Train the trainer workshop

August/September 2024 CodeRefinery train the trainer workshop

Do you teach the use of computers and computational tools? Are you curious about scaling your training and learn about tested and proven practices from CodeRefinery workshops and other trainers? Join us for the CodeRefinery train the trainer workshop: four self-contained sessions on tools and techniques for computational training offer a great chance to enhance your teaching skills and learn about new tools and practices. What you will learn is also used a lot outside CodeRefinery, whenever good beginner friendly training is needed.

Learning objectives:

- Learn about tools and techniques and practice to create engaging online teaching experiences (screenshare, audio, etc.).
- Become mentally ready to be an instructor in a collaborative interactive online workshop (not only large workshops but in general).
- Learn how to design and develop lesson material collaboratively.

Target audience:

- Everyone teaching online workshops about computational topics or interested in teaching.
- Previous and future instructors and helpers of CodeRefinery workshops.

Prerequisites: An interest in teaching.

Organizers and instructors:

The workshop is organized by partner organizations of the CodeRefinery project.

Facilitators and instructors:

- Radovan Bast
- Richard Darst
- Bjørn Lindi
- Dhanya Pushpadas
- Jarno Rantaharju
- Stephan Smuts

- Stepas Toliautas
- Samantha Wittke

Content and timing:

The workshop consists of four sessions every Tuesday between August 13th and September 3rd 2024, 9-12 CEST:

- Session 1: About lesson design, deployment and iterative improvement (Aug 13)
- Session 2: Tools and techniques adopted in CodeRefinery workshops (Aug 20)
- Session 3: About teaching & cool things we all would like to share (Aug 27)
- Session 4: Workshop streaming practices and post-workshop tasks (Sep 3)

You can join all sessions, or the just the ones that interest you. More details on each session will be shared later.

The workshop is **free of charge for everyone**, please register below to get the Zoom link and other useful information for the workshop.

Registration

If you have any questions, please write to support@coderefinery.org.

Materials are work in progress

Materials will appear here before the workshop. You can find materials of previous similar trainings using the links below:

- First version of the course in 2019 and then in 2020: https://coderefinery.github.io/train-the-trainer/branch/instructor-training/
- Reworked material for our summer workshop 2022 and for the CarpentryCon 2022 workshop: https://coderefinery.github.io/train-the-trainer/branch/community-teaching/

Lesson design and development

Objectives

- We share our design processes for teaching material and presentations.
- Learn how to design lessons “backwards”, starting from learning objectives and learner personas.
- Learn good practices for improving existing material based on feedback.

Instructor note

- Discussion: 20 min
- Exercises: 35 min

Exercise: How do you design your teaching material?

✍ We collect notes using a shared document (5 min)

- When you start preparing a new lesson or training material, where do you start?
- What tricks help you with “writer’s block” or the empty page problem?
- Maybe you haven’t designed training material yet. But how do you start when creating a new presentation?
- If your design process has changed over time, please describe what you used to do and what you do now instead.
- What do you know now about preparing lessons/training/presentations that you wish you knew earlier?

Creating new teaching material

Typical problems

- Someone creates a lesson, but they think about what is interesting to them, not what is important for the learners.
- “I want to show a number of things which I think are cool about tool X - how do I press these into 90 minutes?”
- Write down material you want to cover and then sprinkle in some exercises.
- Thinking about how I work, not how the learners work.
- Trying to bring learners to their level/setup, not trying to meet the learners where they are.
- Not really knowing the learning objectives or the learner personas.

Better approach

Good questions to ask and discuss with a group of colleagues **from diverse backgrounds**:

- What is the expected educational level of my audience?
- Have they been already exposed to the technologies I am planning to teach?
- What tools do they already use?
- What are the main issues they are currently experiencing?
- What do they need to remember/understand/apply/analyze/evaluate/create (Bloom’s taxonomy)?
- Define learner personas.

- It may be an advantage to share an imperfect lesson with others early to collect feedback and suggestions before the lesson "solidifies" too much. Draft it and collect feedback. The result will probably be better than working in isolation towards a "perfect" lesson.

The process of designing a lesson "backwards"

As described in "A lesson design process" in the book Teaching Tech Together:

1. Understand your learners.
2. Brainstorm rough ideas.
3. Create an summative assessment to know your overall goal.

   [Think of the things your learners will be able to do at the end of the lesson]

4. Create formative assessments to go from the starting point to this.

   [Think of some engaging and active exercises]

5. Order the formative assessments (exercises) into a reasonable order.
6. Write just enough material to get from one assessment (exercise) to another.
7. Describe the course so the learners know if it is relevant to them.

Improving existing lessons

💬 All CodeRefinery lessons are on GitHub

- Overview: https://coderefinery.org/lessons/
- All are shared under CC-BY license and we encourage reuse and modification.
- Sources are all on GitHub: https://github.com/coderefinery
- Web pages are generated from Markdown using Sphinx (more about that in the episode Lessons with version control).
- We track ideas and problems in GitHub issues.

Collect feedback during the workshop:

- Collect feedback from learners and instructors (Example from a past workshop).
- Convert feedback about lessons and suggestions for improvements into issues so that these don't get lost and stay close to the lesson material.

Collect feedback before you start a big rewrite:

- First open an issue and describe your idea and collect feedback before you start with an extensive rewrite.
- For things still under construction, open a draft pull/merge request to collect feedback and to signal to others what you are working on.

Small picture changes vs. big picture changes:

- Lesson changes should be accompanied with instructor guide changes (it's like a documentation for the lesson material).
- Instructor guide is essential for new instructors.
- Before making larger changes, talk with somebody and discuss these changes.

Use case: our lessons

As an example to demonstrate the process of designing and improving lessons, we will have a look at one of our own lessons: Introduction to version control with Git.

- Initial 2014-2016 version
  - https://github.com/scisoft/toolbox-talks and https://toolbox.readthedocs.io/
  - Amazingly they are still findable!
  - Format: Slides and live coding.
  - Exercises were separate, during afternoon sessions.
- Some time in 2014-2015 attended Carpentries instructor training.
- 2016: CodeRefinery started.
- 2017: Started a new repository based on the Carpentries lesson template (at the time using Jekyll).
  - Exercises become part of the lesson.
  - We start in the **command line** and only later move to GitHub.
- 2019: A lot more thought about learning objectives and personas.
  - Also license change to CC-BY.
- 2022: Convert lesson from Jekyll to Sphinx.
  - Using the tools that we teach/advocate.
  - We can have tabs and better code highlighting/emphasis.
  - Easier local preview (Python environment instead of Ruby environment which we were not used to in our daily work).
- 2024: Big redesign. We move the lesson closer to where learners are.
  - Start from GitHub instead of on the command line.
  - Start from an existing repository instead of with an empty one.
  - Offer several tracks to participate in the lesson (GitHub, VS Code, and command line) and learners can choose which one they want to follow.
  - Blog post: We have completely changed our Git lessons. Hopefully to the better.
- Next steps?
  - Making the lesson citable following our blog post.
  - Improvements based on what we learn from this workshop.

The overarching trend was to make the lesson simpler and more accessible - to meet the learners where they are instead of pulling them to the tool choices of the instructors. Looking back, we learned a lot and the learning process is not over yet.

Exercise: Discussion about learning objectives and exercise design

✍️ We work in groups but use the shared document as result (20 min)

1. As a group **pick a lesson topic**. It can be one of the topics listed here but you can also choose something else that your group is interested in, or a topic that you have taught before or would like to teach. Some suggestions:
   - Git: Creating a repository and porting your project to Git and GitHub
   - Git: Basic commands
   - Git: Branching and merging
   - Git: Recovering from typical mistakes
   - Code documentation
   - Jupyter Notebooks
   - Collaboration using Git and GitHub/GitLab
   - Using GitHub without the command line
   - Project organization
   - Automated testing
   - Data transfer
   - Data management and versioning
   - Code quality and good practices
   - Modular code development
   - How to release and publish your code
   - How to document and track code dependencies
   - Recording environments in containers
   - Profiling memory and CPU usage
   - Strategies for parallelization
   - Conda environments
   - Data processing using workflow managers
   - Regular expressions
   - Making papers in LaTeX
   - Making figures in your favorite programming language
   - Linux shell basics
   - Something non-technical, such as painting a room
   - Introduction to high-performance computing
   - A lesson you always wanted to teach
   - …
2. Try to define 2-3 learning objectives for the lesson and write them down. You can think of these as "three simple enough messages that someone will remember the next day" - **they need to be pretty simple**.
3. Can you come up with one or two engaging exercises that could be used to demonstrate one of those objectives? **They should be simple** enough people can actually do them. Creating simple exercises is not easy. Some standard exercise types:
   - Multiple choice (easy to get feedback via a classroom tool - try to design each wrong answer so that it identifies a specific misconception).
   - Code yourself (traditional programming)
   - Code yourself + multiple choice to see what the answer is (allows you to get feedback)
   - Inverted coding (given code, have to debug)

- Parsons problems (working solution but lines in random order, learner must only put in proper order)
- Fill in the blank
- Discussions, self directed learning exercises

Great resources

- Teaching Tech Together
- Our summary of Teaching Tech Together
- Ten quick tips for creating an effective lesson
- Carpentries Curriculum Development Handbook
- Our manual on lesson design

Lessons with version control

Objectives

- Understand why version control is useful even for teaching material
- Understand how version control managed lessons can be modified.
- Understand how the CodeRefinery lesson template is used to create new lessons

Instructor note

- Discussion: 25 min
- Exercises or demos: 20 min

Why version control?

- If you are in CodeRefinery TTT, you probably know what version control is and why it is important.
- The benefits of version control also extend to lessons:
  - Change history
  - Others can submit contributions
  - Others can make derived versions and sync up later
  - Same workflow as everything else
  - Write it like documentation: probably more reading after than watching it as a presentation.
- Disadvantages
  - "What you see is what you get" editing is hard
  - Requires knowing version control

💬 Accepting the smallest contribution

Question: if someone wants to make a tiny fix to your material, can they?

