Introduction to Version Control with Git

Gert-Ludwig Ingold

https://github.com/gertingold/euroscipy-git-tutorial.git

Do you write 100% bugfree code with all features implemented from the very beginning?

yes: 0 points / no: 1 point

Do you write 100% bugfree code with all features implemented from the very beginning?

yes: 0 points / no: 1 point

•

Do you collaborate with others?

yes: 1 point / no: 0 points

Do you write 100% bugfree code with all features implemented from the very beginning?

yes: 0 points / no: 1 point

•

Do you collaborate with others?

yes: 1 point / no: 0 points

•

Do you want to contribute to open software?

yes: 1 point / no: 0 points

Do you write 100% bugfree code with all features implemented from the very beginning?

yes: 0 points / no: 1 point

•

Do you collaborate with others?

yes: 1 point / no: 0 points

•

Do you want to contribute to open software?

yes: 1 point / no: 0 points

One or more points: Version control is for you!

```
myscript.py
myscript.py.bak
myscript.py.bak.bak
myscript.py.bak.bak.bak
myscript.py.bak2
myscript.py.bak3
myscript.py.bak4
myscript.py.old
myscript.py.superold
myscript.py.workedonce
paper.tex
paper-20170801.tex
paper-20170725-ab-20170727.tex
paper-20170720-ab-20170721-xy-20170722-ab.tex
paper.tex.old
paper.tex.veryold
paper.tex.firsttry
```

```
myscript.py
myscript.py.bak
myscript.py.bak.bak
myscript.py.bak.bak.bak
myscript.py.bak2
myscript.py.bak3
myscript.py.bak4
myscript.py.old
myscript.py.superold
myscript.py.workedonce
paper.tex
paper-20170801.tex
paper-20170725-ab-20170727.tex
paper-20170720-ab-20170721-xy-20170722-ab.tex
paper.tex.old
paper.tex.veryold
paper.tex.firsttry
```

A big mess. So sad.

```
myscript.py
myscript.py.bak
myscript.py.bak.bak
myscript.py.bak.bak.bak
myscript.py.bak2
myscript.py.bak3
myscript.py.bak4
myscript.py.old
myscript.py.superold
myscript.py.workedonce
paper.tex
paper-20170801.tex
paper-20170725-ab-20170727.tex
paper-20170720-ab-20170721-xy-20170722-ab.tex
paper.tex.old
paper.tex.veryold
paper.tex.firsttry
```

A big mess. So sad.

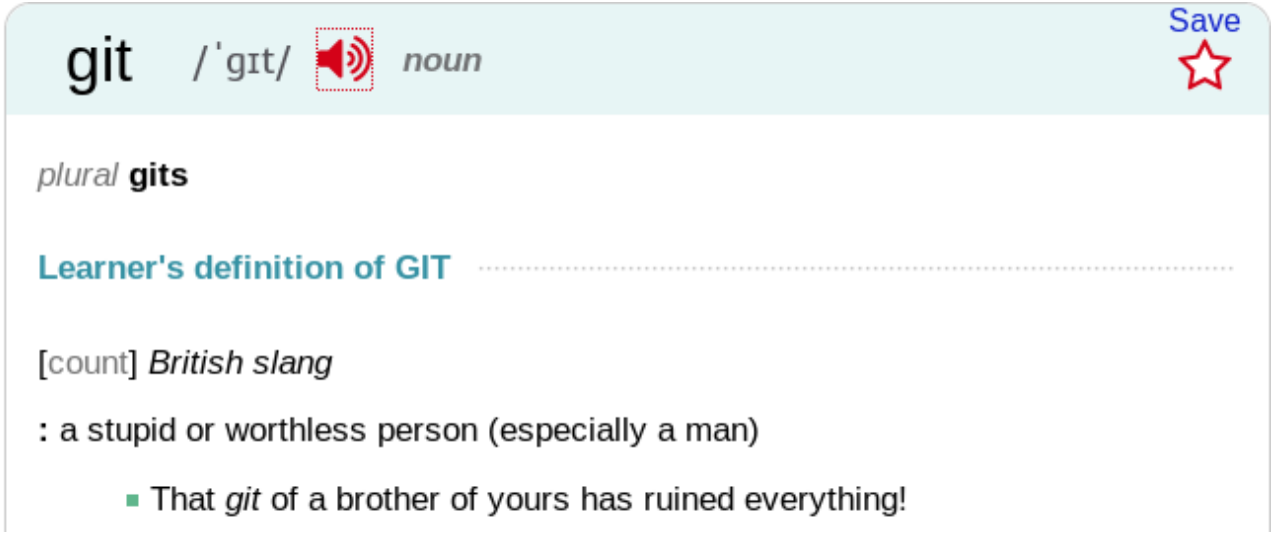
We will use a great VCS. And we will not pay for it.
It will be fantastic.

A short history of version control

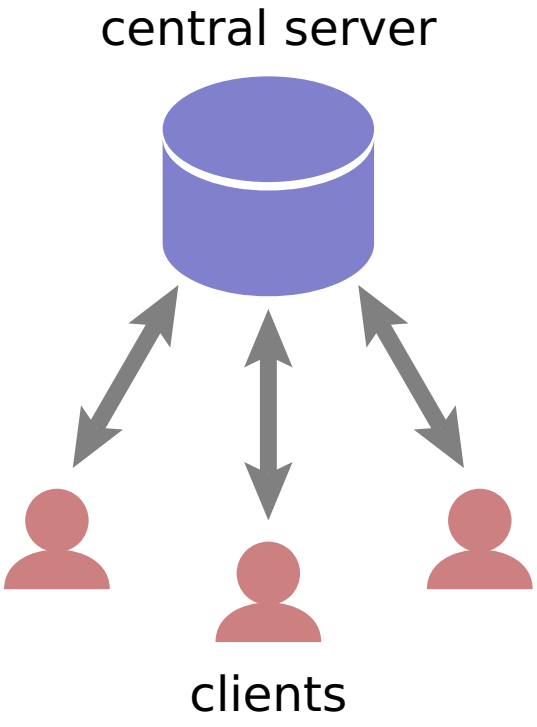
- *SCCS – Source Code Control System (1972)*
- *RCS – Revision Control System (1982)*
  single file oriented, locking mechanism
- *CVS – Concurrent Versions System (1990)*
  Subversion (2000)
  centralized version control system
- *Git, Mercurial, Bazaar (2005)*
  distributed version control systems

