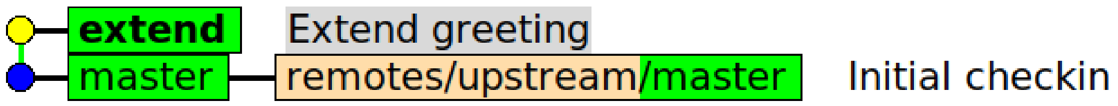
```
git co master
git co -b extend
  Switched to a new branch 'extend'
```



Change content of hello.py to:

```
print("Hello, World!")
```

```
git add -u
git ci -m "Extend greeting"
```

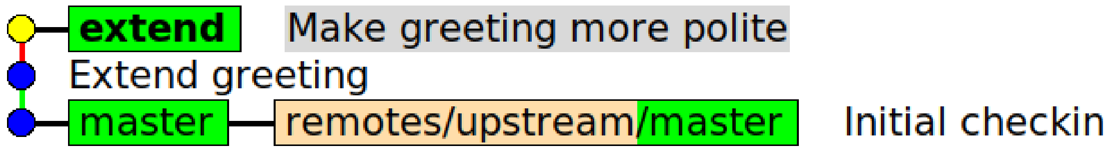


Branch & merge in two repositories (#5)

Change content of hello.py to:

```
print("Hello, World!")
print("How are you doing?")
```

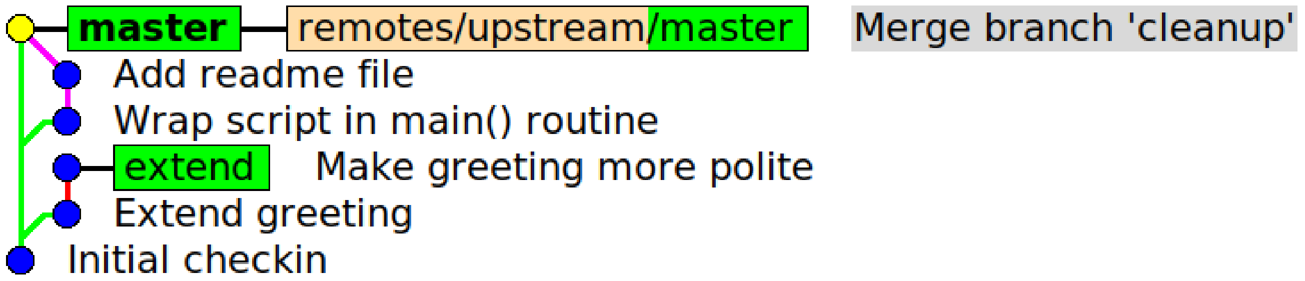
```
git add -u
git ci -m "Make greeting more polite"
```



Contributor: synchronize master branch with upstream master

```
git co master
git pull --ff-only upstream master
```

Fast-forward only: Would fail if master had been modified apart of being pulled from upstream/master



→ Master of developer 2 identical to upstream/master

Branch & merge in two repositories (#6)

Contributor: merge master into feature branch, fix eventual conflicts

```
git co extend
git merge master
   CONFLICT (content): Merge conflict in hello.py
```

```
<<<<<<< HEAD
def main():
    print("Hello!")

if __name__ == "__main__":
    main()
=======
print("Hello, World!")
print("How are you doing?")
>>>>>>> extend
```

$\longrightarrow$

```
def main():
    print("Hello, World!")
    print("How are you doing?")

if __name__ == "__main__":
    main()
```

```
git add hello
git ci
   [extend 2825180] Merge branch 'master' into extend
```

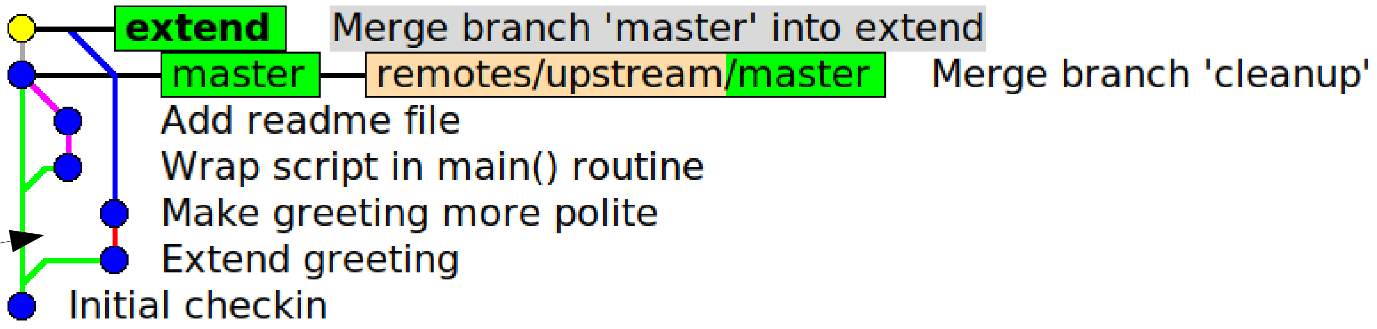
Branch & merge in two repositories (#7)