Tour of lesson templates options

There are different ways to make lessons with git. Some dedicated to teaching:

- CodeRefinery
  - Example: This lesson itself
  - Based on the Sphinx documentation generator
  - sphinx-lesson is *very* minimal extra functionality
- Carpentries
  - Example: https://carpentries.github.io/lesson-example/
  - Based on R and Rmarkdown

Our philosophy is that anything works: it doesn't have to be just designed for lessons

- Jupyter Book
  - Example: https://jupyterbook.org/
  - Note: is based on sphinx, many extensions here are used in CR lessons
- Various ways to make slides out of Markdown
- Cicero: GitHub-hosted Markdown to slides easily
  - Demo: Asking for Help with Supercomputers The source
- Whatever your existing documentation is.

We like the CodeRefinery format, but think that you should use whatever fits your needs the most.

Sphinx

- We build all our lesson material with Sphinx
- Generate HTML/PDF/LaTeX from RST and Markdown.
- Many Python projects use Sphinx for documentation but **Sphinx is not limited to Python**.
- Read the docs hosts public Sphinx documentation for free!
- Also hostable anywhere else, like Github pages, like our lesson material
- For code a selling point for Sphinx is that also API documentation is possible.

Sphinx is a doc generator, not HTML generator. It can:

- Markdown, Jupyter, and ReST (and more...) inputs. Executable inputs.
  - jupyter-book is Sphinx, so anything it can do we can do. This was one of the inspirations for using Sphinx
- Good support for inline code. Much more than static code display, if you want to look at extensions.
- Generate different output formats (html, single-page html, pdf, epub, etc.)
- Strong cross-referencing within and between projects

CodeRefinery lesson template

It is “just a normal Sphinx project” - with extensions:

- Sphinx lesson extension
  - adds is various directives (boxes) tuned to lesson purposes
  - provides a sample organization and template repo you can use so that lessons look consistent
- Sphinx gives us other nice features for free
  - Tabs: they are very important for us and allow us to customize in-place instead of copying everything and fragmenting lessons
  - Emphasize lines: they make it easier to spot what has changed in longer code snippets
  - Various input formats
    - Markdown (via the MyST-parser), ReStructured text, Jupyter Notebooks.
  - Many other features designed for presenting and interacting with code
- It’s fine if you use some other static site generator or git-based lesson method.

👀 Instructors go through the building and contributing process

Depending on the course, instructors will demo what is roughly exercise 4 below. Or a course might go straight to exercises.

- Instructors decide what change they would want to make
- Instructors clone the repository
- **Instructors make the change**
- **Instructors set up the build environment**
- **Instructors build and preview**
- Instructors command and send upstream

Exercises

Some exercises have prerequisites (Git or Github accounts). Most instances of this course will have you do **1 and 2** below.

✍ Lesson-VCS-1: Present and discuss your own lesson formats

We don’t want to push everyone towards one format, but as long as you use Git, it’s easy to share and reuse.

- Discuss what formats you do use
- Within your team, show examples of the lessons formats you use now. Discuss what is good and to-be-improved about them.
- Look at how they source is managed and how easy it might be to edit.

✍ Lesson-VCS-2: Tour a CodeRefinery or Carpentries lesson on Github

- Look at either a CodeRefinery or Carpentries lesson
  - CodeRefinery Git-Intro: Lesson, Github repo
  - Carpentries Linux shell: Lesson, Github repo
- Can you find
  - Where is the content of the lessons?
  - What recent change propsals (pull requests) have been made?
  - What are the open issues?
  - How you would contribute something?
  - How would you use this yourself?

✍ Lesson-VCS-3: Modify a CodeRefinery example lesson on Github

In this, you will make a change to a lesson on Github, using only the Github interface. (Github account required, and we assume some knowledge of Github. Ask for help in your team if it is new to you!)

- Navigate to the example lesson we have set up: repo, web
- Go to some page and follow the link to go to the respective page on Github. (Alternatively, you can find the page from the Github repo directly).
- Follow the Github flow to make a change, and open a Pull Request with the change proposal:
  - Click on the pencil icon
  - Make a change
  - Follow the flow to make a commit and change. You'll fork the repository to make your own copy, add a message, and open a pull request.

We will look at these together and accept some changes.

✍ Lesson-VCS-4: Clone and build a CodeRefinery lesson locally

In this exercise, you will use Git to clone one a CodeRefinery lesson and try to preview it locally. It assumes installation and knowledge of Git.

- Use this sample repository: git-intro (or whatever else you would like)
- Clone the repository to your own computer
- Create a virtual environment using the `requirements.txt` contained within the repository.
- Build the lesson.
  - Most people will probably run: `sphinx-build content/ _build/`
  - If you have `make` installed you can `make html`
  - Look in the `_build` directory for the built HTML files.

Overall, this is pretty easy and straightforward, if you have a Python environment set up. The CodeRefinery documentation lesson teaches this for every operating system.

This same tool can be used to build documentation for other software projects and is pretty standard.

✍ Lesson-VCS-5: (advanced) Create your own lesson using the CodeRefinery format

In this lesson, you'll copy the CodeRefinery template and do basic modifications for your own lesson.

- Clone the lesson template: https://github.com/coderefinery/sphinx-lesson-template
- Attempt to build it as it is (see the previous exercise)
- How can you do tabs?
- How can you highlight lines in code boxes?
- How can you change color, logo and fonts?
- What directives are available?

Summary

Keypoints

- Version control takes teaching materials to the next level: shareable and easy to contribute
- There are different formats that use version control, but we like Sphinx with a sphinx-lesson extension.

How we collect feedback and measure impact

Objectives

- Discuss how one can collect feedback from learners ("what can we improve?").
- Discuss how we convert feedback into actionable items.
- Discuss how we measure the impact of teaching ("did we achieve our goals?").
- Discuss the "why".
- Get to know the reasons and sources of inspiration behind major lesson and workshop updates.

Instructor note

- Discussion: 20 min
- Exercises: 10 min

Asking questions before the workshop

- Motivation: Know your audience.
- Until 2021 we had a pre-workshop survey:
  - Data, questions, and notebook: https://github.com/coderefinery/pre-workshop-survey/
  - Zenodo/DOI: https://doi.org/10.5281/zenodo.2671578
- After 2021 we incorporated some of the questions into the registration form.
  - Easier registration experience for participants.
  - After 5 years of running workshops, we had a good idea of what to expect.

Collecting feedback as we teach

- Each day we ask for feedback in the collaborative notes.
  - One good thing about today.
  - One thing to improve for next time.
  - Any other comments?
- During the workshop we sometimes check in and ask about the pace, example:

```
How is the speed so far? (add an "o")
- Too fast:   oooooo
- Too slow:   ooo
- Just right: ooooooooooooooooooo
```

- We publish all questions, answers, and feedback. Example: https://coderefinery.github.io/2024-03-12-workshop/questions/
- How we follow up:
  - Some problems we can fix already before the next workshop day.
  - We convert feedback/problems into GitHub issues and track these close to the lesson material.

Trying to measure impact with longer-term surveys

- Motivation: Understand the long-term impact of our workshops. Have something to show to funders.
- 2024 post-workshop survey:
  - Blog post: https://coderefinery.org/blog/2024/08/10/post-workshop-survey/
  - Questions, notebook, and figures: https://github.com/coderefinery/2024-post-workshop-survey
  - Zenodo/DOI: https://doi.org/10.5281/zenodo.13292363
- 2021 version:
  - Data, questions, notebook, and figures: https://github.com/coderefinery/post-workshop-survey

    - Zenodo/DOI: https://doi.org/10.5281/zenodo.2671576
- How we use the results:
    - When reporting to funders.
    - When planning future workshops and bigger picture changes.

Lessons learned

- Think about how to measure impact/success from the beginning.
- **Make the feedback and survey results public**.
- Make the results persistent and citable.
- Have a mechanism to follow-up on feedback.
- Anonymization is more than just removing or dissociating names.
- Take time designing your survey and collect feedback on the survey itself.

Take time designing your survey

We are not experts in survey design but we reached out to RISE Research Institutes of Sweden to get feedback on our survey questions. They provided us with a lot of valuable feedback and suggestions for improvements. Below are few examples.

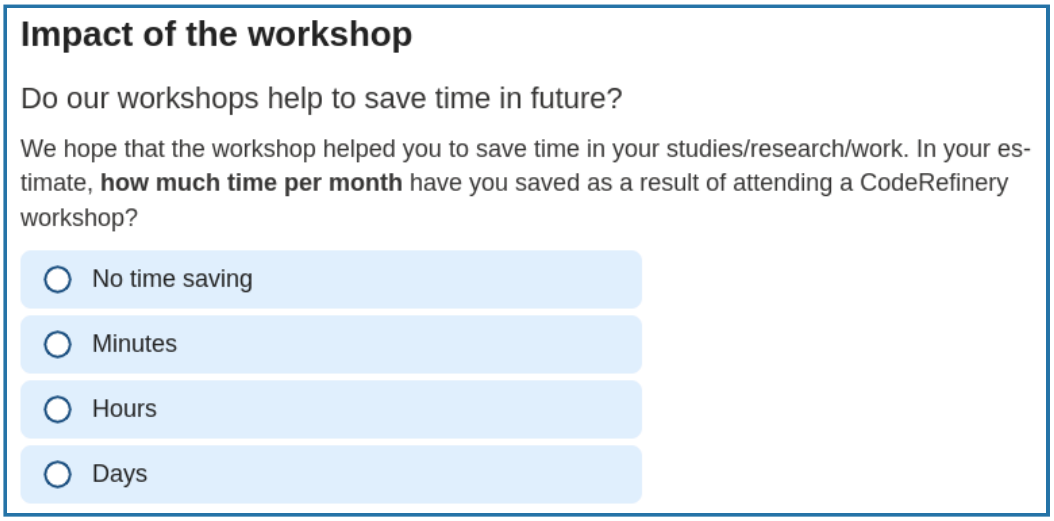
Example 1

First version of our survey question about impact:

- "Do our workshops help to save time in future?" Please quantify in hours saved: ...