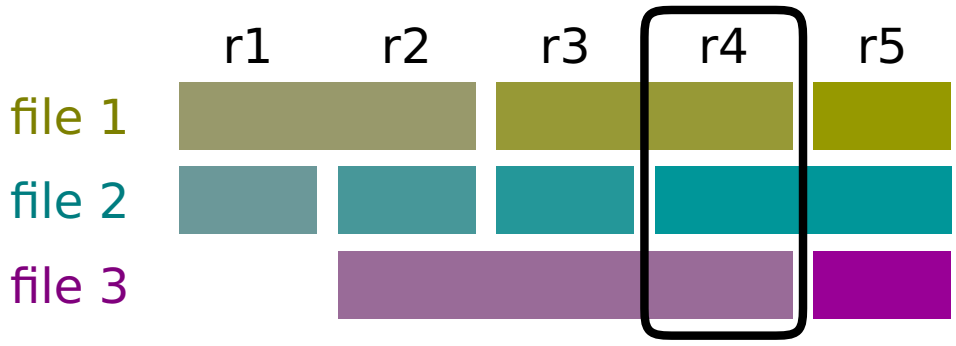
A short history of version control

- *SCCS – Source Code Control System (1972)*
- *RCS – Revision Control System (1982)*
  single file oriented, locking mechanism
- *CVS – Concurrent Versions System (1990)*
  Subversion (2000)
  centralized version control system
- *Git, Mercurial, Bazaar (2005)*
  distributed version control systems



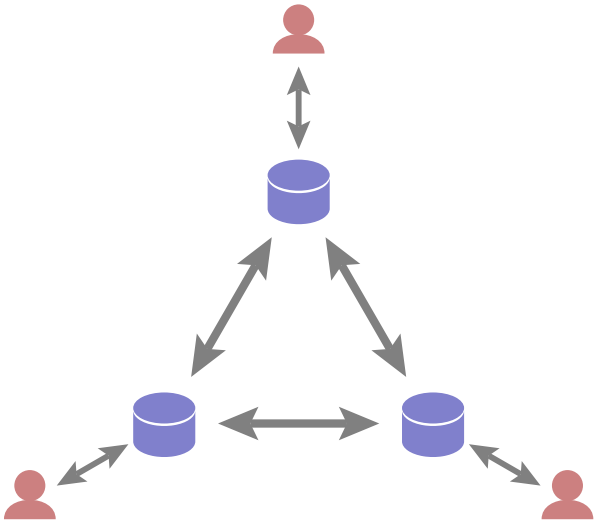
Centralized version control systems

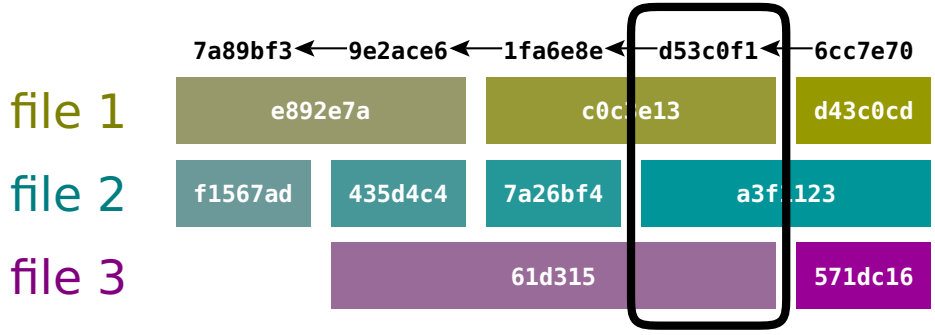




At any time, the central server contains well defined revisions of file sets which can be consecutively numbered.