- Updated feature branch now contains all changes from the original feature branch as well as all changes happened on upstream/master since the feature branch was created

Current feature branch head containing both change sets

Parallel changes on master

Feature branch changes



```
extend  Merge branch 'master' into extend
master  remotes/upstream/master  Merge branch 'cleanup'
Add readme file
Wrap script in main() routine
Make greeting more polite
Extend greeting
Initial checkin
```

→ Feature branch is ready to be merged into upstream/master without any conflicts (as they had been resolved by developer 2)

→ Issue **pull request**: Ask main developer to pull and merge the updated feature branch into upstream/master

Branch & merge in two repositories (#7)

Main developer: Fetch and investigate changes from contributor

```
git remote add contrib ../../contrib/hello
git remote -v
   contrib  ../../contrib/hello (fetch)
   contrib  ../../contrib/hello (push)
```

Register contributors repository (needed only once)

```
git fetch contrib extend
   From ../../contrib/hello
    * branch            extend     -> FETCH_HEAD
    * [new branch]      extend     -> contrib/extend
```

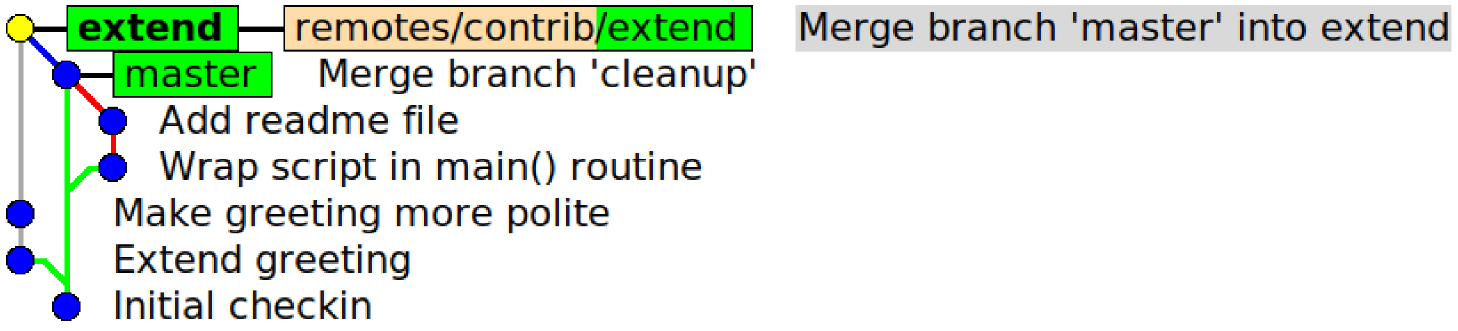
Fetch content of “extend” branch from contrib repository

```
git co extend
   Branch 'extend' set up to track remote branch
   'extend' from 'contrib'.
   Switched to a new branch 'extend'
```

Check out contrib/extend as extend for further inspection

Branch & merge in two repositories (#8)