Feedback:

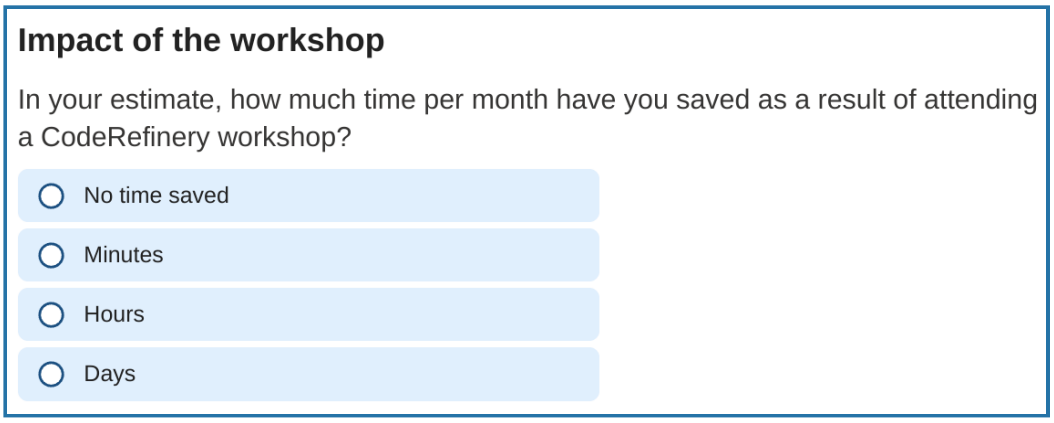
- Difficult to answer.
- Since when? Until all eternity?



Earlier version of the survey question about impact.

Feedback:

- The question “Do our workshops help to save time in future?” is unspecified, unnecessary, and leading.
- “No time saving” does not match the wording “save time” in the question.



Later version of the survey question about impact.

- The wording “have you saved” now matches “No time saved”.

Example 2

- Earlier version:

```
Would you judge your code to be better reusable/reproducible/modular/documented
as a result of attending the workshop?

- More reusable
- More reproducible
- More modular
- Better documented
- None of the above
```

- Feedback: The question is not neutrally formulated and risks leading to over-reporting of yes answers. Consider balancing so that the questions are formulated more neutrally.
- Reformulated to:

```
After attending the workshop, would you judge your code to be
more reusable or not more reusable?

- My code is more reusable
- My code is not more reusable
- Not sure
```

Example 3

- Early version:

```
What else has changed in how you write code since attending the workshop?

[free-form text field]
```

- Feedback: Leading and assumes that something has changed.
- Reformulated to:

```
Has anything else changed in how you write code for your research after
attending the workshop?

[free-form text field]
```

Example 4

- Earlier version:

```
Has it become easier for you to collaborate on software development with your
colleagues and collaborators?

- Yes
- Not sure
- No
```

- Feedback:
  - Leading question.
  - Avoid “Yes” and “No” response options because respondents tend to answer “Yes” if that option is available, leading to a risk of measurement error (acquiescence bias).
  - Do not place “Not sure” in the middle of the scale because it captures participants who actually don’t know and should therefore be placed at the bottom instead of in the middle of the scale.
- Reformulated to:

```
After attending the workshop, has it become easier or not for you to
collaborate on software development with your colleagues and collaborators?

- Collaboration is easier
- Collaboration is not easier
- Not sure
```

Exercise: Group discussion (10 min)

✍️ Group discussion using the collaborative notes

- What tricks/techniques have you tried in your teaching or seen in someone else’s teaching that you think have been particularly effective in collecting feedback from learners?
- Can you give tips or share your experiences about how to convert feedback into actionable items?
- How do you measure the impact of your teaching? Any tips or experiences about what you have tried or seen other courses do?

- Anybody knows of good resources on survey design? Please link them in the collaborative notes.

About the CodeRefinery project and CodeRefinery workshops in general

Keypoints

- Teaches intermediate-level software development tool lessons
- Training network for other lessons, too
- Publicly-funded discrete projects (3 projects actually) transitioning towards an open community project
- We have online material, teaching, and exercise sessions
- We want more people to work with us, and to work with more people

CodeRefinery is a Nordic e-Infrastructure Collaboration (NeIC) project that has started in October 2016 and is funded until February 2025.

The funding from 2022-2025 is designed to keep this project active beyond 2025 by forming a support network and building a community of instructors and contributors.

History

The CodeRefinery project idea grew out of two SeSE courses given at KTH Stockholm in 2014 and 2016. The project proposal itself was submitted to NeIC in 2015, accepted in 2015, and started in 2016.

We have started by porting own lessons to the Carpentries teaching style and format, and collaboratively and iteratively grew and improved the material to its present form.

Main goals

- Develop and maintain **training material on software best practices** for researchers that already write code. Our material addresses all academic disciplines and tries to be as programming language-independent as possible.
- Provide a code repository hosting service that is open and free for all researchers based in universities and research institutes from Nordic countries.
- Provide **training opportunities** in the Nordics using (Carpentries and) CodeRefinery training materials.
- Articulate and implement the CodeRefinery **sustainability plan**.

Impact

We collect feedback and survey results to measure our impact.

3-24 months after attending a workshop, past participants are asked to complete a short post-workshop survey. The survey questions aim to establish what impact CodeRefinery workshops have on how past participants develop research software.

Pre- and post-workshop survey results

- Overall quality of research software has improved: more reusable, modular, reproducible and documented.
- Collaboration on research software development has become easier
- Past participants share their new knowledge with colleagues
- Usage of several tools is improved, and new tools are adopted

Free-form answers also suggest that workshops are having the intended effects on how people develop code. A common theme is:

> *I wish I had known this stuff already as a grad student 10+ years ago...*

We would love to get suggestions on how we can better quantify our impact. This would make it easier for us to convince institutions to partner with us and also open up funding opportunities.

Target audience

Carpentries audience

The Carpentries aims to teach computational **competence** to learners through an applied approach, avoiding the theoretical and general in favor of the practical and specific.

Mostly, learners do not need to have any prior experience in programming. One major goal of a Carpentry workshop is to raise awareness on the tools researchers can learn/use to speed up their research.

By showing learners how to solve specific problems with specific tools and providing hands-on practice, learners develops confidence for future learning.

Novices

We often qualify Carpentry learners as **novices**: they do not know what they need to learn yet. A typical example is the usage of version control: the Carpentry `git` lesson aims to give a very high level conceptual overview of Git but it does not explain how it can be used in research projects.

CodeRefinery audience

In that sense, CodeRefinery workshops differ from Carpentry workshops as we assume our audience already writes code and scripts and we aim at teaching them **best software practices**.

Our learners usually do not have a good overview of **best software practices** but are aware of the need to learn them. Very often, they know the tools (Git, Jupyter, etc.) we are teaching but have difficulties to make the best use of them in their software development workflow.

Whenever we can, we direct learners that do not have sufficient coding experience to Carpentries workshops.

Competent practitioners

We often qualify CodeRefinery learners as **competent practitioners** because they already have an understanding of their needs.

Best software practices for whom?

It can be useful to ask the question: *best software practices for whom*? CodeRefinery teaches *best software practices* derived from producing and shipping software. These practices are also very good for sharing software, though our audience will probably not need to embrace *all* aspects of software engineering.

Workshop overview and roles

Objectives

- Understand the general structure of CodeRefinery workshops and why it is the way it is
- Get to know the different roles of a workshop and which ones are the most essential ones

Instructor note

- Teaching: ? min
- Exercises: ? min

Note

This episode focuses on the large scale streamed workshop setup, we do still also provide smaller scale online and in-person workshops and can also support you offering your own!

💬 Discussion

Discussion: Benefits/Challenges of online and in-person teaching

- some answers leading over to “best of both worlds”

A typical CodeRefinery workshop

- Everyone watches stream
- Co-instructors present content
- Collaborative notes
  - Q&A
  - Feedback / check-in
  - Comments
- Hands-on
  - individual
  - with others

How did we get here?

- collaborative in-person workshops since 2016
- moving online in 2020
- started with “traditional zoom” workshops
- thought about scaling and ease of joining

Workshop roles and their journeys

Individual learner journey

- Registration
- Installation
- (if appicable: Invite to local meetup)
- Workshop
  - Watch stream
  - Q&A in notes
  - Exercises alone or in team
  - Daily feedback
- Post workshop survey

Instructor journey

- Indicate interest in CR chat
- Onboarding
- Lesson material updates
- Teaching
- (if wished: support answering questions in notes)

- Debrief

Team lead / Local host journey

- Registration
- (if applicable: run own registration)
- Onboarding
- Workshop
  - Watch stream
  - Facilitate exercises/discussions
  - Daily feedback
- Debrief / feedback / survey

Collaborative notes manager

Before workshop:

- Update copy-pastable notes template
- Prepare Icebreakers

During workshop:

- Add headers as the workshop progresses
- Keep anonymous
- Make sure all questions have an answer
- Add feedback section at end of session

After workshop:

- Archive the document on workshop website

Other roles

Director and broadcaster roles will be discussed in Session 4. For more in-depth descriptions and other roles (host, instructor, registration coordinator), see manuals: https://coderefinery.github.io/manuals/roles-overview/

💬 Discussion

Discussion: Choose a role and check the manuals, do they prepare you for that role? What did you learn?

❗ Keypoints

- Here we summarize keypoints.

Onboarding

Objectives

- Understand the importance of installation instructions and how they contribute to learners success
- Understand the importance of onboarding and a welcoming community for volunteers in a workshop

Instructor note

- Teaching: ? min
- Exercises: ? min

Onboarding learners

- E-mail communication outlining what will happen during workshop
- Installation instructions: https://coderefinery.github.io/installation/
- Installation support session

Discussion

Installation instructions vs providing readymade environment (cloud, container, ..) for learners

Onboarding team leaders / local host

- Onboarding sessions before the workshop : https://coderefinery.github.io/manuals/team-leaders/
  - outline of what will happen during the workshop
  - support
  - Q&A

Onboarding instructors

- This workshop :)
- E-mail / chat communication for coordinating lessons
- One-on-one with instructor coordinator and/or previous instructors

Keypoints

- Here we summarize keypoints.

Collaborative notes

Objectives

- Be able to provide a highly interactive online workshop environment with collaborative documents

Instructor note

- Teaching: 25 min
- Questiongs & Answers: 5 min

Introduction

The Collaborative document is how you interact with the participants. The participants can ask questions and give feedback through the collborative document. During a CodeRefinery session there can be a large a volume of questions. A dedicated person, a HackMD-manager, is needed to answer and edit the collaborative document. Let us see how the collaborative document is used, then discuss the role of the editor or HackMD-manager.