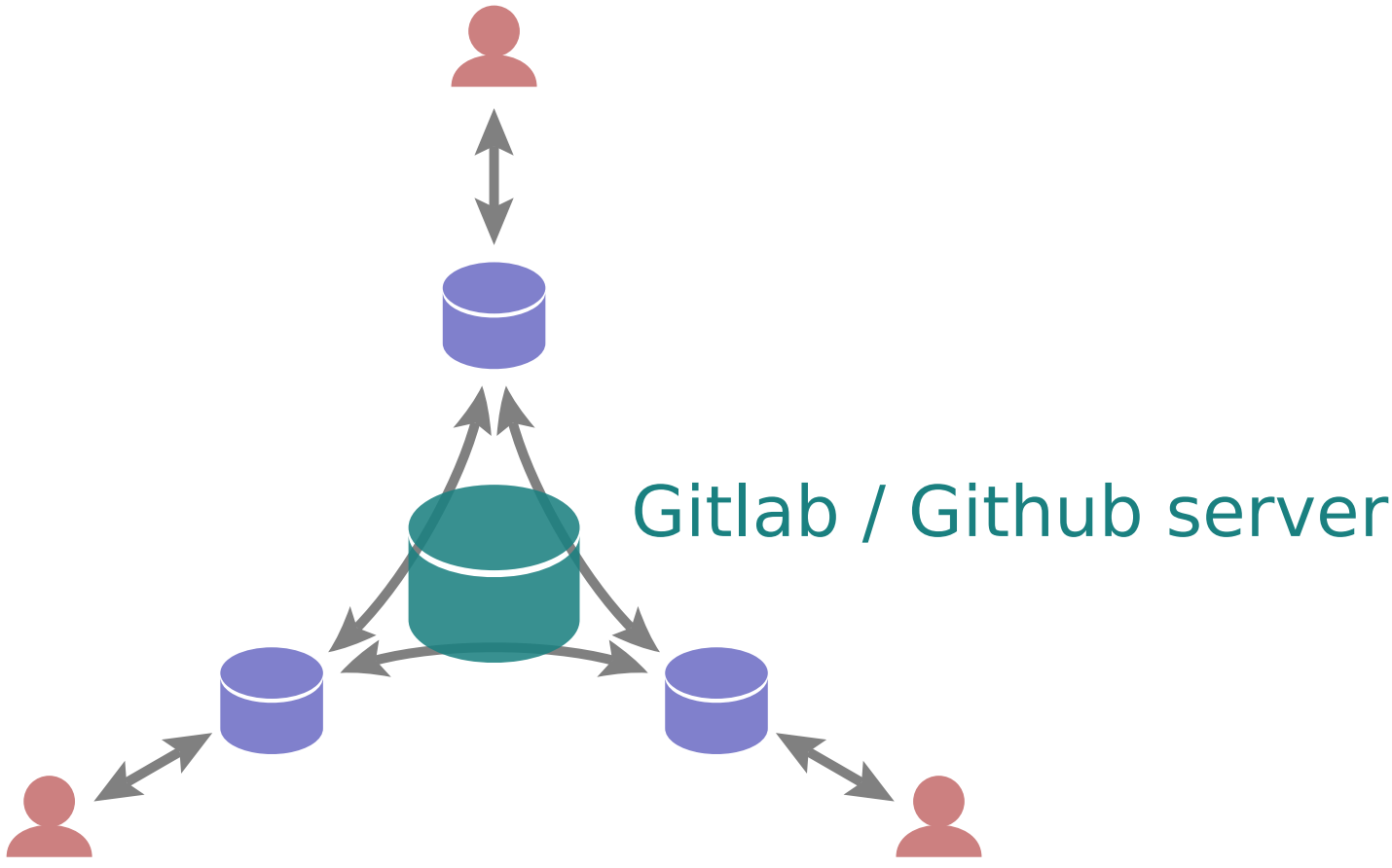
Distributed version control systems





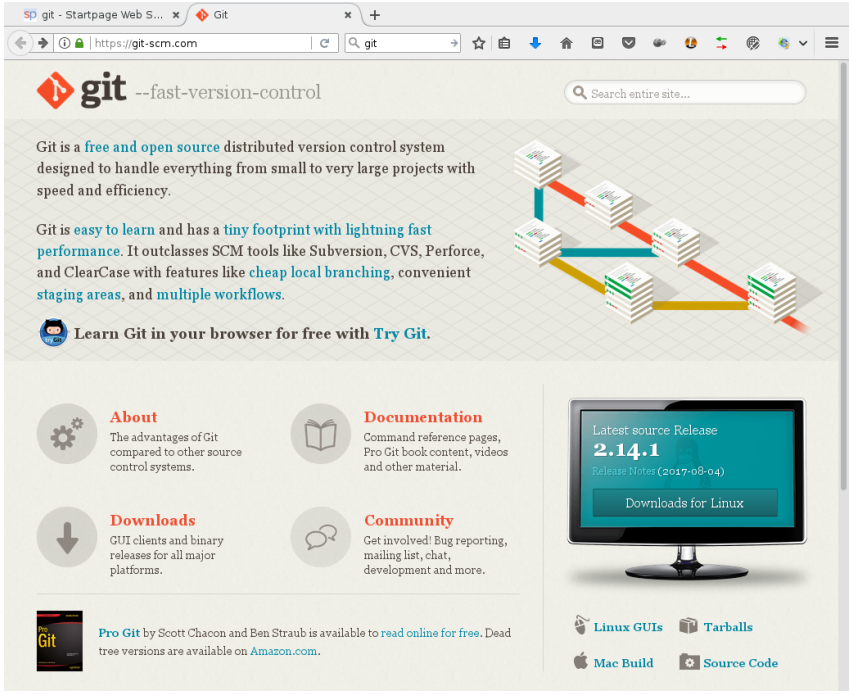
- each individual repository has its own history
- each object is identified by a SHA1 hash consisting of 40 hexadecimal values
- there are more than 10^{48} different SHA1 hashes
- often the first seven hex digits are sufficient for identification

Distributed VCS with Gitlab / Github



Where to get the software and more information

https://git-scm.com/



- binaries for Windows and Mac OS X, install instructions for Linux and Solaris
- reference documentation
- online version of the *Pro Git* book by S. Chacon and B. Straub including several translations
- some instructional videos
- and more . . .

How to get help: `git help`

```
$ git help
These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   branch     List, create, or delete branches
   checkout   Switch branches or restore working tree files
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects
```

How to get help on a specific command

`git help <command>`

```
$ git help add
NAME
       git-add - Add file contents to the index

SYNOPSIS
       git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i]
                 [--patch | -p] [--edit | -e] [--[no-]all | --[no-]ignore-removal |
                 [--update | -u]] [--intent-to-add | -N] [--refresh] [--ignore-errors]
                 [--ignore-missing] [--chmod=(+|-)x] [--] [<pathspec>...]

DESCRIPTION

       This command updates the index using the current content found in the working
       tree, to prepare the content staged for the next commit. It typically adds
       the current content of existing paths as a whole, but with some options it
       can also be used to add content with only part of the changes made to the
       working tree files applied, or remove paths that do not exist in the working
       tree anymore.

...
```

Accessing Git tutorials

```
$ git help -g
The common Git guides are:

   attributes   Defining attributes per path
   everyday     Everyday Git With 20 Commands Or So
   glossary     A Git glossary
   ignore       Specifies intentionally untracked files to ignore
   modules      Defining submodule properties
   revisions    Specifying revisions and ranges for Git
   tutorial     A tutorial introduction to Git (for version 1.5.1 or newer)
   workflows    An overview of recommended workflows with Git

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

```
$git help everyday

NAME
       giteveryday - A useful minimum set of commands for Everyday Git

SYNOPSIS
       Everyday Git With 20 Commands Or So

DESCRIPTION
       Git users can broadly be grouped into four categories for the purposes of
       describing here a small set of useful command for everyday Git.

       ·   Individual Developer (Standalone) commands are essential for anybody who
           makes a commit, even for somebody who works alone.

       ·   If you work with other people, you will need commands listed in the
           Individual Developer (Participant) section as well.
...
```

The prime time project

Question

How many prime numbers can be interpreted as time in the format HH:MM?

Examples

2179 is a prime, but 21:79 is not a valid time

2137 is a prime and 21:37 is a valid time

953 is a prime and 9:53 is a valid time

89 is a prime, but 0:89 is not a valid time

41 is a prime and 0:41 is a valid time

7 is a prime and 0:07 is a valid time

The prime time project

Question

How many prime numbers can be interpreted as time in the format HH:MM?

Examples

2179 is a prime, but 21:79 is not a valid time
2137 is a prime and 21:37 is a valid time
953 is a prime and 9:53 is a valid time
89 is a prime, but 0:89 is not a valid time
41 is a prime and 0:41 is a valid time
7 is a prime and 0:07 is a valid time

And, of course, we are going to use a Git repository.

Creating a respository

Initializing a new repository

```
~/primetime$ git init
```

What has happened?

```
~/primetime$ ls -a
.  ..  .git
~/primetime$ ls .git
branches  description  hooks  objects
config    HEAD         info   refs
```

Creating a respository

Initializing a new repository

```
~/primetime$ git init
```

What has happened?

```
~/primetime$ ls -a
.  ..  .git
~/primetime$ ls .git
branches  description  hooks  objects
config    HEAD         info   refs
```

Keep your hands off the `.git` directory!!!

Tell Git who you are

Specify your name and your email address

```
$ git config --global user.name <your name>
$ git config --global user.email <your email>
```

. . . and, if you want, your preferred editor

```
$ git config --global core.editor <editor>
```

example configuration

```
$ git config --list
user.email=gert.ingold@physik.uni-augsburg.de
user.name=Gert-Ludwig Ingold
core.editor=vim
...
```

It’s prime time now

primetime.py

```
def istime(n):
    hh, mm = divmod(n, 100)
    return 0 <= hh <= 23 and 0 <= mm <= 59

print(sum(istime(n) for n in range(2360)))
```

What is the status?

