Concurrent development with Git

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

https://www.bccms.uni-bremen.de/people/b-aradi/wissen-progr/python/2023

Outline

- One repository, multiple branches
- Multiple repositories, multiple branches

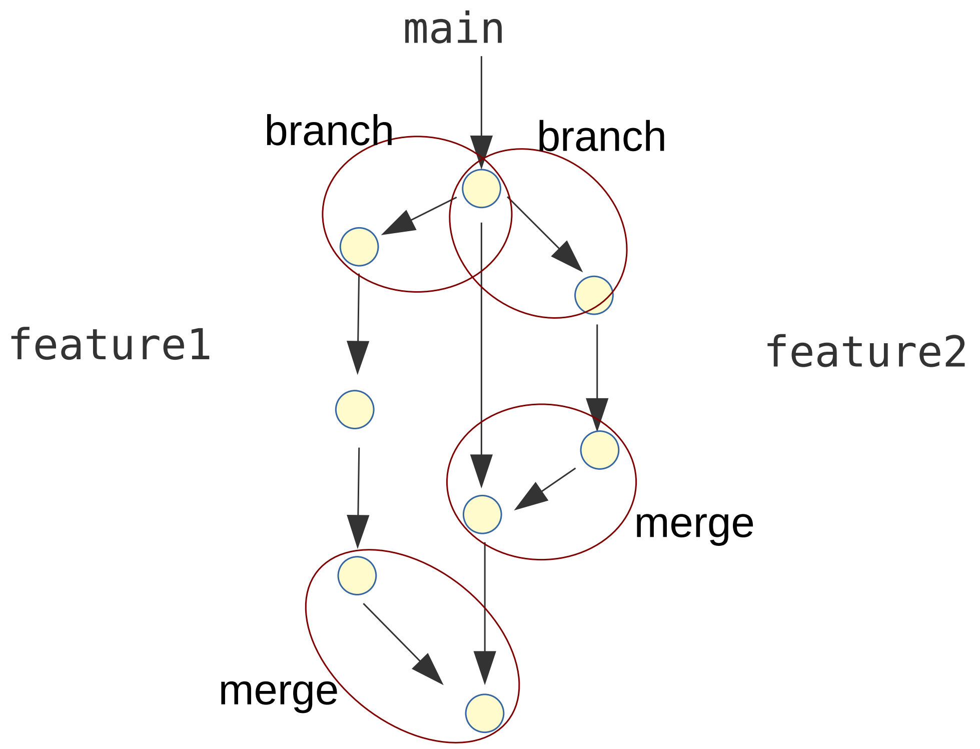
One repository, multiple branches

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 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

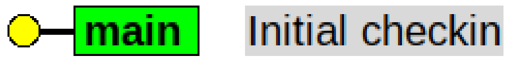
```
mkdir -p gitdemo/hello
cd !$
git init
```

Last argument ($)
of last command (!)

hello.py

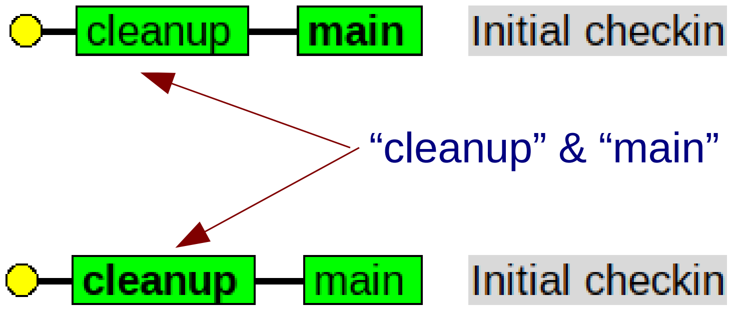
```
print("Hello!")
```

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



Creating branch **cleanup**

```
git branch cleanup
```



Switching to branch **cleanup**

```
git switch cleanup
```

Checking current branch

```
git branch
```

```
* cleanup
  main
```

All branches, current one marked with "*"

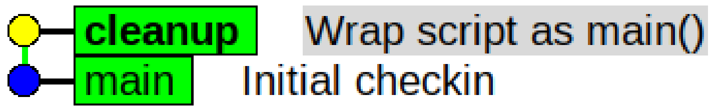
Branch & merge in one repository (#2)

Developing on branch **cleanup**

hello.py

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

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

```
git add -u
git commit -m "Wrap script as main()"
```



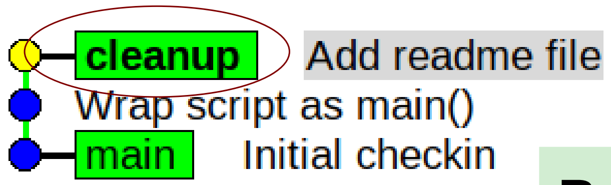
Pointer “cleanup” (actual branch) advanced, “main” remains.