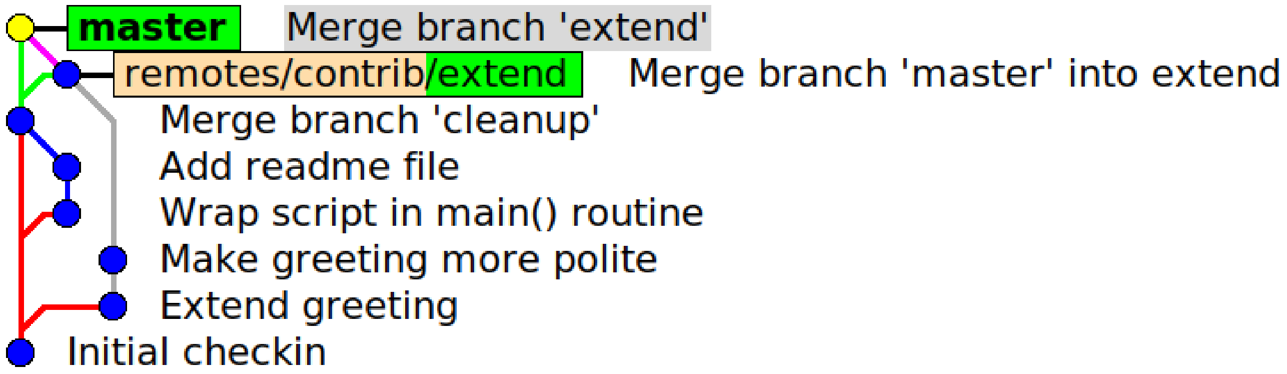
- Main developer has an exact local copy of contributors feature branch



```
Merge branch 'master' into extend
Merge branch 'cleanup'
Add readme file
Wrap script in main() routine
Make greeting more polite
Extend greeting
Initial checkin
```

Main developer: merge feature branch into upstream/master

```
git co master
git merge --no-ff extend
git branch -d extend
```



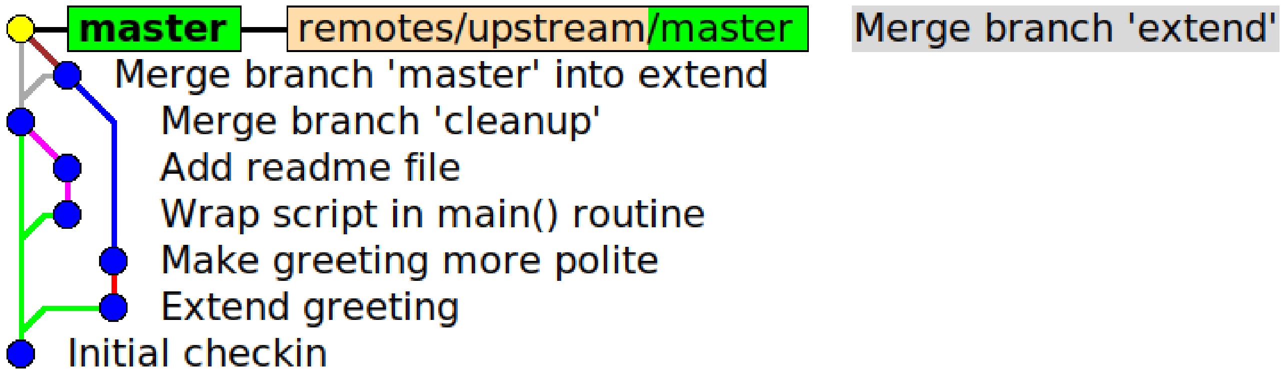
```
Merge branch 'extend'
Merge branch 'master' into extend
Merge branch 'cleanup'
Add readme file
Wrap script in main() routine
Make greeting more polite
Extend greeting
Initial checkin
```

→ **Upstream/master contains all previous commits + changes from contributor**

Branch & merge in two repositories (#9)

Contributor: sync master branch with upstream/master

```
git co master
git pull --ff-only upstream master
git branch -d extend
```



```
master  remotes/upstream/master  Merge branch 'extend'
Merge branch 'master' into extend
Merge branch 'cleanup'
Add readme file
Wrap script in main() routine
Make greeting more polite
Extend greeting
Initial checkin
```

Master branch absolulteley indentical to upstream/master!