```
~/primetime$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        primetime.py

nothing added to commit but untracked files present (use "git
    ↪ add" to track)
```

- Git has noticed our new file but ignores it
- Git tells us how to put the file under version control

Adding a file

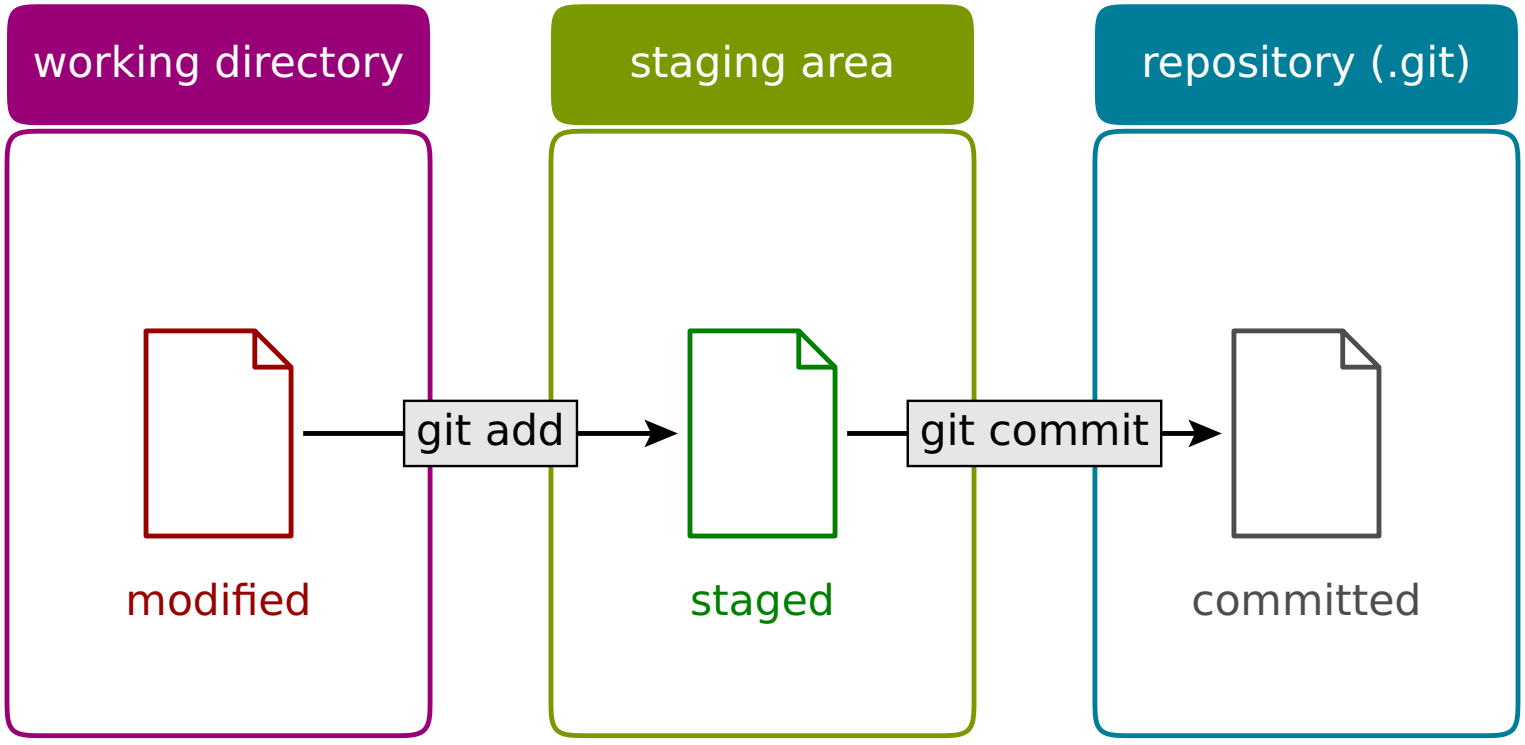
```
~/primetime$ git add primetime.py
```

How did the status change?

```
~/primetime$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   primetime.py
```

- Our file is now in the staging area, it is not under version control yet!
- Our file can now be committed, but we can add more files first
- Git tells us how we can remove the file from the staging area if we put it there by accident

The way into the repository



- All files present in the staging area are committed
- At a given time, different versions of a specific file can exist in any of the three areas

Committing a file

```
~/primetime$ git commit -m’added function to identify times’
[master (root-commit) 3733ef5] added function to identify times
 1 file changed, 5 insertions(+)
 create mode 100644 primetime.py
```

- The option `-m` allows to specify a commit message
- Without this option, an editor is opened
- Limit the commit message to 50 characters

Check status to make sure that everything is fine

```
~/primetime$ git status
On branch master
nothing to commit, working tree clean
```

Some tips on committing

- A commit can contain several files, but all changes should represent a logical unit
- *atomic commit:* Each commit refers only to a single basic change
- In most cases it is not a good idea to commit only at the end of a long working day
- If you have made unrelated changes but want to do an atomic commit, take a look at
  `git add -p`

A first complete version of primetime

`primes.py` (very inefficient)

```
def isprime(n):
    if n < 2: return False
    for divisor in range(2, n):
        if not n % divisor:
            return False
    return True
```

`primetime.py`

```
from primes import isprime

def istime(n):
    hh, mm = divmod(n, 100)
    return 0 <= hh <= 23 and 0 <= mm <= 59

print(sum(isprime(n) for n in range(2360) if istime(n)))
```

It works:

```
~/primetime$ python primetime.py
211
```

The new status

```
~/primetime$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
      ↪ directory)

        modified:   primetime.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        __pycache__/
        primes.py

no changes added to commit (use "git add" and/or "git commit -a")
```

- Git noticed that `primetime.py` has been modified
- `primes.py` is new and not yet tracked by Git
- The directory `__pycache__/` should not be under version control
- Note also the help given by Git

The `.gitignore` file

- List in `.gitignore` line-by-line all directories and files to be ignored by Git
- * can be used as a wildcard
- Everything after a # is a comment
- Put `.gitignore` under version control
- `.gitignore` can help to make sure that passwords never are put under version control

In our example:
`.gitignore`

```
__pycache__/
```

Add and commit

```
~/primetime$ git add .gitignore
~/primetime$ git commit -m’.gitignore added’
[master c552c10] .gitignore added
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

```
~/primetime$ git add primetime.py
~/primetime$ git add primes.py
~/primetime$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   primes.py
        modified:   primetime.py

~/primetime$ git commit -m’first version of primetime completed’
[master 116fd26] first version of primetime completed
 2 files changed, 9 insertions(+), 1 deletion(-)
 create mode 100644 primes.py
```

Branches

```
~/primetime$ git status
On branch master
nothing to commit, working tree clean
```

- We are on branch `master`, the default branch

```
gert@gli7440:~/primetime$ git branch
* master
```

What are branches good for?