README.rst

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

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

Create README.rst

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

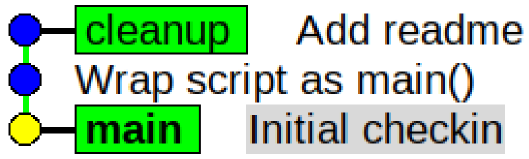


Branch name = Named pointer pointing to a given commit representing the end of a named development line

Branch & merge in one repository (#3)

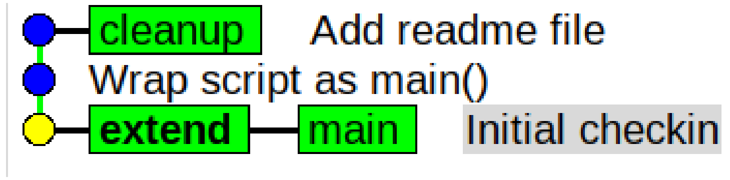
Switching back to **main** branch

```
git switch main
```



Creating a new branch **extend** starting from the state of the project on “main”

```
git switch -c extend
```



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

hello.py

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

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

Branch & merge in one repository (#4)

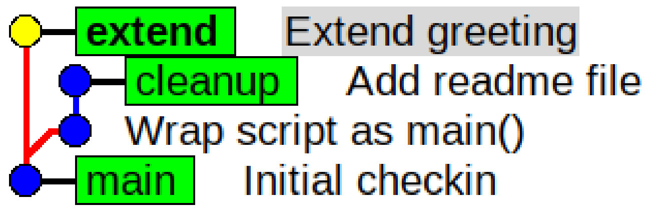
Developing on branch "extend"

Change file content

hello.py

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

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



Developing on branch "extend"

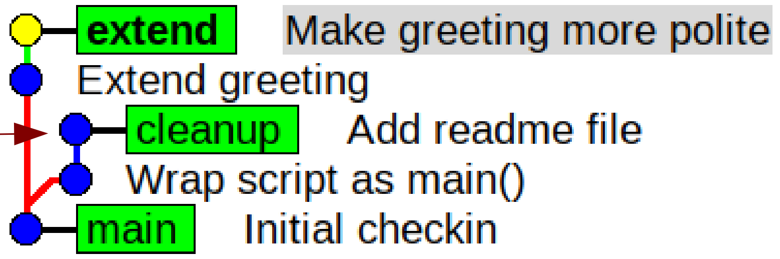
Change file content

hello.py

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

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

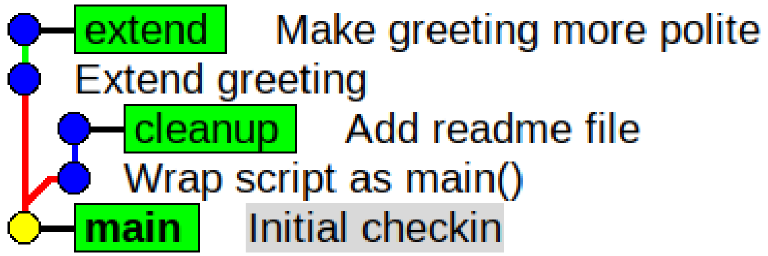
Branches **extend** and **cleanup** diverged



Branch & merge in one repository (#5)

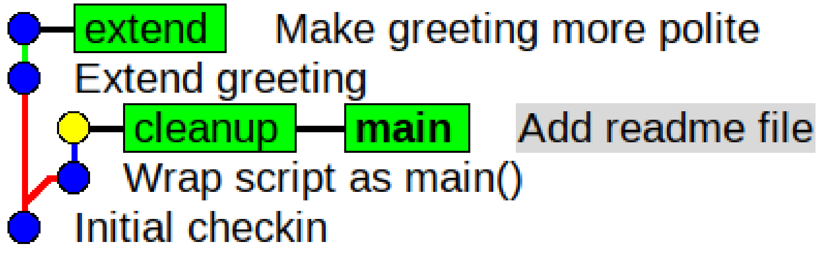
Merging changes from first branch to **main** branch

```
git switch main
```



```
git merge cleanup
```

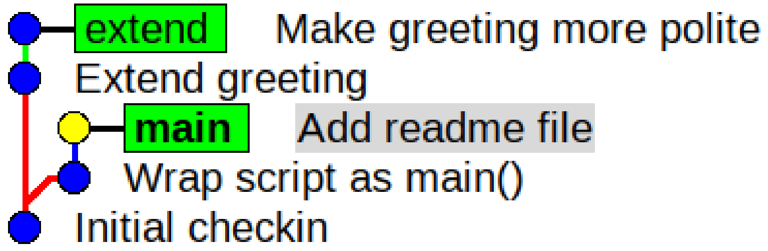
```
Updating b97c415..d66bbe7
Fast-forward
```



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

```
git branch -d cleanup
```

```
Deleted branch cleanup ...
```



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

Branch & merge in one repository (#6)

Merging changes from second branch to main project

```
git switch main
git merge extend
```

Just to make sure we are on the main branch

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

hello.py

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

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

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

Branch & merge in one repository (#7)

Fix merge conflicts and commit merge

hello.py
(resolved version)

