International Institute for
Applied Systems Analysis
IIASA www.iiasa.ac.at

TU WIEN

Lecture 2:
Hands-on example of working with git

Open-Source Energy System Modeling
TU Wien, VU 370.062

Dipl.-Ing. Dr. Daniel Huppmann

Part 1

Working with `git` version control

A quick introduction to version control using `git`

Git is so much more than just keeping track of code changes over time

Key differences between `git` version control vs. folder synchronization (e.g. Dropbox, Google Drive)

⇒ You define the relevant unit or size of a change by making a ***commit***

⇒ Adding comments to your commits allows to attach relevant info to your code changes

⇒ *Branches* allow you to switch to a "parallel universe" within a version control repository

⇒ It’s a decentralized version control tool that supports offline, parallel work

⇒ There is a well-defined routine for ***merging*** developments from parallel branches

Several `git` implementations (e.g., GitHub) provide additional project management tools

⇒ User interfaces for code review using ***pull requests***

⇒ Issue tracking and discussion, kanban boards, ...

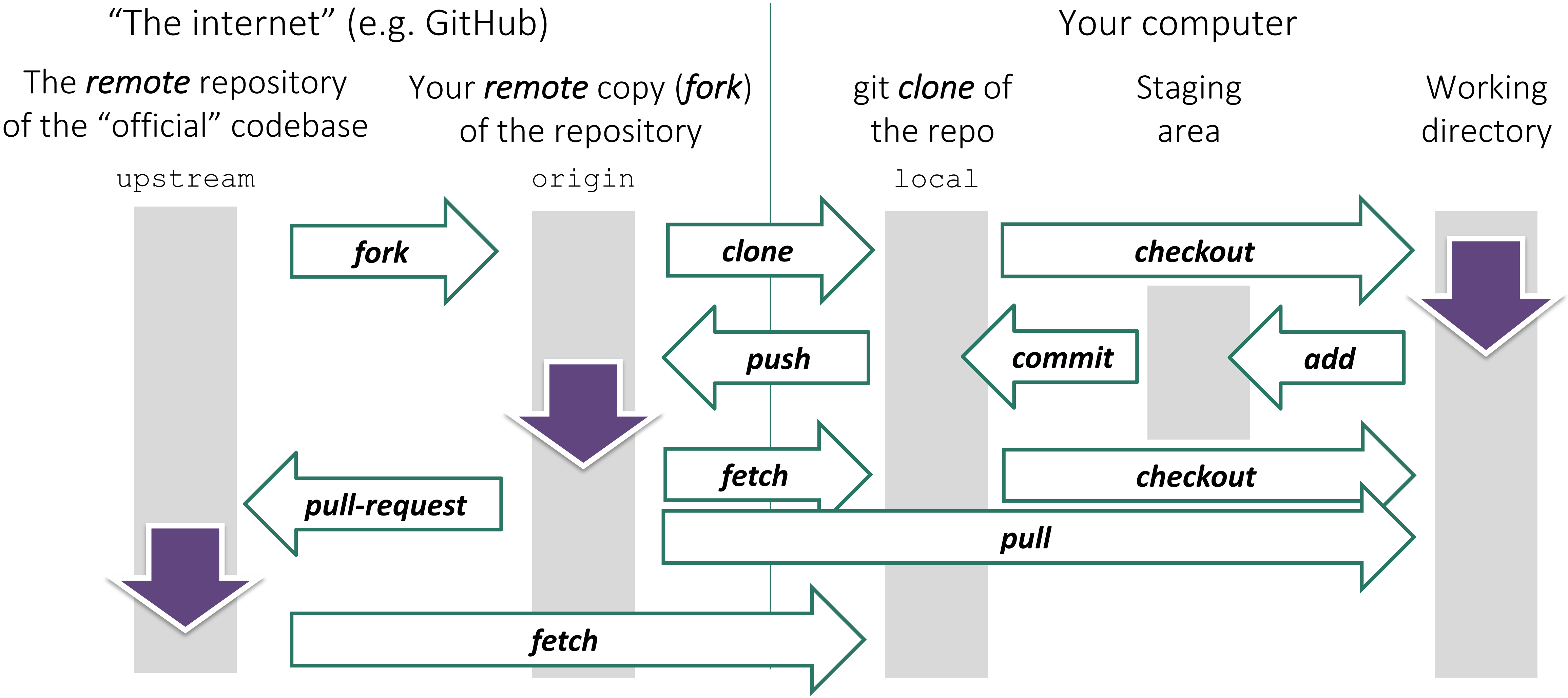
GitHub

However, keep in mind that `git` is great for uncompiled code and text with simple mark-up

⇒ Use other version control tools for data, presentations, compiled software, ...