- Branches allow development of features without polluting the master branch
- Tip: Use a separate branch for each feature
- Unsuccessful experiments can easily be removed by deleting a branch
- In collaborative work, a pull request (see later) can easily focus on the new code
- Code can be merged into other branches

Adding a new branch

In our prime time project, we are unhappy with our code for prime tests. Instead, we want to create a list of primes by means of the sieve of Eratosthenes. We develop the new feature in a separate branch so that the version in the master branch is always working.

```
~/primetime$ git checkout eratosthenes
error: pathspec 'eratosthenes' did not match any file(s) known
    ↪ to git.
```

- `git checkout` changes between branches, but our branch does not yet exist. Use option `-b`

```
gert@gli7440:~/primetime$ git checkout -b eratosthenes
Switched to a new branch 'eratosthenes'
```

Let us verify the branch

```
gert@gli7440:~/primetime$ git branch
* eratosthenes
  master
```

Improved primetime code (I)

primes.py

```
from math import sqrt
import numpy as np

def isprime(n):
    if n < 2: return False
    for divisor in range(2, n):
        if not n % divisor:
            return False
    return True

def eratosthenes(nmax):
    sieve = np.ones(nmax+1, dtype=np.bool)
    sieve[:2] = False
    for candidate in range(2, int(sqrt(nmax))+1):
        if sieve[candidate]:
            sieve[candidate*candidate::candidate] = False
    primes = np.arange(nmax+1)[sieve]
    return primes
```

```
~/primetime$ git commit -a -m'added sieve of Eratosthenes'
[eratosthenes 982ae11] added sieve of Eratosthenes
 1 file changed, 12 insertions(+)
```

Improved primetime code (II)

primetime.py

```
from primes import eratosthenes

def istime(n):
    hh, mm = divmod(n, 100)
    return 0 <= hh <= 23 and 0 <= mm <= 59

print(sum(istime(n) for n in eratosthenes(2359)))
```

```
~/primetime$ git commit -a -m’make use of Eratosthenes in primetime’
[eratosthenes c9dfe03] make use of Eratosthenes in primetime
 1 file changed, 2 insertions(+), 2 deletions(-)
```

Working on branches in parallel

In the meantime, an idea comes up to improve the function `isprime`. We make the corresponding changes in the master branch.
First switch to the master branch:

```
~/primetime$ git checkout master
Switched to branch 'master'
```

Here, `primes.py` does not contain the function `eratosthenes`. We replace the code by

```
from math import sqrt

def isprime(n):
    if n < 2: return False
    for divisor in range(2, int(sqrt(n))+1):
        if not n % divisor:
            return False
    return True
```

```
~/primetime$ git commit -a -m'improved prime test'
[master ebc424d] improved prime test
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Commits in different branches

```
~/primetime$ git log  --graph --branches --oneline
* ebc424d improved prime test
| * c9dfe03 make use of Eratosthenes in primetime
| * 982ae11 added sieve of Eratosthenes
|/
* 116fd26 first version of primetime completed
* c552c10 .gitignore added
* 3733ef5 added function to identify times
```

Merging the content of the `eratosthenes` branch into the `master` branch

```
~/primetime$ git branch
  eratosthenes
  * master
  ~/primetime$ git merge eratosthenes
  Auto-merging primes.py
  CONFLICT (content): Merge conflict in primes.py
  Automatic merge failed; fix conflicts and then commit the result.
```

- `primetime.py` corresponds now to the `eratosthenes` branch
- In `primes.py` a merge conflict needs to be resolved

A merge conflict

```
from math import sqrt
<<<<<<< HEAD
=======
import numpy as np
>>>>>>> eratosthenes

def isprime(n):
    if n < 2: return False
    for divisor in range(2, int(sqrt(n))+1):
        if not n % divisor:
            return False
    return True

def eratosthenes(nmax):
...
```

- Alternative 1 between >>>>>>> and =======
  HEAD = version being merged into
- Alternative 2 between ======= and <<<<<<<
  `eratosthenes` = version being merged
- Bring the file into the desired form and commit it

Resolving a merge conflict

new version of `primes.py`

```
from math import sqrt
import numpy as np

def isprime(n):
    if n < 2: return False
...
```

```
~/primetime$ git commit -a
[master 5782233] Merge branch 'eratosthenes'
```

```
~/primetime$ git log --graph --branches --oneline
*   5782233 Merge branch 'eratosthenes'
|\
| * c9dfe03 make use of Eratosthenes in primetime
| * 982ae11 added sieve of Eratosthenes
* | ebc424d improved prime test
|/
* 116fd26 first version of primetime completed
* c552c10 .gitignore added
* 3733ef5 added function to identify times
```

Deleting a branch

```
~/primetime$ git branch -D eratosthenes
Deleted branch eratosthenes (was c9dfe03).
```

- option D must be uppercase = Attention, danger!

```
~/primetime$ git branch
* master
```

- The old versions still exist

```
~/primetime$ git log --graph --branches --oneline
*   5782233 Merge branch 'eratosthenes'
|\
| * c9dfe03 make use of Eratosthenes in primetime
| * 982ae11 added sieve of Eratosthenes
* | ebc424d improved prime test
|/
* 116fd26 first version of primetime completed
* c552c10 .gitignore added
* 3733ef5 added function to identify times
```

Accessing a specific version

```
~/primetime$ git log --oneline --graph --branches --decorate
*   5782233 (HEAD -> master) Merge branch 'eratosthenes'
|\
| * c9dfe03 make use of Eratosthenes in primetime
| * 982ae11 added sieve of Eratosthenes
* | ebc424d improved prime test
|/
* 116fd26 first version of primetime completed
* c552c10 .gitignore added
* 3733ef5 added function to identify times
```

- absolute reference: HEAD or SHA1 hash
- relative reference:

parent	HEAD^	ebc424d
parent	c552c10^	3733ef5
2nd parent	HEAD^2	c9dfe03
grandparent	HEAD^^	116fd26
grandparent	HEAD~2	116fd26

Detached HEAD

```
~/primetime$ git checkout 982ae11
Note: checking out ’982ae11’.

You are in ’detached HEAD’ state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 982ae11... added sieve of Eratosthenes
```

```
~/primetime$ git log --oneline --graph --branches --decorate
*   5782233 (master) Merge branch ’eratosthenes’
|\
| * c9dfe03 make use of Eratosthenes in primetime
| * 982ae11 (HEAD) added sieve of Eratosthenes
* | ebc424d improved prime test
|/
* 116fd26 first version of primetime completed
* c552c10 .gitignore added
* 3733ef5 added function to identify times
```

- HEAD is not attached to a branch, it is detached
- In order to commit something, a branch needs to be created

Attaching HEAD to a branch

```
~/primetime$ git branch
* (HEAD detached at 982ae11)
  master
```

```
~/primetime$ git checkout -b experimental
Switched to a new branch 'experimental'
~/primetime$ git branch -v
* experimental 982ae11 added sieve of Eratosthenes
  master       5782233 Merge branch 'eratosthenes'
```

- Now we have a branch at 982ae11 on which we can work and commit changes

Tagging versions

Sometimes it is useful to mark a specific version, e.g. the submitted version of a paper.

Tagging the present version