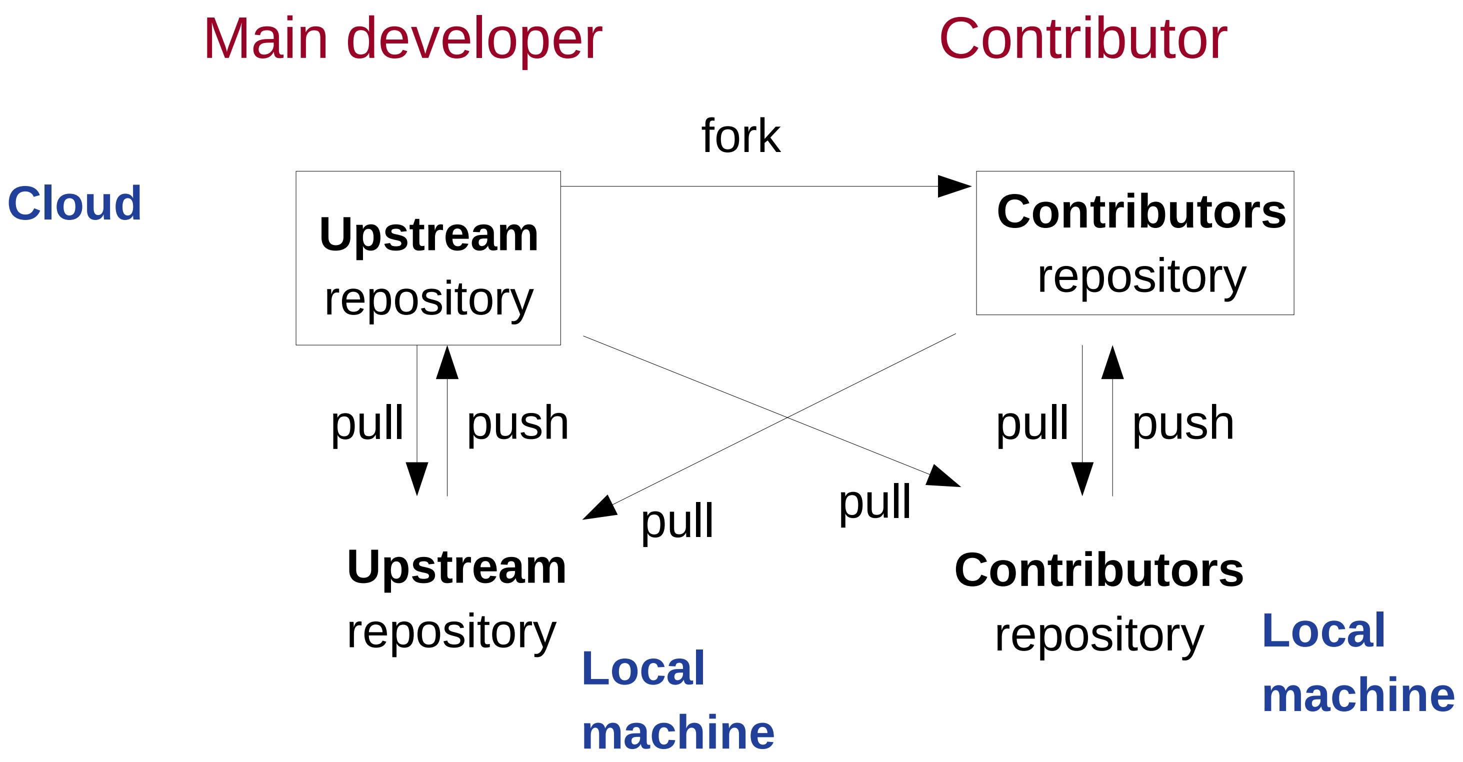
Publishing a repository

Publish repository, so that others can clone it and pull from it

- Allow read-access to repository in local file system (multi-user environment)
- Upload repository to public file-/webserver
- Send repository (including .git/) as an archive
- Allow access via git daemon (see “man git-daemon”, be careful with it!)
- Publish repository on a git hosting site (e.g. GitHub, GitLab, Bitbucket)
    - Very convenient and de-facto standard for open-source projects
    - **Note:** Those site are **commercial** ones (with commercial interests), but usually offer free of charge services for private persons, students, etc.
- Run your own git hosting infrastructure (e.g. self-hosted GitLab)

Remote git-hosting

- Public git hosting sites use the “fork-pull-push” workflow
- Similar to “branch & merge in two repositories”
- Local repository is “published” via **push** to public hosting site
- Changes from other repositories are imported via **pulls** from the public repositories at the hosting site



Some random final git notes

- Git is very flexible and powerful, allowing for almost **arbitrary workflow**
  → Most project document their git-workflow (e.g. DFTB+ git-workflow)
  → If you start your own project, pick a common one (e.g. **GitHub flow**)!

- Public git-hosting sites are usually offering very good tutorials on git and git-workflows (see for example GitHub guides)

- The free “git book” **Pro Git** contains an excellent introduction to git.

- Instead of merging a source branch into a target one, one can also rebase the target branch upon the source branch. (**Rebasing** is not trivial, so make sure you understand its consequences, before you do it.)