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

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

hello.py

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

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

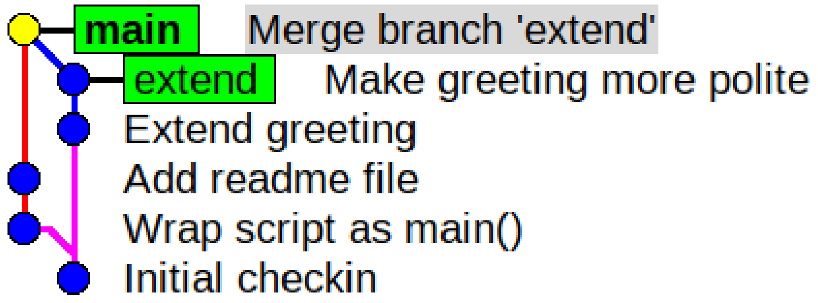
Conflicting change on current (main) branch

Conflicting change on branch being merged (extend)

```
git add hello.py
git commit
```

Tells git that conflict has been manually resolved

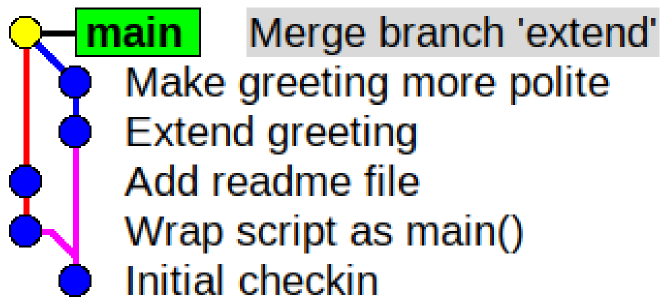
Commits merge
(= changes from merged branch
+ manual changes for conflict resolution)



Branch & merge in one repository (#8)

```
git branch -d extend
```

```
Deleted branch extend (was 779ffb1).
```



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

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

Main 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) a single merge commit

Forcing merge commits

- The **--no-ff** option can enforce an explicit merge commit, even when fast forward were possible

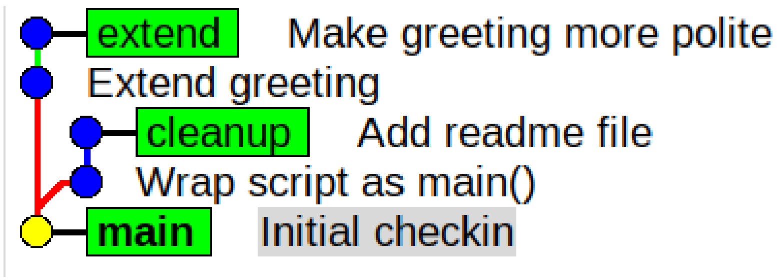
```
git merge --no-ff cleanup
```

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

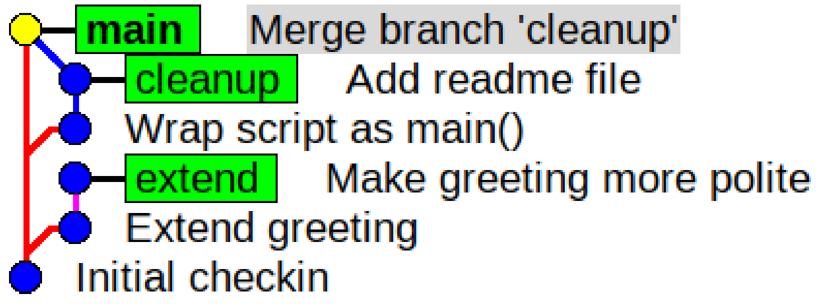
```
git merge --no-ff extend
```

```
CONFLICT (content): Merge conflict in
hello.py
```