```
~/primetime$ git tag -a first_final -m'a first reasonably
    ↪ complete version'
```

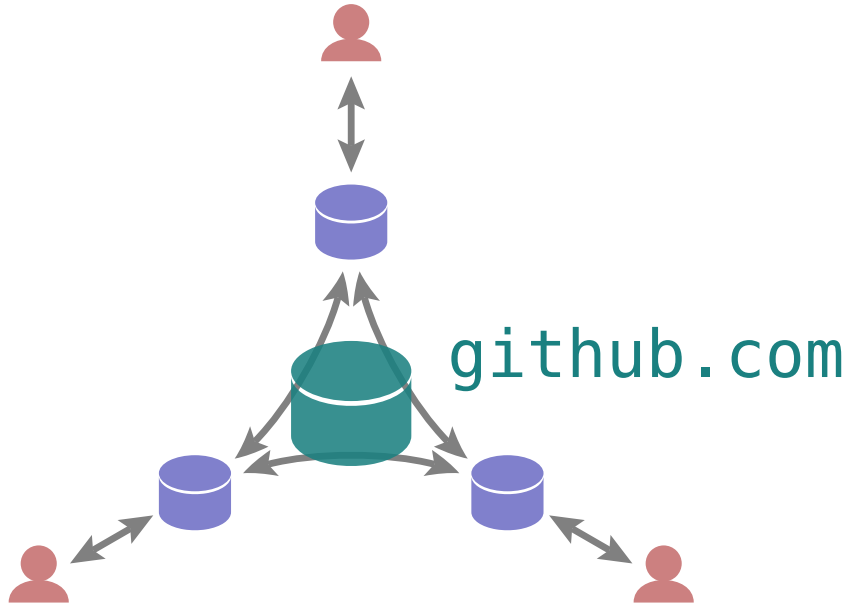
Tagging an older version

```
~/primetime$ git tag -a v0.1 116fd26 -m'early version'
```

```
~/primetime$ git tag
first_final
v0.1
```

```
~/primetime$ git show v0.1
tag v0.1
Tagger: Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
Date:   Mon Aug 14 01:12:07 2017 +0200

early version

commit 116fd26220dddb7ec3eb359d3c35ea9c312bb49c
Author: Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
Date:   Sun Aug 13 16:16:46 2017 +0200

    first version of primetime completed

...
```

Working with Github



What is Github?

- Github hosts Git projects and facilitates collaborative development by means of Git
- *Public* repositories readable for everybody are available free of charge
- *Private* repositories are not free, but may be free for academic use on request
- Github is very popular for open source projects

Cloning a repository

Scenario: You want to get the content of a repository without contributing to its development.

```
$ git clone https://github.com/gertingold/euroscipy-git-tutorial.git
Cloning into 'euroscipy-git-tutorial'...
remote: Counting objects: 205, done.
remote: Compressing objects: 100% (30/30), done.
remote: Total 205 (delta 40), reused 61 (delta 37), pack-reused 138
Receiving objects: 100% (205/205), 354.93 KiB | 0 bytes/s, done.
Resolving deltas: 100% (121/121), done.
Checking connectivity... done.
```