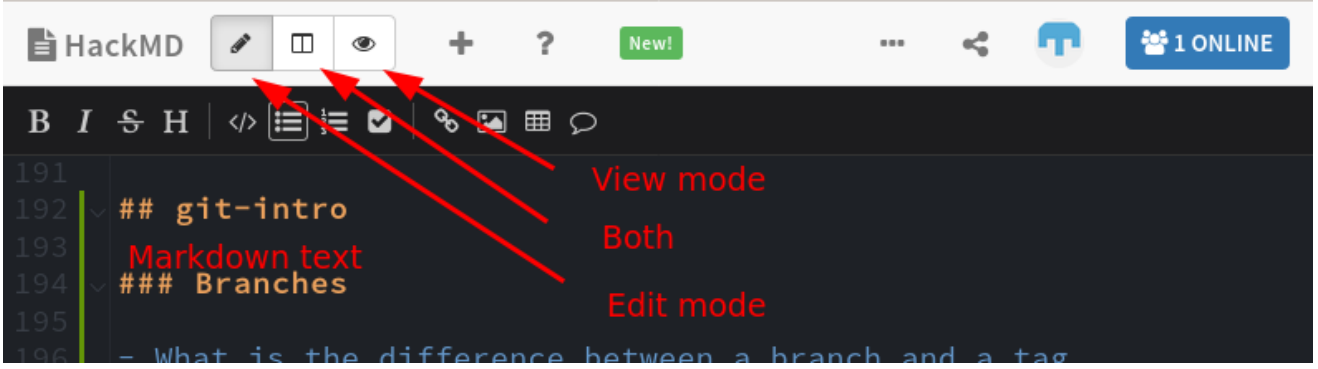
Collaborative document mechanics and controls

Hackmd or HedgeDoc are real-time text editor online. We use it to:

- As a threaded chat, to **answer questions and provide other information** without interrupting the main flow of the room.
- provide everyone with a **more equal opportunity to ask questions**.
- **create notes** which will be archived, for your later reference.

You do not need to login/create an account to be able to edit the document.

Basic controls

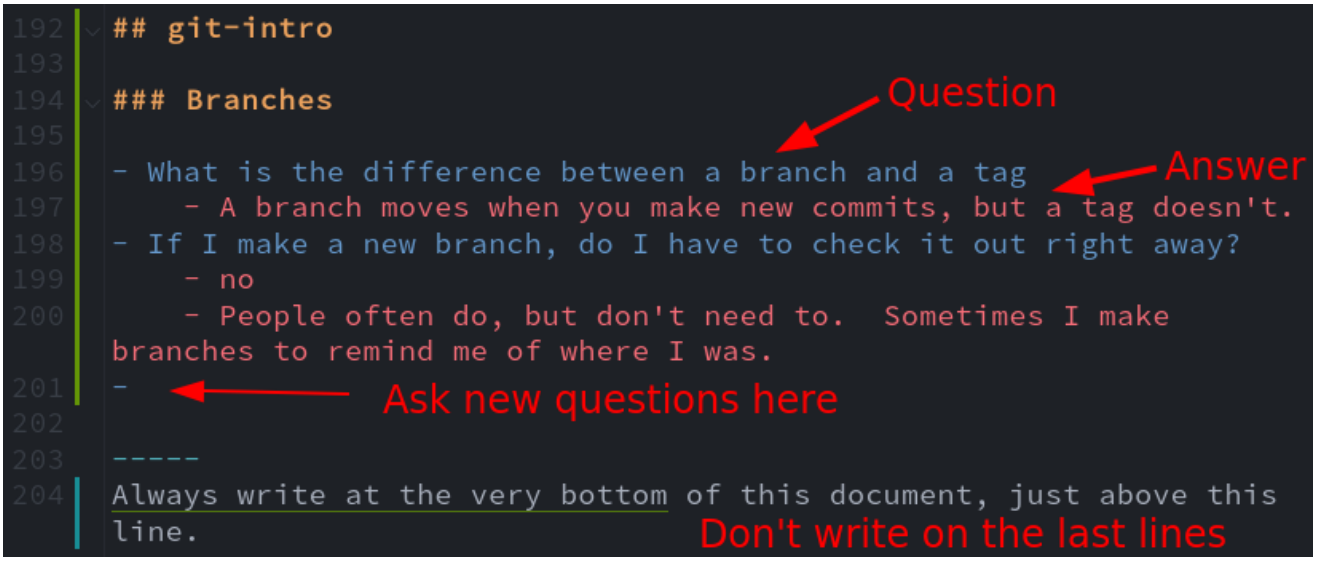


This may look slightly different on mobile devices and small windows.

- At the top (left or right), you can switch between **view**, **edit**, and **split view and edit** modes.
- You write in markdown here. Don't worry about the syntax, just see what others do and try to be like that! Someone will come and fix any problems there may be.
- Please go back to view mode if you think you won't edit for a while - it will still live update.

Asking questions

Always ask questions and add new sections at the very bottom. You can also answer and comment on older questions, too.



Questions and answers in bullet points

Since we plan to publish the questions and answers later as part of the workshop page, we recommend to not use any names. You can indicate your own name to make it easier to discuss more during the workshop but then always use this form: `[name=Myname]`. This makes it easier for us to automatically remove all names before publishing the notes.

Other hints:

- Use `+1` to agree with a statement or question (we are more likely to comment on it).
- Please leave some blank lines at the bottom
- NOTE: Please don't "select all", it highlights for everyone and adds a risk of losing data (there are periodic backups, but not instant).
- It can be quite demanding to follow the collaborative document closely. Keep an eye on it, but consider how distracted you may get from the course. For things beyond the scope of the course, we may come back and answer later.

Don't get overwhelmed

There can be a flood of information on the collaborative document. Scan for what is important, then if you would like come back later. But it is important to keep checking it.

Privacy

- Assume the collaborative document is **public and published: you never need to put your name there**.
- The collaborative document will be **published on the website afterwards**. We will remove all non-instructors names, but it's easier if you don't add it there in the first place.
- Please keep the link private during the workshop, since since security is "editable by those who have the link".
- You can use `[name=YOURNAME]`, to name yourself. We *will* remove all names (but not the comments) before archiving the notes (use this format to make it easy for us).

HackMD manager

We have one person who is a "HackMD helper". This isn't the only person that should edit and answer, but one person shouldn't have too much else on their mind so can focus on it. They also make sure that HackMD is updated with exercise, break, and other meta-information to keep people on track.

Below, (*) = important.

Before the workshop

- Create a new hackmd for the workshop
- make sure that **editing is enabled for anyone without login**
- Add workshop information, links to the workshop page and material and an example question and answer to the top of the hackmd (see below)

Most things to edit (everyone)

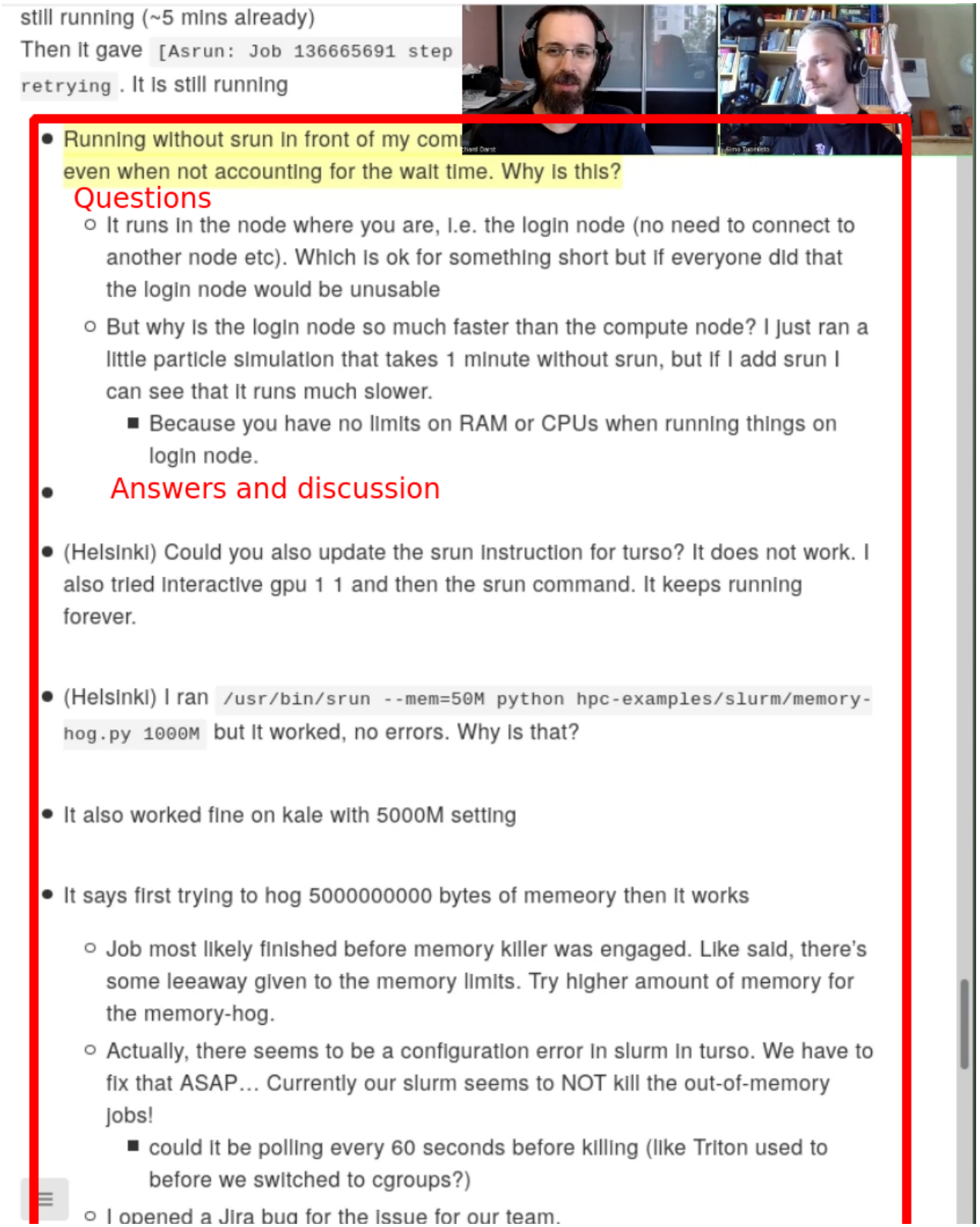
Make it easy to post after the course and consistent to follow:

- Tag all names with `[name=XXX]` (so they can be removed later), remove other personal data or make it obvious.
- Add in information on exercises (new section for them, link, end time, what to accomplish)
- Make a logical section structure (`#` for title, `##` for sections, `###` for episodes, etc. - or what makes sense)

General HackMD practices

Keep it formatted well:

- (*) Tag names you see with `[name=XXX]` so that we can remove it later.
- Heading level `#` is only the page title



A live demo of HackMD during a Q&A time. The two instructors are discussing some of the import answers. Multiple learners have asked questions, multiple answers, and some remaining to be answered

- Add a new `##` heading when a new *lesson* or similar thing is started (introduction, icebreaker, break between lessons, etc)
- Add a new `###` heading when a new *episode*, *exercise*, *break* (within exercise session)
- Ensure people are asking questions at the bottom, direct them there if they aren't.
- (*) Ensure each question is a bullet point. Each answer or follow-up should be a bullet point below.