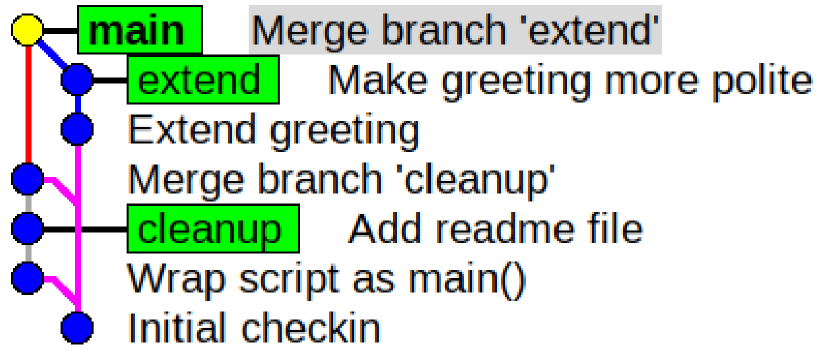
```
git add hello.py
git commit
```



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



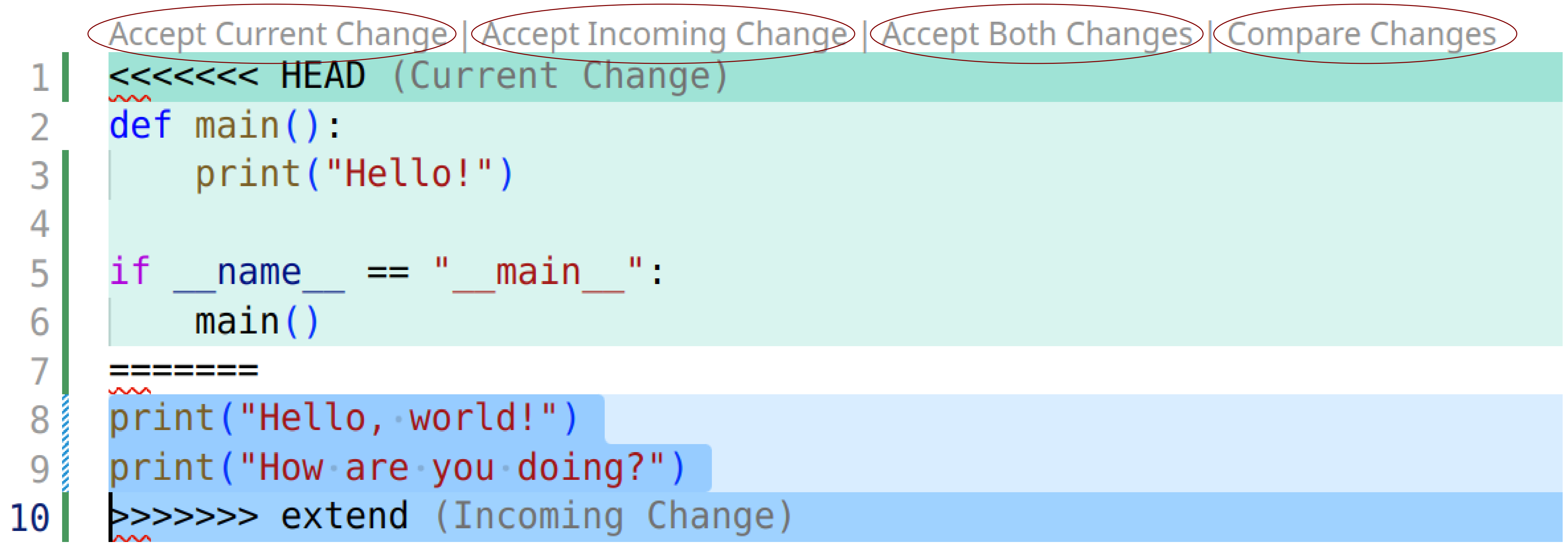
```
main   Merge branch 'cleanup'
cleanup   Add readme file
Wrap script as main()
extend   Make greeting more polite
Extend greeting
Initial checkin
```



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

Mainpulating conflicts in IDEs

- Most IDEs allow the manpulation of files with conflict markers:


Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes

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

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

Multiple repositories, multiple branches

Branch & merge in two repositories

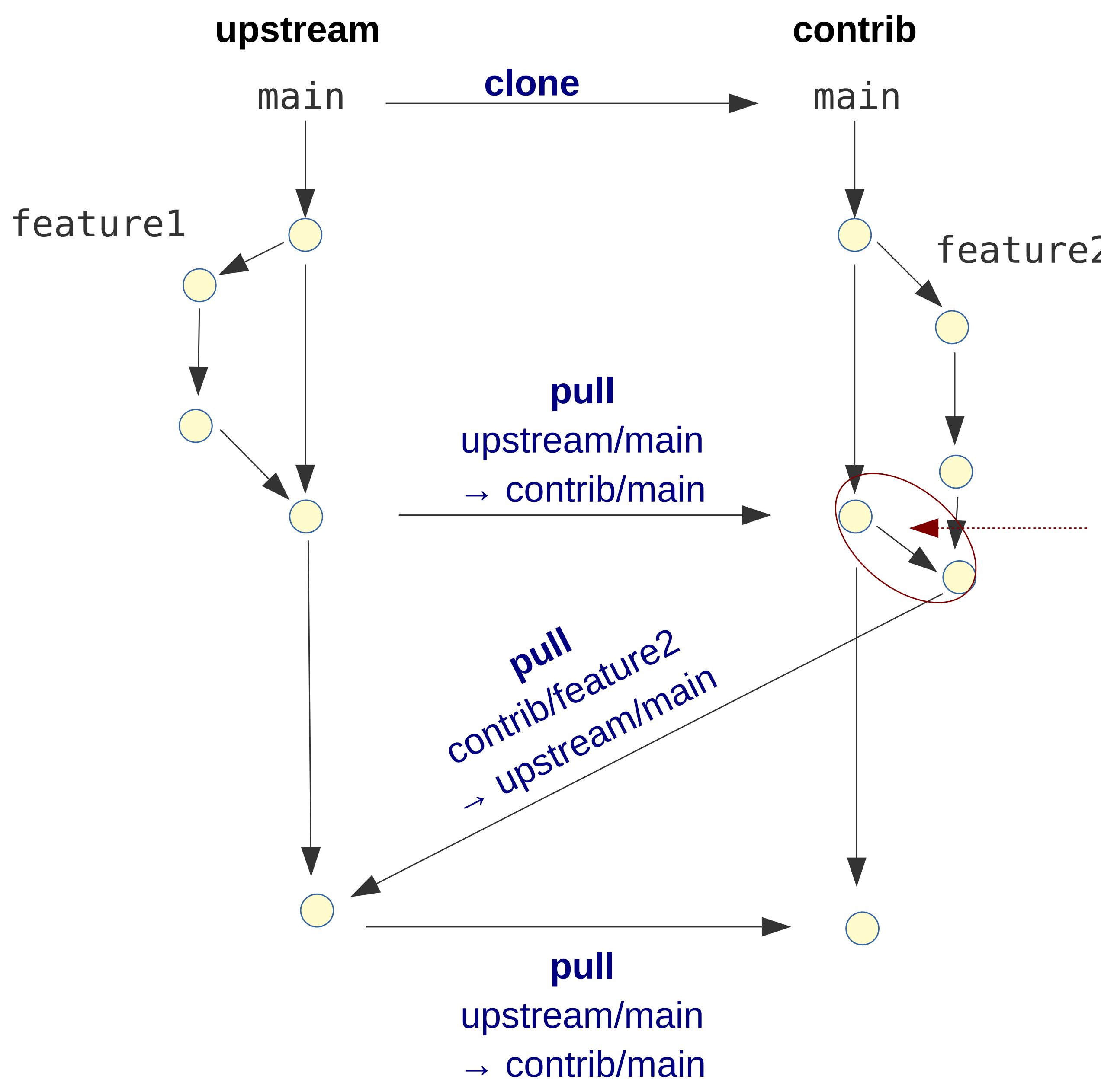
Typical scenario (e.g. open source projects)