A full `git` workflow

Git is a decentralized version control system geared for collaboration

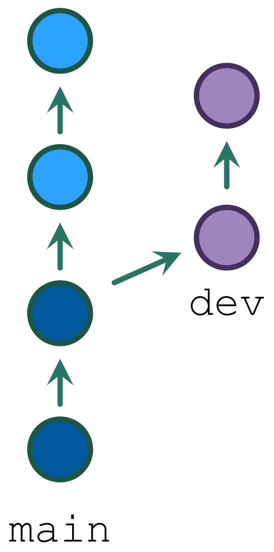


Branching and merging with `git`

There are multiple methods to bring parallel developments back together

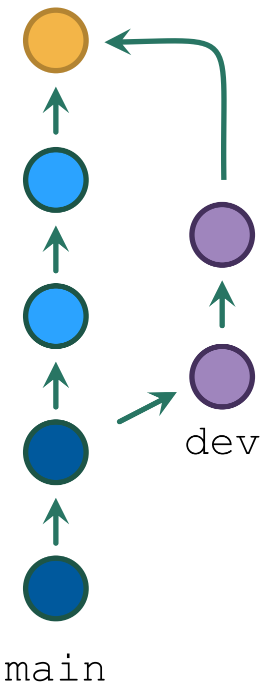
Getting started with branching

○ ... a commit

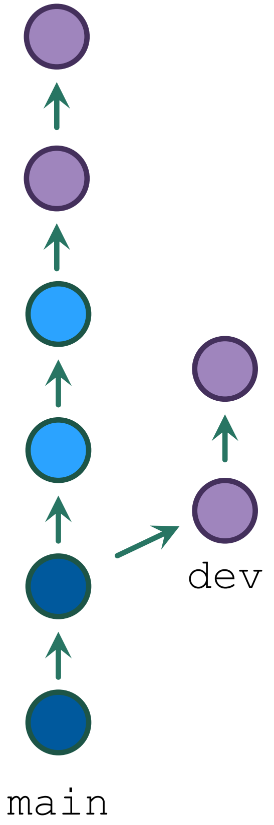


Three options to ***merge*** the changes from `dev` into `master`

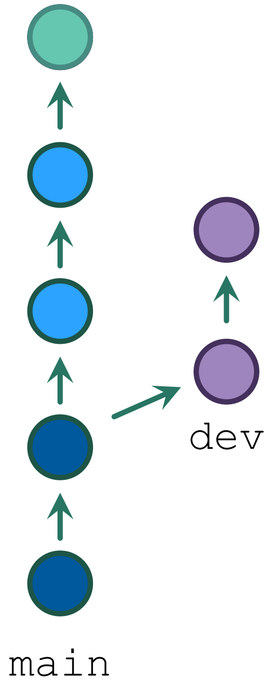
1) A merge commit



2) Rebase



3) Squash and merge



Writing good `git` commit messages

If at the end of the day/week/year, you don’t remember what you did…

Useful recommendations to help you (and your colleagues) keep track of your work

⇒ Limit the subject line (summary) to 50 characters

⇒ Capitalize the subject line

⇒ Do not end the subject line with a period

⇒ Use the imperative mood in the subject line

⇒ Use the body to explain what and why vs. how

A properly formed Git commit summary should be able to complete the following sentence:

If applied, this commit will *your subject line here*

- If applied, this commit will *update getting started documentation*
- If applied, this commit will *release version 1.0.0*
- If applied, this commit will *merge pull request #123 from user/branch*

Selected items from chris.beams.io/posts/git-commit/

Part 2

Setting up a simple repository
with unit tests and continuous integration

The first rule of live demos: Never do a live demo. So let’s do a live demo.

Hands-on exercise: github.com/danielhuppmann/lecture-spring-2023

- Set up a new public GitHub repository at www.github.com
- Add a license (why not start with APACHE 2.0?)
- Update the README (formatting using markdown)
- “Clone” the repository to your computer (recommended for novices: gitkraken.com)
- Add the license statement and the badge to the readme
- Start developing a little Python function (recommended for novices: anaconda.com)
- Add a unit test
- Add a gitignore file
- Add continuous integration using a new branch
  ⇒ GitHub Actions to execute unit tests
  ⇒ stickler-ci to implement linter and code style verification
- Create a pull request to execute the CI and merge the new branch into `main`

Hands-on exercise (Part II)