    - Should you use more deeply nested bullet points, or have only one level below the initial question? It depends on the context, but if a conversation goes on too long, try not to let it go too deep.

Update with meta-talk, so that learners can follow along easily:

- Add Icebreaker and introductory material of the day. Try to talk to people as they joined to get them to open HackMD and answer.
- Anything important for following along should not be only said via voice. It needs to be in the HackMD, too.
- New lessons or episodes, with links to them.
- For exercises, link to exercise and add the duration, end time, goals. If these are unclear, bring it up to the instructor by voice.
- Add a status display about breaks.

Screenshare it when necessary:

- During breaks and other times, share the HackMD (including the notification about break, and when it ends).
- It is nice if the arrangement allows some of the latest questions to be seen, so people are reminded to ask there.
- Someone else may do this, but should make sure it happens.

Answer questions

- If there is an question that should be answered by the instructor by voice, bring it up (by voice) to the instructor immediately.
- How soon do you answer questions? Two points of view:
    - Answer questions right away: can be really intense to follow.
    - Wait some so that we don't overload learners: reduces the info flow. But then do people need to check back more often.
    - You need to find your own balance. Maybe a quick answer right away, and more detailed later. Or delay answers during the most important parts of the lecture.
- Avoid wall-of-text answers. If reading an answer takes too long, it puts the person (and other people who even try to read it) behind even more by taking up valuable mental energy. If an answer needs a wall of text, consider these alternatives:
    - Progressive bullet points getting more detailed (first ones useful alone for basic cases)
    - Don't be worried to say "don't worry about this now, let's talk later."
    - Figure out the root problem instead of answering every possible interpretation
    - Declare it advanced and that you will come back later.

Ensure it can be posted quickly:

- HackMD gets posted to the workshop webpage. For this, it needs some minimal amount of formatting (it doesn't need to be perfect, just not horrible).

- All names and private information needs to be stripped. This is why you should rigorously tag all names with `[name=XXX]` so they can be removed (see above).
  - Learner names can be completely removed. CR staff names can be `[name=CR]` or something similar.
  - There may be other private URLs at the top or bottom.
- If possible, send the PR adding the HackMD to the workshop webpage (though others can do this, too).

HackMD format example

```
# Workshop, day 1


## Lesson name
https://coderefinery.github.io/lesson/

### Episode name
https://coderefinery.github.io/01-episode/

- This is a question
  - Anwser
  - More detailed answer
- question
  - answer

### Exercises:
https://link-to-exercise/.../.../#section
20 minutes, until xx:45
Try to accomplish all of points 1-3.  Parts 4-5 are optional.

Breakout room status:
- room 2, need help with Linux permissions
- room 5, done

### Break
:::danger
We are on a 10 minute break until xx:10
:::


## Lesson 2
https://coderefinery.github.io/lesson-2/
```

Posting HackMD to website

HackMD should be posted sooner rather than later, and hopefully the steps above will make it easy to do so quickly. You could wait a few hours, to allow any remaining questions to be asked an answered.

- Download as markdown
- Remove any private links at the top
- Adjust headings so that they are reasonable
- Look for private info and remove it

- Search document for `[name=???]` (change to `[name=staff]` or `[name=learner]`)
- Any names not tagged with `[name=]`
- usernames in URLs
- private links

Feedback template

```
## Feedback, day N

:::info
### News for day N+1
- .
- .
:::

### Today was (multi-answer):
- too fast:
- just right:
- too slow:
- too easy:
- right level:
- too advanced:
- I would recommend this course to others:
- Exercises were good:
- I would recommend today to others:
- I wouldn't recommend today:

### One good thing about today:
- ...
- ...

### One thing to be improved for next time:
- ...
- ...

### Any other comments:
- ...
- ...
```

Keypoints

- Here we summarize keypoints.

Sound

Objectives

- Be aware of the importance of a well balanced sound quality
- Test tips and tricks for achieving good sound quality

Instructor note

- Teaching: ? min

- Exercises: ? min

Here the episode sections and text ...

Keypoints

- Here we summarize keypoints.

How to prepare a quality screen-share

Objectives

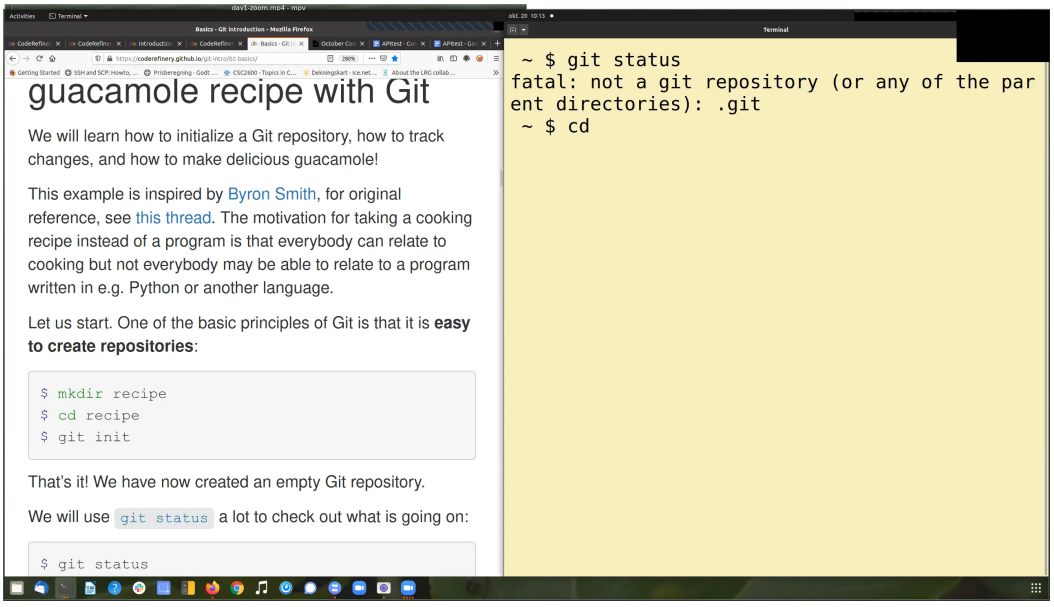
- Discuss the importance of a well planned screen-share.
- Learn how to prepare and how to test your screen-share setup.
- Know about typical pitfalls and habits to avoid.

Instructor note

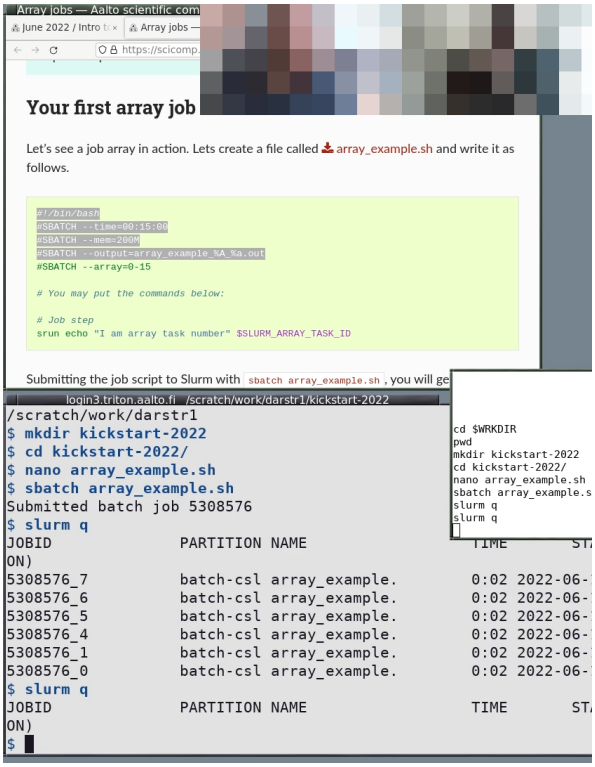
- Discussion: 25 min
- Exercises: 20 min

Share portrait layout instead of sharing entire screen when teaching online

- Many learners will have a smaller screen than you.
- You should plan for learners with **only one small screen**.
- A learner will **need to focus on both your screen share and their work**.
- Share a **portrait/vertical half of your screen** (840 × 1080 is our standard and your maximum).
- Zoom provides a “Share a part of screen” that is good for this.



A FullHD 1920x1080 screen shared. Learners have to make this really small if they want to have something else open.

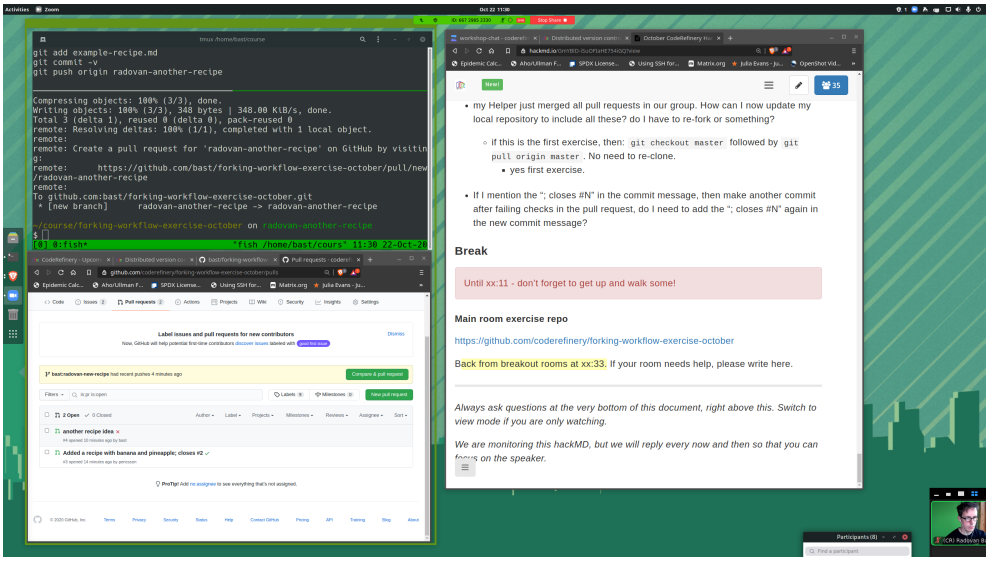


Portrait layout. Allows learners to have something else open in the other half.