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

Branch & merge in two repositories



- Workflow works very well also with **large nr. of contributors**
- Developers need write access only to their own repositories

Merge (contrib/main → contrib/feature2)

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

Branch & merge in two repositories (#1)

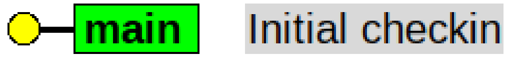
Developer 1: create “official” reporitory

```
mkdir -p gitdemo/devel1/hello
cd !$
git init
```

hello.py

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

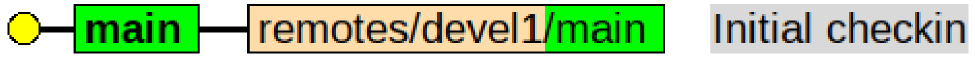
```
git add hello.py
git commit -m "Initial checkin"
```



Developer 2: clone repository of Developer 1

```
mkdir -p gitdemo/devel2
cd !$
git clone -o devel1 ../devel1/hello
```

How we refer to cloned repository (default: origin)



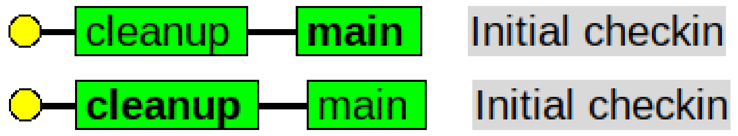
main in current repo

main in devel1’s repo

Branch & merge in two repositories (#2)

Developer 1: develop feature in feature branch and merge into main

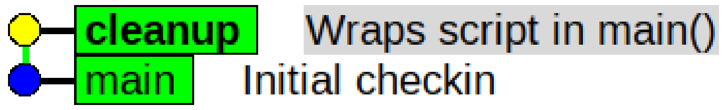
```
git branch cleanup
git switch cleanup
```



hello.py

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

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

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



Branch & merge in two repositories (#3)

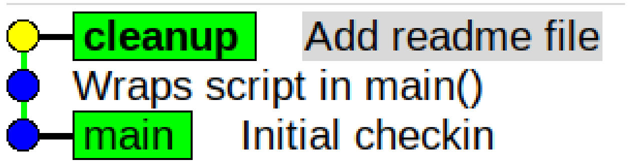
README.rst

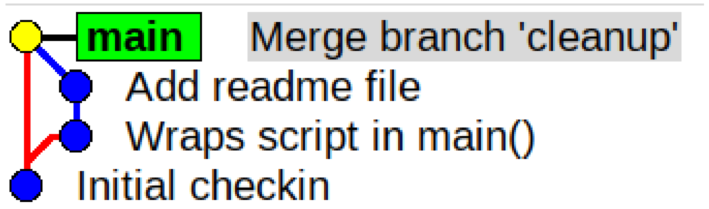
```
*****
Hello
*****

Trivial demo project.
```

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

```
git switch main
git merge --no-ff cleanup
git branch -d cleanup
```

Optional, in case you want to avoid fast-forward

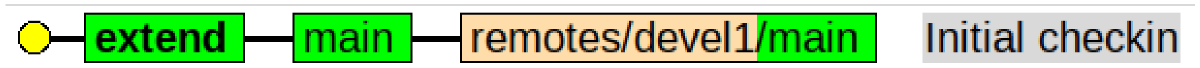




Branch & merge in two repositories (#4)