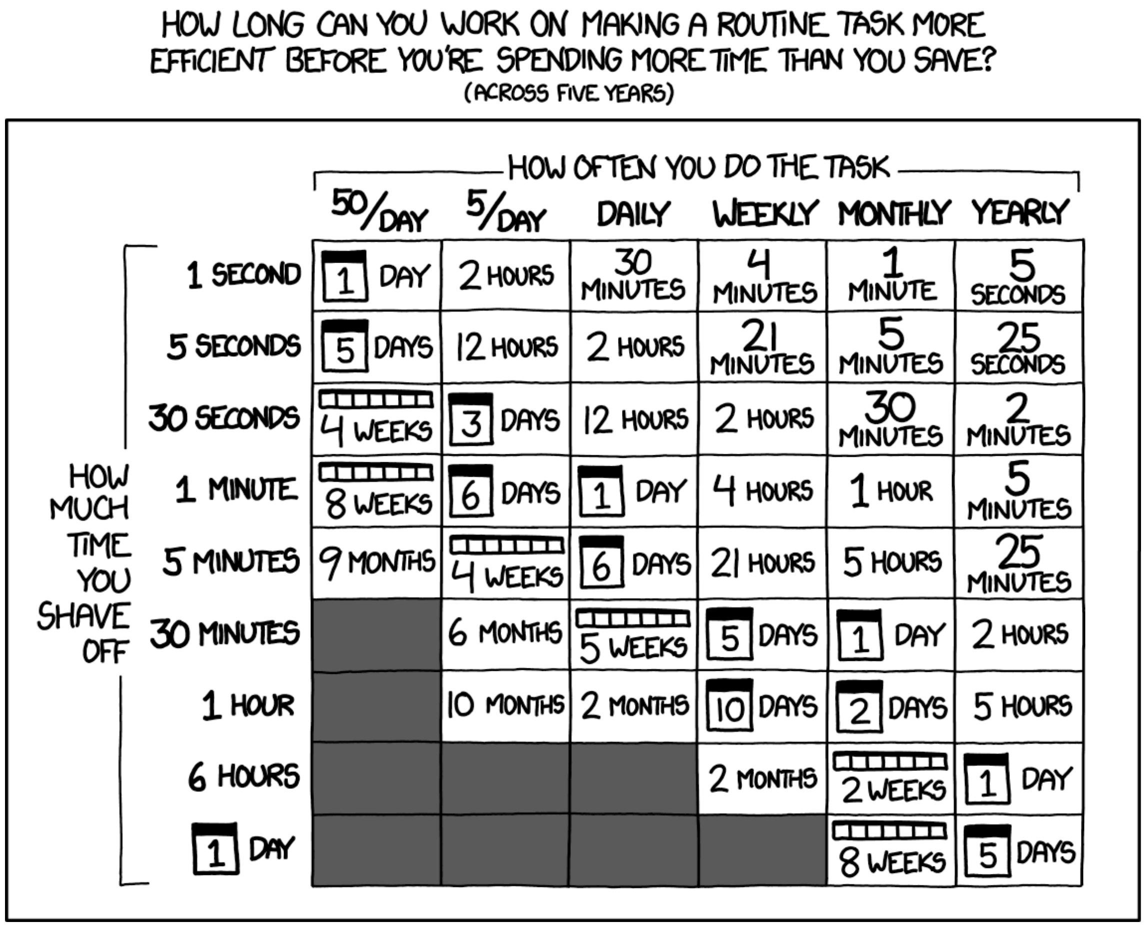
- If a non-admin user wants to push commits, you have to “fork” the repo (create a copy under your GitHub user)
- Clone the fork to your computer
- Start a new branch
- Add a new function or extend some feature such that the unit tests fail
- Make a pull request to the upstream repository
- Fix the code such that unit tests pass
- Ask someone else to perform code review
- Merge the new development (by an admin)
- Add contributing guidelines, set up templates for pull requests
- Create a release

Part 3

Some practical considerations and advice

Time allocation for increasing efficiency through automation

Is it worth the time to automate repetitive tasks? Probably not really…



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK					
	50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES	▨	6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR	▨	10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS	▨	▨	▨	2 MONTHS	2 WEEKS	1 DAY
1 DAY	▨	▨	▨	▨	8 WEEKS	5 DAYS

xkcd by Randall Munroe

Good enough scientific programming

You don't have to have a PhD in IT to do decent scientific programming!
In fact, it might actually help...