Motivation for portrait layout:

- This makes it easier for you to look at some other notes or to coordinate with other instructors at the same time without distracting with too much information.
- This makes it possible for participants to have something else open in the other screen half: terminal or browser or notebook.

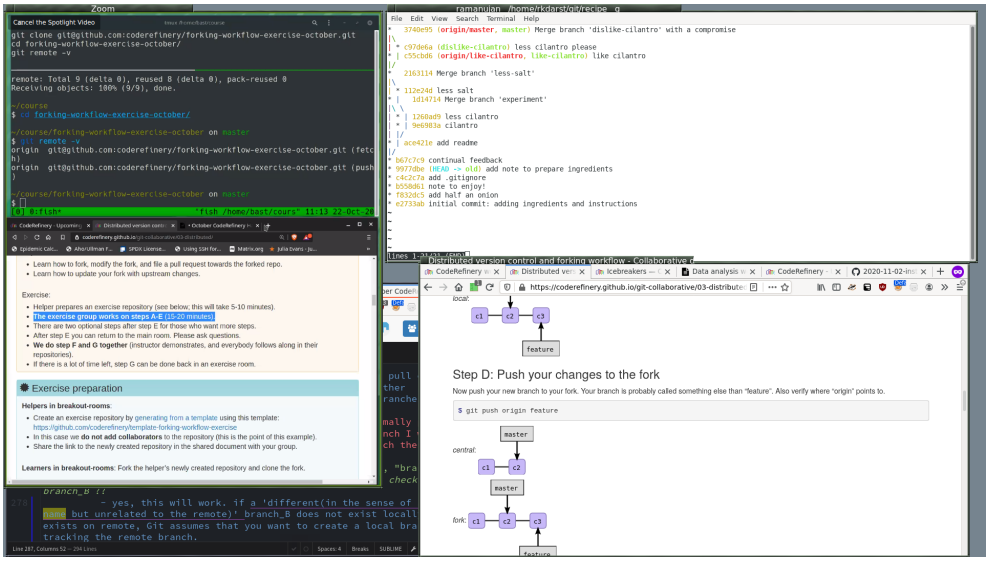
Instructor perspective



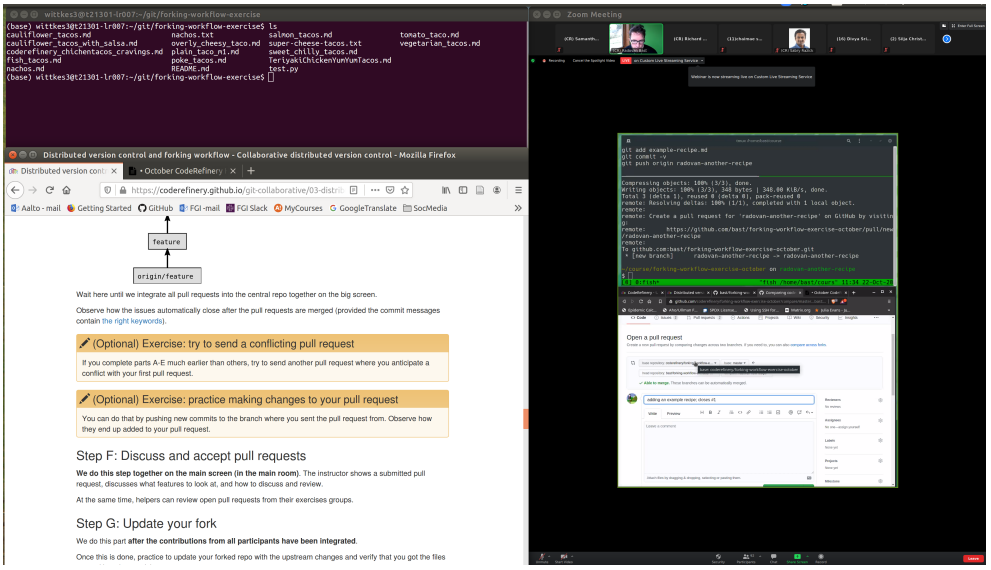
***I1**: This is how it can look for the instructor. Zoom is sharing a portion of the left side, the right side is free for following notes, chat, etc.*

Learner perspective

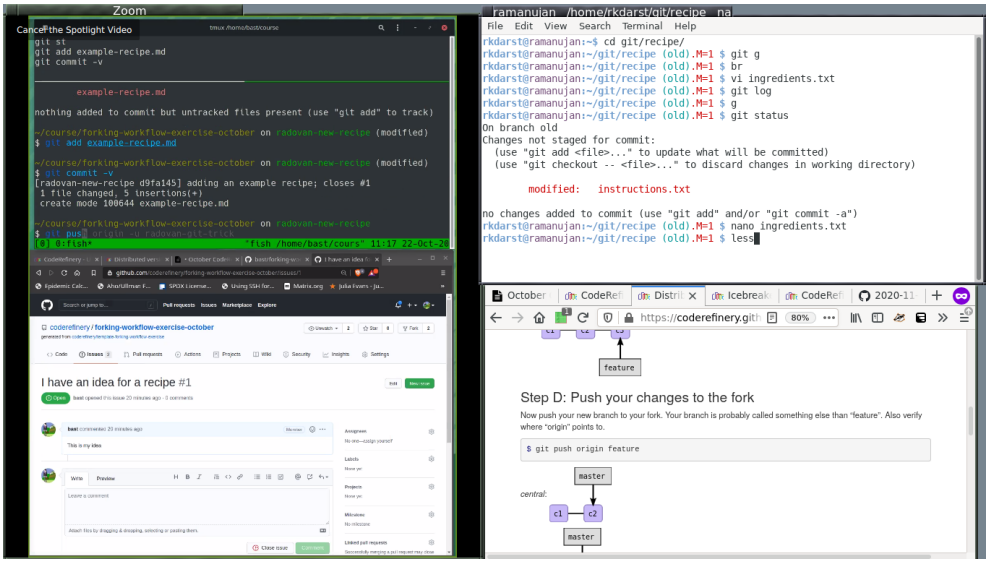
Here are three examples of how it can look for the learner.



***L1**: Learner with a large screen, Zoom in dual-monitor mode so that the instructor pictures are not shown. Screen-share is on the left side, collaborative notes at bottom left, terminal and web browser on the right.*



***L2**: A learner with a single large screen (Zoom in "single monitor mode"). Instructor screen share at right, learner stuff at left.*



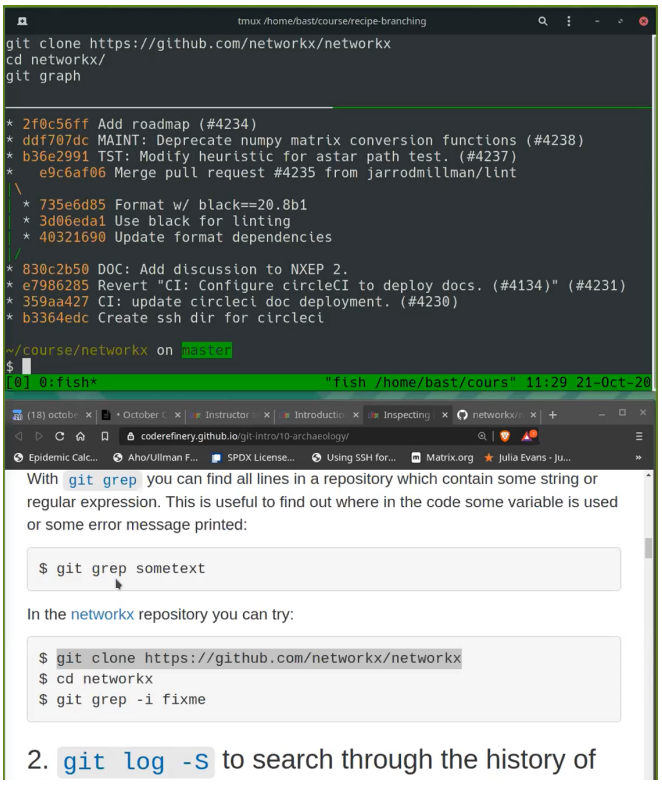
***L3**: A learner with a particularly small screen. Instructor screen-share at left, your windows at right.*

Share the history of your commands

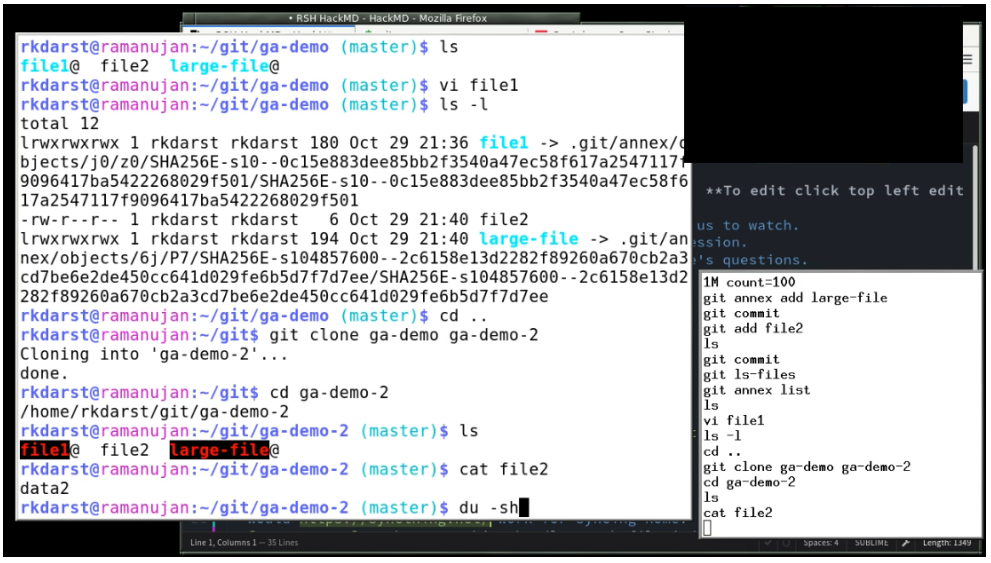
Even if you type slowly it is almost impossible to follow every command. We all get distracted or want to read up on something or while following a command fails on the learner laptop while instructor gets no error.