Developer 2: develop feature in a branch

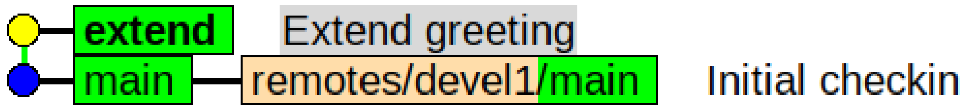
```
git switch main
git switch -c extend
```



hello.py

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

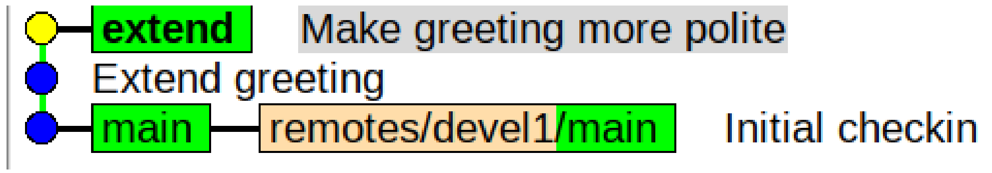
```
git add -u
git commit -m "Extend greeting"
```



hello.py

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

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

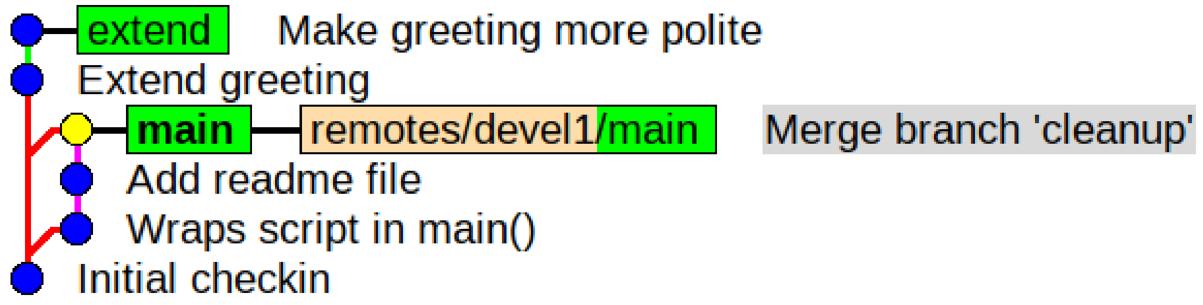


Branch & merge in two repositories (#5)

Developer 2: synchronize main branch with devel1/main

```
git switch main
git pull --ff-only devel1 main
```

Fast-forward only: Would fail if main had been modified apart of being pulled from devel1/main



→ Main branch of developer 2 identical to devel1/main

Branch & merge in two repositories (#6)

Developer 2: merge updated main into feature branch, fix eventual conflicts

```
git switch extend
git merge main
```

hello.py

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

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

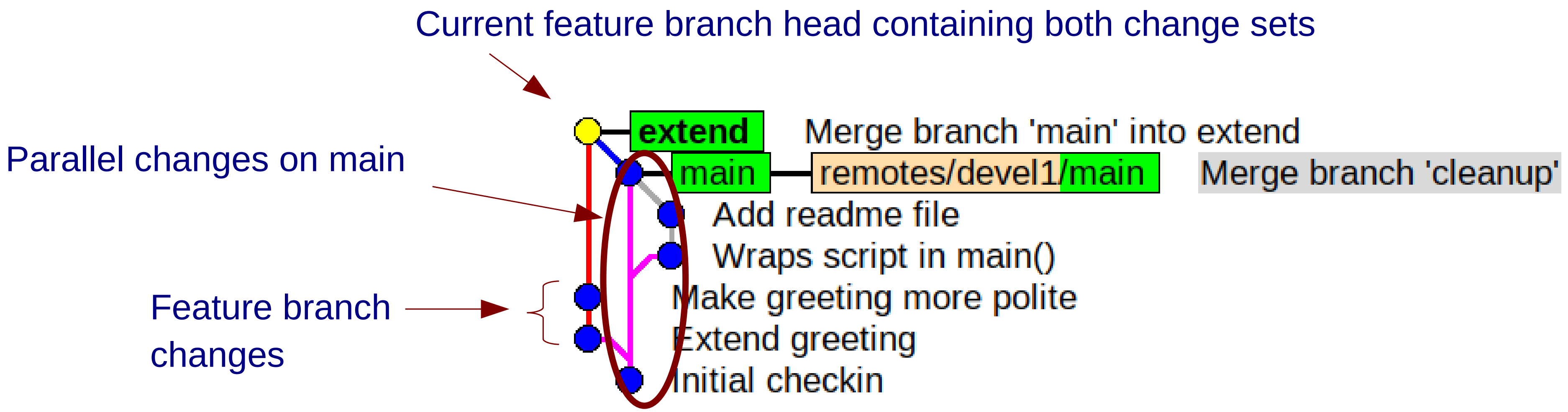
hello.py
(resolved)

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

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

```
git add hello.py
git commit
```

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 devel1/main since the feature branch was created



→ Feature branch is ready to be merged into devel1/main
No conflicts expected (as they had been resolved by Developer 2)

→ Publish feature branch (send repository to Developer 1, push it to GitHub/GitLab