Data management:

⇒ save both raw and intermediate forms, create tidy data amenable to analysis

Software:

⇒ write, organize, and sharing scripts and programs used in the analysis following best practices

Collaboration:

⇒ make it easy for existing and new collaborators to understand and contribute to a project

Project organization:

⇒ organize the digital artefacts of a project to ease discovery and understanding

Manuscripts:

⇒ write manuscripts with a clear audit trail and minimize manual merging of conflicts

Adapted from Greg Wilson et al. Good enough practices in scientific computing. *PLoS Comput. Biol.* 13(6), 2017.
doi: 10.1371/journal.pcbi.1005510

Good enough scientific programming – Software

Your worst collaborator? Yourself from six months ago...

- Place a brief explanatory comment at the start of every program.
- Do not comment and uncomment sections of code to control a program's behaviour.
- Decompose programs into functions, and try to keep each function short enough for one screen.
- Be ruthless about eliminating duplication.
- Always search for well-maintained software libraries that do what you need.
- Test libraries before relying on them.
- Give functions and variables meaningful names.
- Make dependencies and requirements explicit.
- Provide a simple example or test data set.
- Submit code to a reputable DOI-issuing repository (e.g., zenodo).

Adapted from Greg Wilson et al. Good enough practices in scientific computing. *PLoS Comput. Biol.* 13(6), 2017.
doi: 10.1371/journal.pcbi.1005510

Code style guides

Programming should be seen as a (not foreign) language

Which programming language to use, which other conventions to follow?

⇒ If you don’t have a strong preference: follow the community or your room (office) mate!

Some practical guidelines:

⇒ Follow a coding etiquette, e.g., Black & PEP8 for Python, Google’s R style guide

⇒ For larger projects, agree on a folder structure and hierarchy early (source data, etc.)

⇒ Only change folder structure when it’s really necessary

⇒ For more complex code (e.g., packages), use tools to automatically build documentation such as Sphinx and readthedocs.org

Keep in mind...

⇒ Code is read more often than it is written

⇒ Good code should not need a lot of documentation

⇒ Key criteria: readability and consistency with (future) collaborators *and yourself!*

Software releases and semantic versioning

If a piece of software is used by multiple people, clear versioning is critical

Semantic versioning uses a structure like `<MAJOR>.<MINOR>.<PATCH>`

For a new release (i.e., a published version), you *MUST* increment...

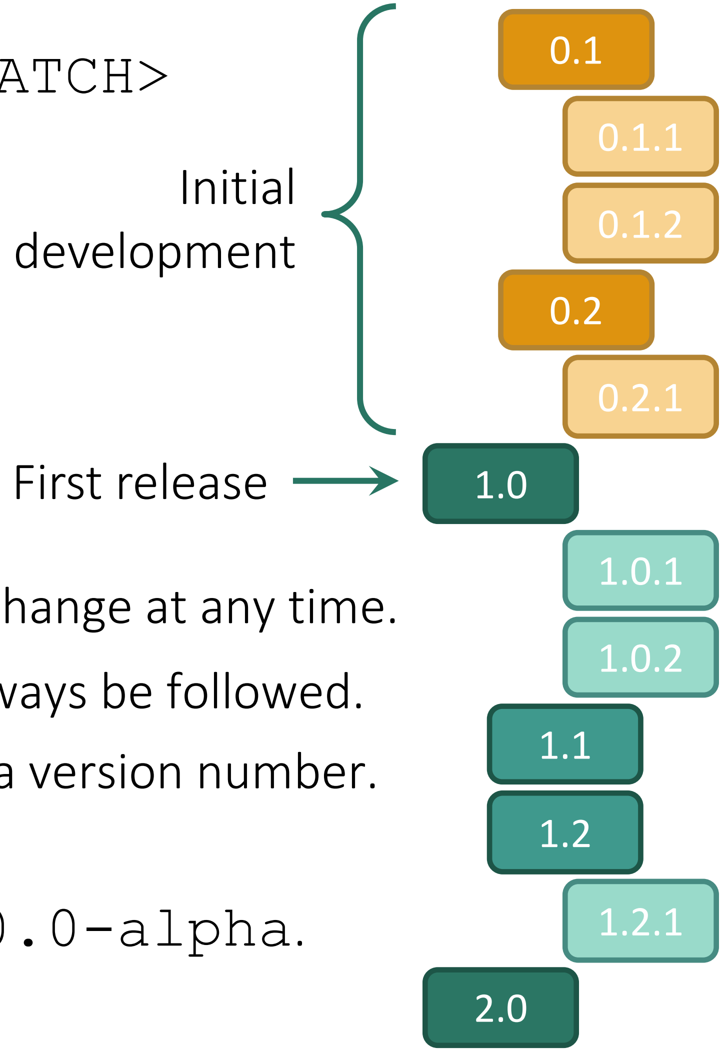
⇒ `MAJOR` when making incompatible API changes,

⇒ `MINOR` when adding backwards-compatible functionality,

⇒ `PATCH` when making backwards-compatible bug fixes.