Add pauses and **share the commands that you have typed** so that one can catch up.

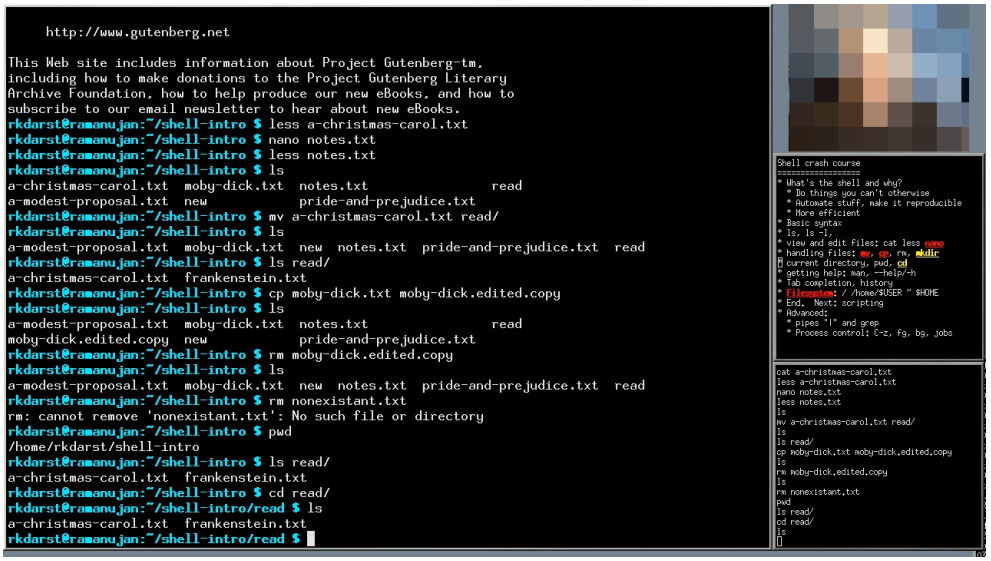
Below are some examples (some more successful than others) of sharing history of commands.



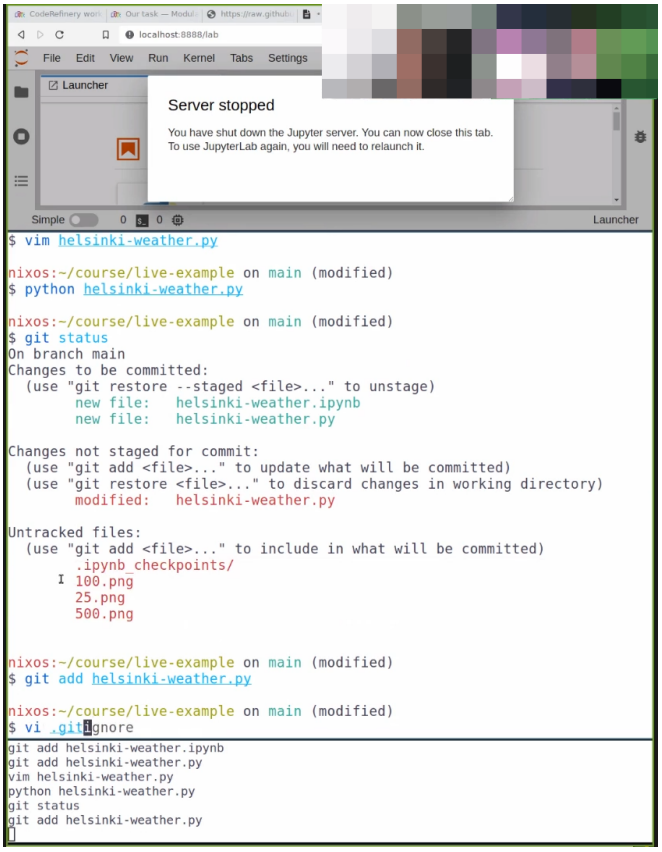
***H1**: A vertical screen layout shared. Note the extra shell history at the top. The web browser is at the bottom, because the Zoom toolbar can cover the bottom bit.*



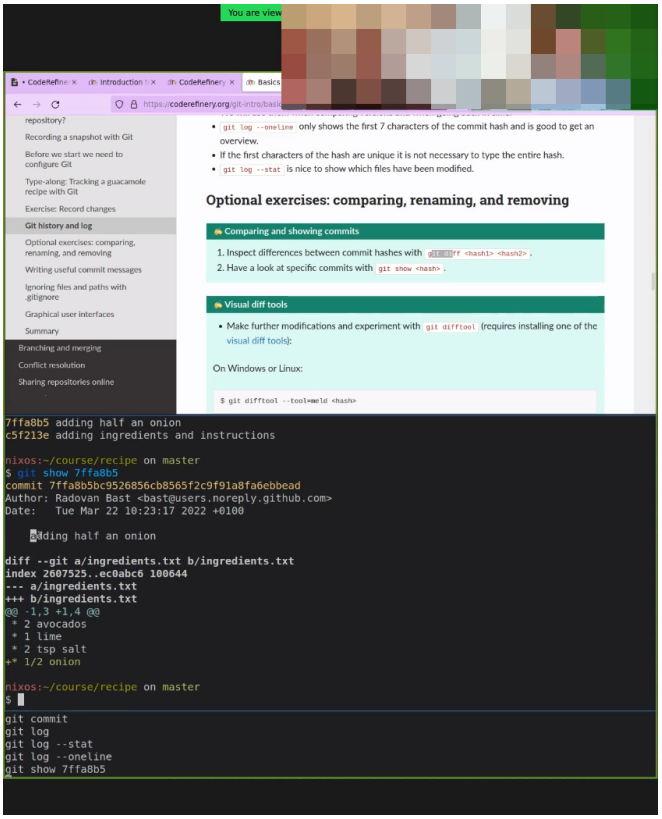
***H2**: This isn't a screen-share from CodeRefinery, but may be instructive. Note the horizontal layout and shell history at the bottom right.*



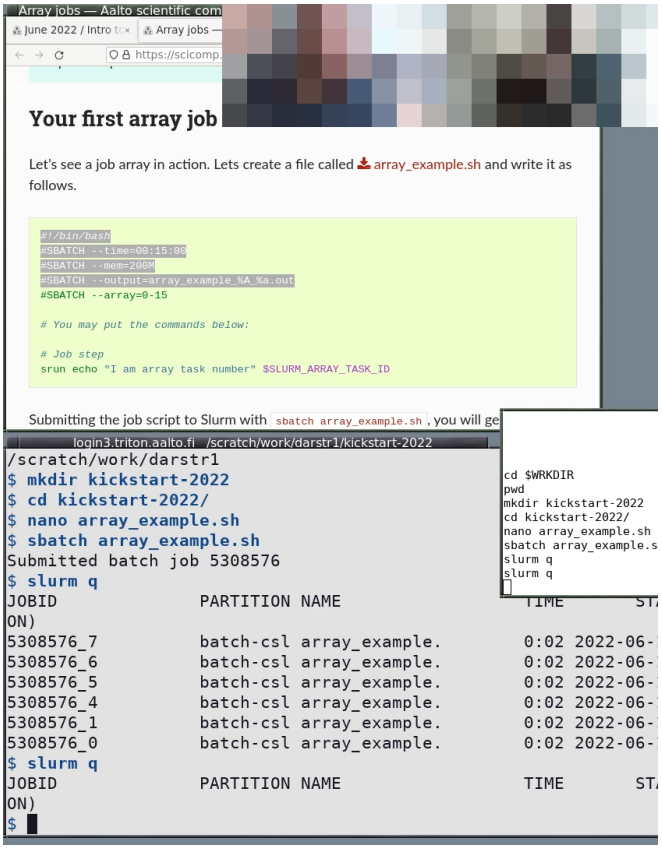
***H3**: Similar to above, but dark. Includes contents on the right.*



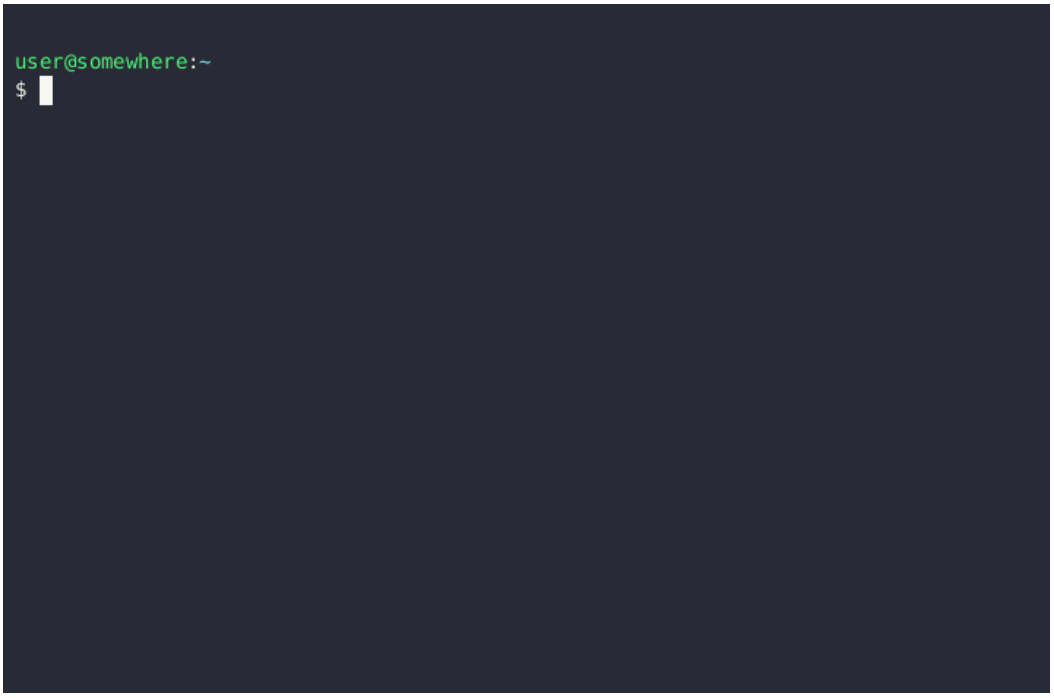
***H4**: Jupyter + terminal, including the `fish` shell and the terminal history.*



H5: Lesson + terminal, `tmux` *plus terminal history and dark background.*



H6: HPC Kickstart course. Note the colors contrast of the windows and colors of the prompt and text. The history is smaller and doesn't take up primary working space. The working directory is in the window title bar.