```
$ ls euroscipy-git-tutorial/
images  LICENSE  presentation.tex  README.md
```

or if you just want the current version

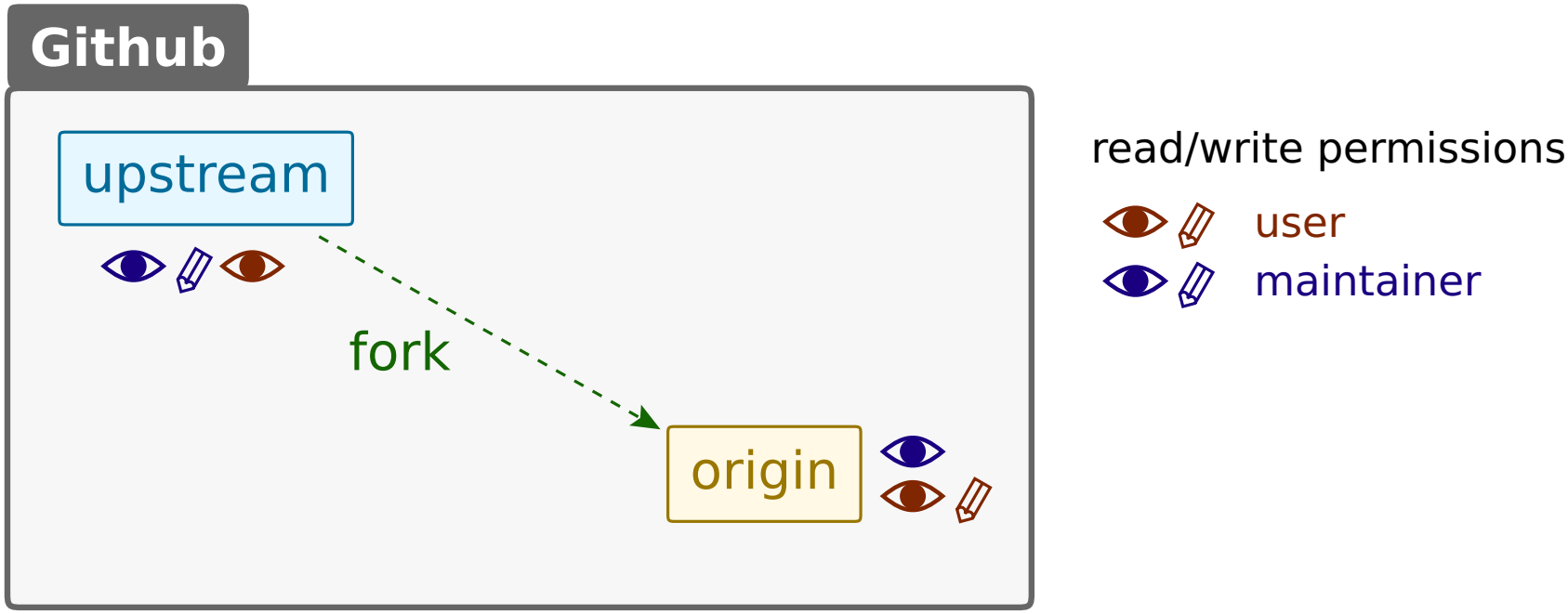
```
$ git clone --depth=1 https://github.com/gertingold/euroscipy-git-tutorial.git
Cloning into 'euroscipy-git-tutorial'...
remote: Counting objects: 23, done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 23 (delta 3), reused 14 (delta 0), pack-reused 0
Unpacking objects: 100% (23/23), done.
Checking connectivity... done.
$ ls euroscipy-git-tutorial/
images  LICENSE  presentation.tex  README.md
```

Typical workflow for project development

Scenario: You want to contribute to a project, but only the maintainers can make changes to the repository.

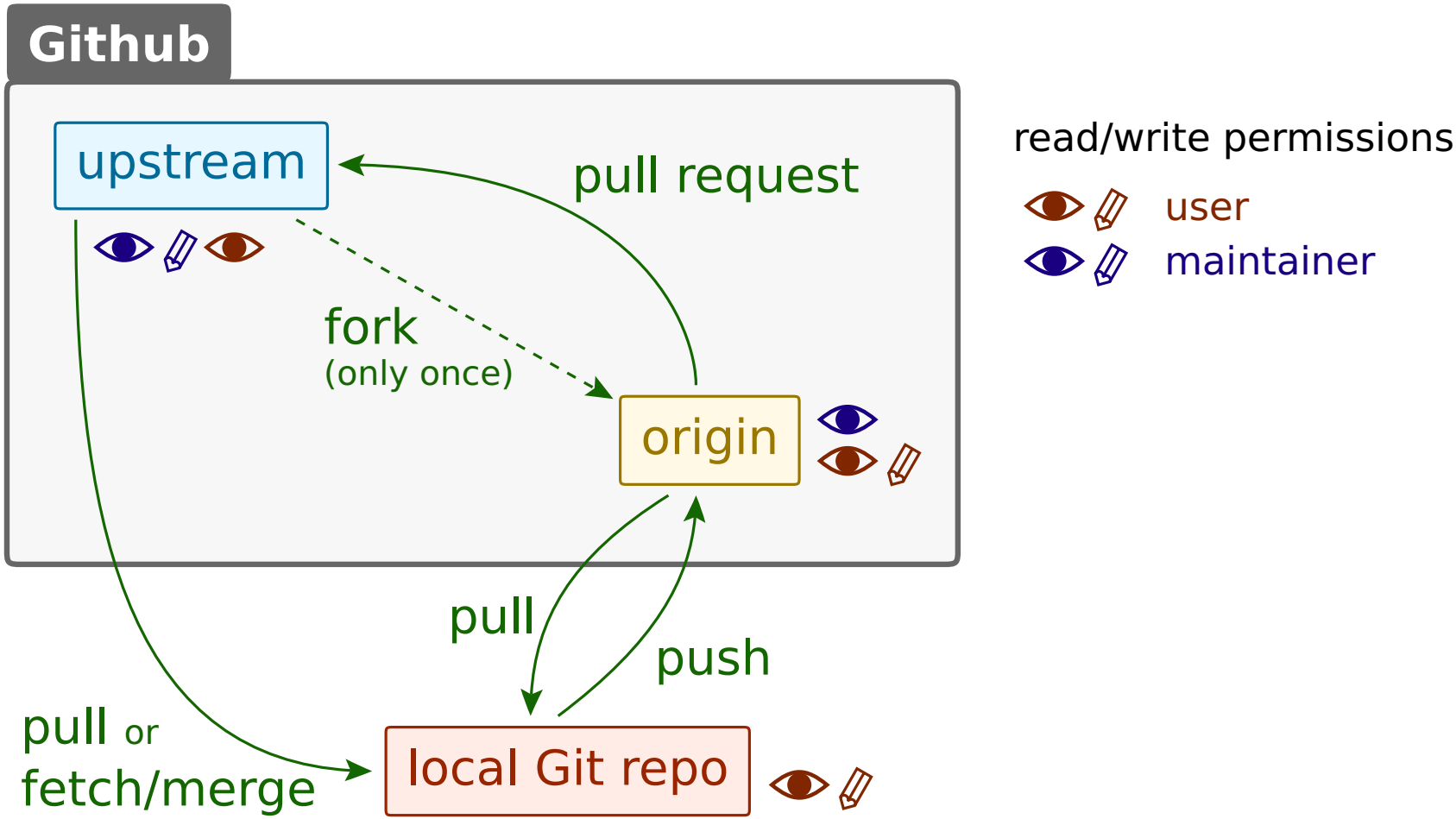
Typical workflow for project development

Scenario: You want to contribute to a project, but only the maintainers can make changes to the repository.

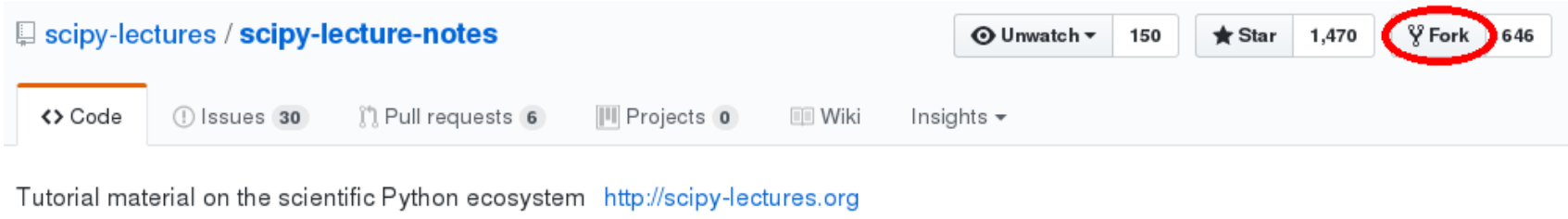


Typical workflow for project development

Scenario: You want to contribute to a project, but only the maintainers can make changes to the repository.



Remote branches



clone your forked project

```
$ git clone git@github.com:gertingold/scipy-lecture-notes
...
$ cd scipy-lecture-notes
scipy-lecture-notes$ git remote -v
origin  git@github.com:gertingold/scipy-lecture-notes.git (fetch)
origin  git@github.com:gertingold/scipy-lecture-notes.git (push)
```

tell Git about the upstream repository

```
gert@teide:[...]/scipy-lecture-notes: git remote add upstream git@github.com:scipy-
     ↪ lecture-notes/scipy-lecture-notes.git
gert@teide:[...]/scipy-lecture-notes: git remote -v
origin  git@github.com:gertingold/scipy-lecture-notes.git (fetch)
origin  git@github.com:gertingold/scipy-lecture-notes.git (push)
upstream        git@github.com:scipy-lecture-notes/scipy-lecture-notes.git (fetch)
upstream        git@github.com:scipy-lecture-notes/scipy-lecture-notes.git (push)
```

Updating from the upstream repository

fetch and merge

```
git fetch upstream
```

- download objects and refs (branches and tags) from the upstream repository
- the current working copy remains unchanged → opportunity to check the changes

```
git merge upstream/master
```

- merge the fetched objects from `upstream/master` into the current working copy

pull