Other considerations:

- Major version zero (`0.y.z`) is for initial development. Anything may change at any time.
- Version `1.0.0` defines the public API. After that, rules above must always be followed.
- Downstream version numbers *MUST* be reset to 0 when incrementing a version number.
- You *MAY* increment when substantial new internal features are added.
- A pre-release version *MAY* be denoted by appending a string, e.g., `1.0.0-alpha`.

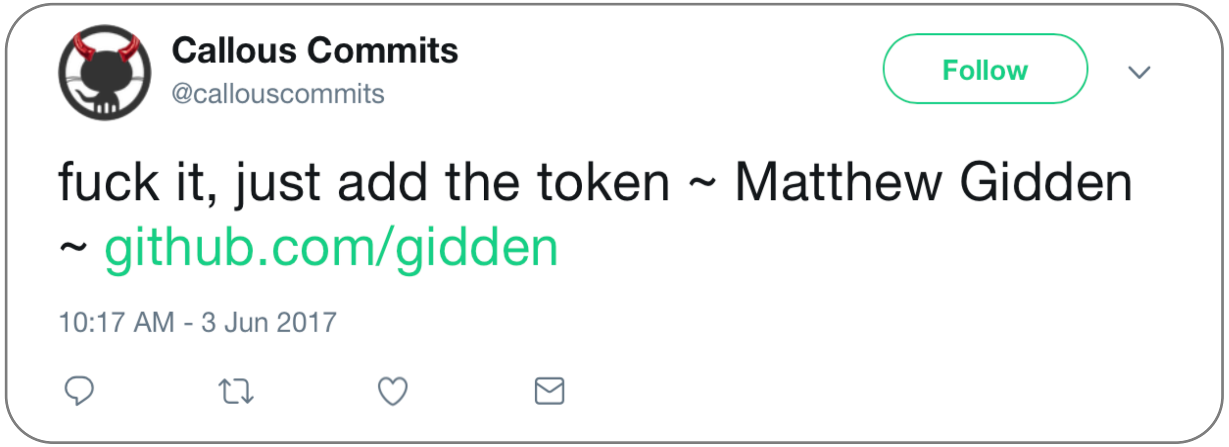


Adapted from Semantic Versioning 2.0.0, semver.org

Coding etiquette

Keep in mind that the internet remembers everything

When you search for my colleague Matthew Gidden on Twitter, the first tweet you find is...



Callous Commits
@callouscommits

fuck it, just add the token ~ Matthew Gidden ~ github.com/gidden

10:17 AM - 3 Jun 2017

Social etiquette

Be kind and respectful in collaboration, code review and comments

Collaborative scientific programming is about communication, not code...

⇒ It’s the people, stupid!

⇒ And don’t be annoyed when, sometimes, some collaborators are stubborn...

Keep in mind that discussions via e-mail, chat, pull requests comments, code review, etc.
lack a lot of the social cues that human interaction is built upon

If there are two roughly equivalent ways to do something
and a code reviewer suggests that you use the other approach...

⇒ Just do it her/his way if there is no good reason not to – out of respect for the reviewer and to avoid getting bogged down in escalating discussions

Give credit generously to your collaborators and contributors!

Homework assignment

Create a simple repository based on any of your real-life projects

- Start a new GitHub repository, add a license and set up continuous-integration (CI) tools
- Add functions or small features from any real-life project relevant to your work or interests
  - The codebase should include 2-4 functions, 20-40 lines of code **including documentation**
  - The repository should work as "stand-alone" project
    (i.e., no need for other parts of your project/work that are not part of this repository)
  - If you need any dependencies/packages, add a simple list in a file `requirements.txt`
  - Add at least one test per function and make sure that these are executed on CI
  - If data is necessary to understand the scope of the functions, add a stylized dataset
- The README should explain the scope of the project and the purpose of the functions
- Invite me as a collaborator to your repository when the project is ready to be reviewed/graded

⇒ Programming languages: Python (preferred), R, Julia

⇒ Invitation to collaborate due by Monday, April 10, 23:59 (please do not push any commits after)

Thank you very much for your attention!

Dr. Daniel Huppmann

Senior Research Scholar – Energy, Climate, and Environment Program

International Institute for Applied Systems Analysis (IIASA)
Schlossplatz 1, A-2361 Laxenburg, Austria

huppmann@iiasa.ac.at

@daniel_huppmann

@daniel_huppmann@mastodon.social

www.iiasa.ac.at/staff/daniel-huppmann