→ Issue **pull request / merge request**:
Ask Developer 1 to pull and merge the updated feature branch into devel1/main

Branch & merge in two repositories (#8)

Developer 1: Fetch and investigate changes from Developer 2

```
git remote add devel2 ../../devel2/hello
git remote -v
```

Register contributors repository (needed only once)

```
devel2  ../../devel2/hello (fetch)
devel2  ../../devel2/hello (push)
```

```
git fetch devel2 extend
```

Fetch content of “extend” branch from devel2’s repository

```
From ../../devel2/hello
* branch            extend     -> FETCH_HEAD
* [new branch]      extend     -> devel2/extend
```

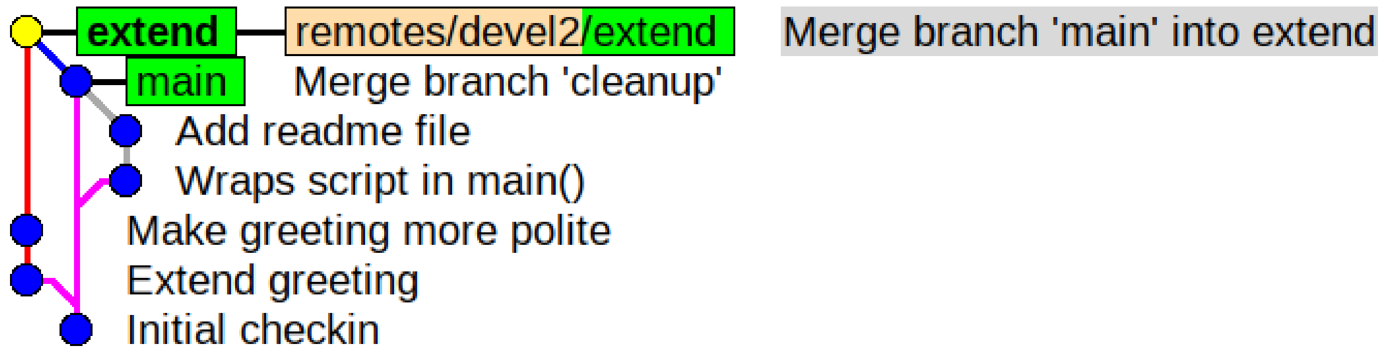
```
git switch extend
```

Check out contrib/extend as extend for further inspection

```
Branch 'extend' set up to track remote branch 'extend' from 'devel2'.
Switched to a new branch 'extend'
```

Branch & merge in two repositories (#9)

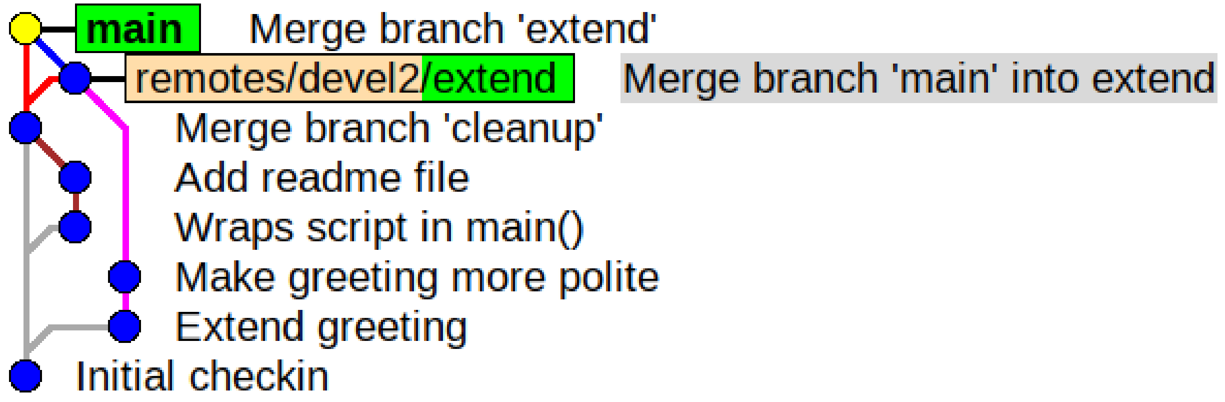
- Developer 1 has an exact local copy of Developer 2's feature branch



```
extend  remotes/devel2/extend  Merge branch 'main' into extend
main  Merge branch 'cleanup'
Add readme file
Wraps script in main()
Make greeting more polite
Extend greeting
Initial checkin
```

Developer 1: merge feature branch into main

```
git switch main
git merge --no-ff extend
git branch -d extend
```