```
git pull upstream
```

- fetch and merge in one step
- no control over merged changes

Updating the origin repository

push

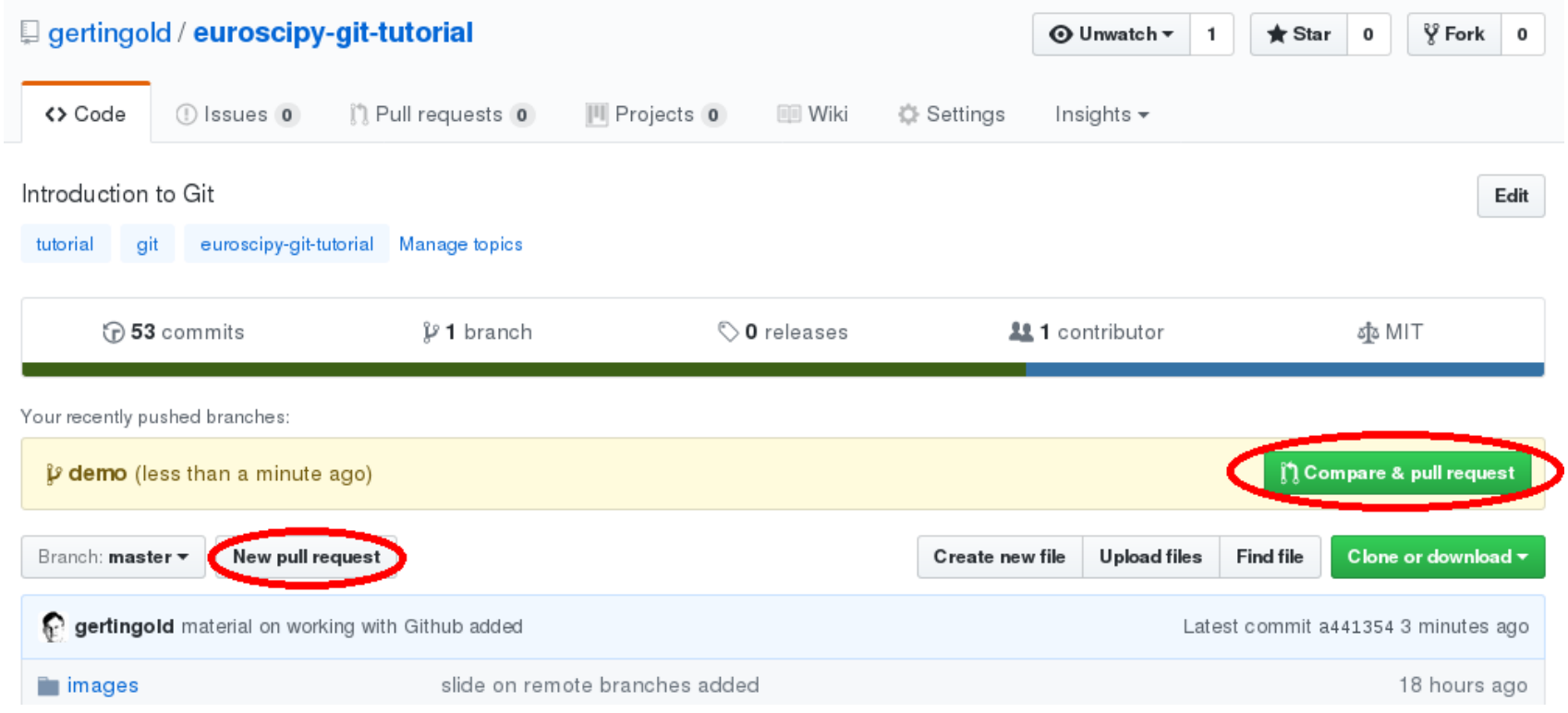
```
git push origin
```

- `origin` can only brought up-to-date with `upstream` via the local repository
- push new material to `origin` so that it can be made available to `upstream` by means of a pull request

Making a pull request I

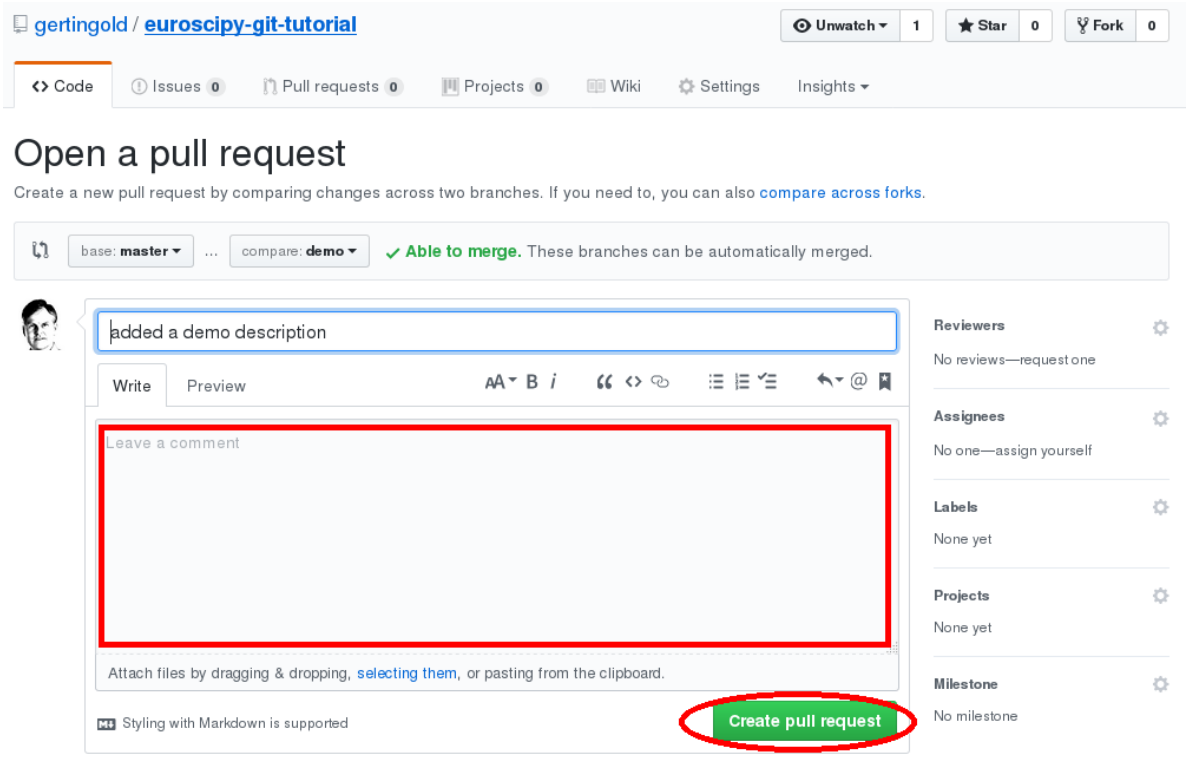
scenario: You want to have your new material included in the `upstream` repository.

1. develop the new material in a separate branch
2. push the new material to `origin`
3. make a pull request (= merge request on Gitlab)



Making a pull request II

4. describe the purpose of your pull request
5. create the pull request



- new commits to the feature branch will be added to the pull request once they are pushed to `origin`