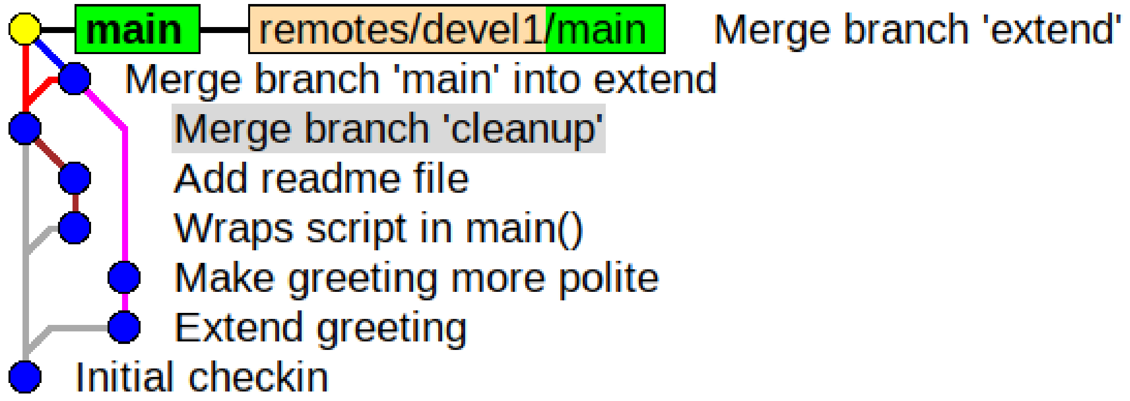
```
main  Merge branch 'extend'
remotes/devel2/extend  Merge branch 'main' into extend
Merge branch 'cleanup'
Add readme file
Wraps script in main()
Make greeting more polite
Extend greeting
Initial checkin
```

Developer 1's main contains all previous commits + changes from contributor

Branch & merge in two repositories (#10)

Developer 2: sync main branch with Developer 1’s main

```
git switch main
git pull --ff-only devel1 main
git branch -d extend
```



```
main  remotes/devel1/main  Merge branch 'extend'
Merge branch 'main' into extend
Merge branch 'cleanup'
Add readme file
Wraps script in main()
Make greeting more polite
Extend greeting
Initial checkin
```

Developer 2’s main branch indentical to devel1/main!

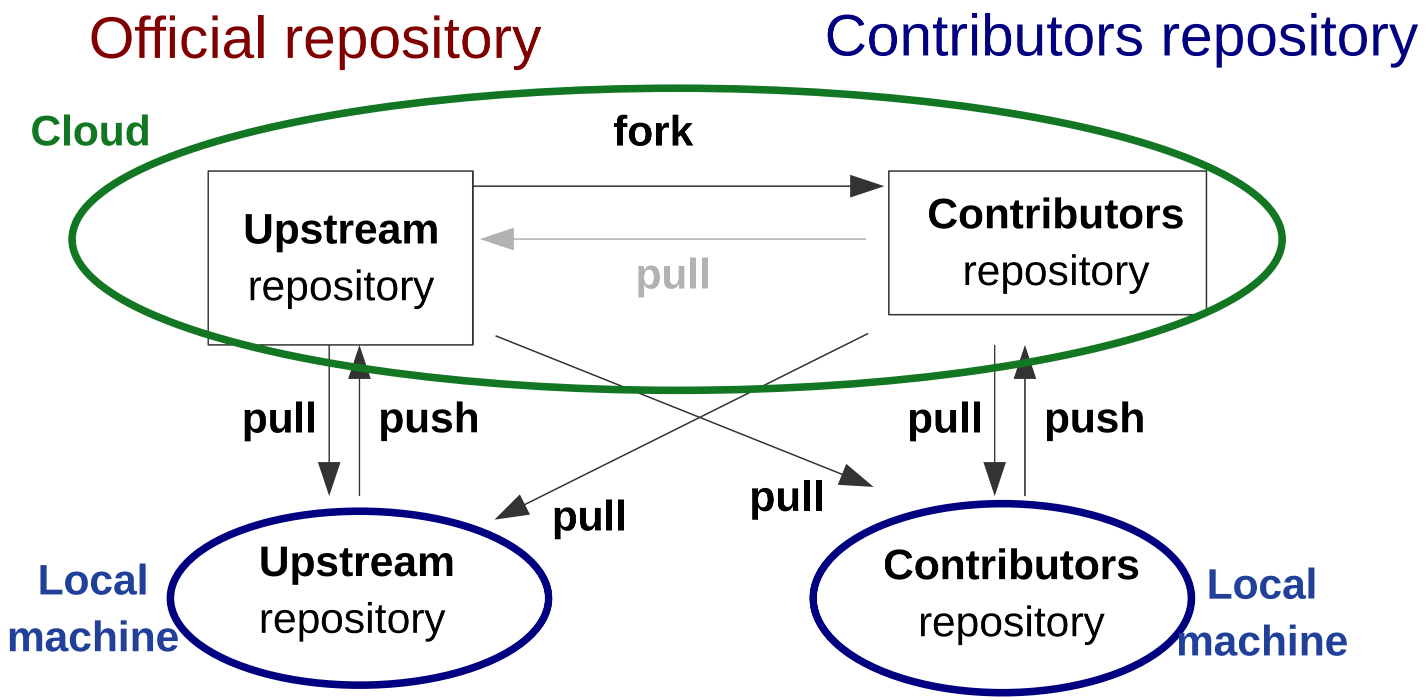
Publishing a repository

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
- 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 git notes

- Git is very flexible and powerful, allowing for almost **arbitrary workflows**
  → Most open source projects 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.)

Have fun!