H7: Show command history "picture-in-picture", in the same terminal window.

How to configure history sharing

You need to find a way to show the recent commands you have entered, so that learners can see the recent commands. Below are many solutions. Try them out and see what works for you.

- prompt-log: It adds a interesting idea that the command you enter is in color and also provides terminal history before the command returns.
- **Simple**: The simple way is `PROMPT_COMMAND="history -a"` and then `tail -f -n0 ~/.bash_history`, but this doesn't capture ssh, sub-shells, and only shows the command after it is completed.
- **Better yet still simple**: Many Software Carpentry instructors use this script, which sets the prompt, splits the terminal window using tmux and displays command history in the upper panel. Requirement: tmux
- **Better (bash)**: This prints the output before the command is run, instead of after. Tail with `tail -f ~/demos.out`.

```
BASH_LOG=~/demos.out
bash_log_commands () {
    # https://superuser.com/questions/175799
    [ -n "$COMP_LINE" ] && return  # do nothing if completing
    [[ "$PROMPT_COMMAND" =~ "$BASH_COMMAND" ]] && return # don't cause a preexec
for $PROMPT_COMMAND
    local this_command=`HISTTIMEFORMAT= history 1 | sed -e "s/^[ ]*[0-9]*[ ]*//"`;
    echo "$this_command" >> "$BASH_LOG"
}
trap 'bash_log_commands' DEBUG
```

- **Better (zsh)**: This works like above, with zsh. Tail with `tail -f ~/demos.out`.

```
preexec() { echo $1 >> ~/demos.out }
```

- **Better (fish)**: This works like above, but for fish. Tail with `tail -f ~/demos.out`.

```
function cmd_log --on-event fish_preexec ; echo "$argv" >> ~/demos.out  ; end
```

- **Better (tmuxp)**: This will save some typing. TmuxP is a Python program (`pip install tmuxp`) that gives you programmable `tmux` sessions. One configuration that works (in this case for `fish` shell):

```
session_name: demo
windows:
  - window_name: demo
    layout: main-horizontal
    options:
      main-pane-height: 7
    panes:
      - shell_command:
          - touch /tmp/demo.history
          - tail -f /tmp/demo.history
      - shell_command:
          - function cmd_log --on-event fish_preexec ; echo "$argv" >>
/tmp/demo.history  ; end
```

- **Windows PowerShell**: In Windows Terminal, a split can be made by pressing `CTRL+SHIFT+=`. Then, in one of the splits, the following PowerShell command will start tracking the shell history:

```
Get-Content (Get-PSReadlineOption).HistorySavePath -Wait
```

  Unfortunately, this only shows commands after they have been executed.

- Tavatar: shell history mirroring teaching tool can copy recent history to a remote server.
- history-window: Show command history "picture-in-picture" when teaching command line. Requires Bash.

Font, colors, and prompt

Terminal color schemes

- Dark text on light background, *not* dark theme. Research and our experience says that dark-text-on-light is better in some cases and similar in others.
- You might want to make the background light grey, to avoid over-saturating people's eyes and provide some contrast to the pure white web browser. (this was an accessibility recommendation when looking for ideal color schemes)
- Do you have any yellows or reds in your prompt or program outputs? Adjust colors if possible.

Font size

- Font should be large (a separate history terminal can have a smaller font).
- Be prepared to resize the terminal and font as needed. Find out the keyboard shortcuts to do this since you will need it.

Prompt

At the beginning of the workshop your goal is to have a shell **as easy to follow as possible** and **as close to what learners will see on their screens**:

- Your prompt should be minimal: few distractions, and not take up many columns of text.
- prompt-log does this for you.
- The minimum to do is is `export PS1='\$ '`.
- Blank line between entries: `export PS1='\n\$ '`.
- Have a space after the `$` or `%` or whatever prompt character you use.
- Strongly consider the Bash shell. This is what most new people will use, and Bash will be less confusing to them.
- Eliminate menu bars and any other decoration that uses valuable screen space.
- Add colors only if it simplifies the reading of the prompt.

Later in the workshop or in more advanced lessons:

- Using other shells and being more adventurous is OK - learners will know what is essential to the terminal and what is extra for your environment.

Try to find a good balance between:

- Showing a simple setup and showing a more realistic setup.
- Showing a consistent setup among all instructors and showing a variety of setups.

Habits we need to un-learn

- **Do not clear the terminal**. Un-learn CTRL-L or `clear` if possible. More people will wonder what just got lost than are helped by seeing a blank screen. Push `ENTER` a few times instead to add some white space.
- **Do not rapidly switch between windows** or navigate quickly between multiple terminals, browser tabs, etc. This is useful during your own work when nobody is watching, but it is very hard to follow for learners.
- Avoid using **aliases** or **shortcuts** that are not part of the standard setup. Learners probably don't have them, and they will fail if they try to follow your typing. Consider even to rename corresponding files (`.bashrc` , `.gitconfig` , `.ssh/config` , `.ssh/authorized_keys` , `.conda/*`).
- Be careful about using **tab completion** or **reverse history search** if these haven't been introduced yet.

Desktop environment and browser

- Try to remove window title bars if they take up lots of space without adding value to the learner.
- Can you easily resize your windows for adjusting during teaching?
- Does your web browser have a way to reduce its menu bars and other decoration size?
  - Firefox-based browsers: go to `about:config` and set `layout.css.devPixelsPerPx` to a value slightly smaller than one, like `0.75`. Be careful you don't set it too small or large since it might be hard to recover! When you set it to something smaller than 1, all window decorations become smaller, and you compensate by zooming in on the website more (you can set the default zoom to be greater than 100% to compensate). Overall, you get more information and less distraction.

How to switch between teaching setup and work setup?

- Make a dedicated "demos" profile in your terminal emulator, if relevant. Or use a different terminal emulator just for demos.
- Same idea for the browser: Consider using a different browser profile for teaching/demos.
- Another idea is to containerize the setup for teaching. We might demonstrate this during the Cool gems session later.

❗ Keypoints

- Share **portrait layout** instead of sharing entire screen
- **Adjust your prompt** to make commands easy to read
- **Readability** and beauty is important: adjust your setup for your audience
- **Share the history** of your commands
- Get set up a **few days in advance** and get feedback from someone else. Feedback and time to improve is very important to make things clear and accessible. 10 minutes before the session starts is typically too late.

Exercises

✍️ Evaluate screen captures (20 min)

Evaluate screenshots on this page. Discuss the trade-offs of each one. Which one do you prefer? Which are useful in each situation?

Please take notes in the collaborative document.

✍️ Set up your own environment (20 min)

Set up your screen to teach something. Get some feedback from another learner or your exercise group.

Other resources

- https://coderefinery.github.io/manuals/instructor-tech-setup/
- https://coderefinery.github.io/manuals/instructor-tech-online/

Computational thinking

Objectives

- Explain what is computational thinking
- Get to know how the theory of computational thinking can be used in teaching
- Short exercise

Instructor note

- Teaching: 20 min
- Exercises: 10 min

Here the episode sections and text ...

Keypoints

- Computational Thinking consists of 4 main parts: decomposition, pattern recognition, abstraction and algorithmic design.
- How can this be a useful framework for solving problems?
- How can this be used practically?

Teaching philosophies

Objectives

- Get to know the teaching philosophies of CodeRefinery instructors

Instructor note

- Teaching: ? min
- Exercises: ? min

Here the episode sections and text ...

Keypoints

- Here we summarize keypoints.

Co-teaching

Objectives

- Get to know the principle of co-teaching: How we do it and how you can too.

Instructor note

- Teaching: ? min
- Exercises: ? min

Why teach together?

It has been said **a lot**, especially in areas such as code development or scientific research, about the value of collaboration. Yet still today, the effort of teaching is made alone far too often: a person decides to share their knowledge (or gets assigned a study module) and starts building the actual teaching material basically from scratch. It seems much more logical, in the age of FAIR science and open knowledge, to release, develop, iterate, and maintain teaching material – including the contact sessions – **collaboratively** as well.

Ways to teach together

- Develop materials together - avoid duplication.
- Present the materials together ("proper" co-teaching, see Team teaching section on the CR manual).
- Use helpers extensively to tackle specific tasks commonly arising in online teaching process.
- Involve your learners too, e.g. using collaborative document (such as HackMD) for parallel and mass answers.

Advantages

- If you need to teach anyway, combined efforts take up less time.
- More engaging to the audience, taking some of the (sometimes daunting) expectation to "speak up" off of the students.
- Easier on-boarding of new instructors – one of them can be learning at the same time, either subtleties of the material or the teaching itself.
- Swiss-cheese principle: two "imperfect" teachers are **much** easier to find and complement each other than the extensively-prepared, absolute expert.

Challenges

- Additional effort needed of teacher and/or helper coordination – including syncing up their schedules!
- Materials might need to be (hopefully slightly) tuned to a specific target audience.
- Using simultaneous-teaching strategies is a learned skill, not identical to the classical lecturing.
- Online tools (HackMD, type-alongs) can potentially overload learners and teachers alike, if not used with care.

✍ Exercise

(What's better here – practical exercise or discussion?)

(TODO: Here goes the rest of the episode sections and text)

Keypoints

- Here we summarize keypoints.

Cool gems

Objectives

- Here we will list learning objectives

Instructor note

- Teaching: ? min
- Exercises: ? min

Here the episode sections and text ...

Keypoints

- Here we summarize keypoints.

Why we stream

Objectives

- Learn the general history of CodeRefinery streaming.
- Discuss the benefits of streaming and recording
- Discuss the downsides and difficulties

Instructor note

- Discussion: 10 min
- Q&A: 5 min
- Exercises: 0 min

This is a general discussion of the topics of the day, focused on the history of why we stream and record, how we got here, and how people feel about it. We won't focus on how anything is done.

What is streaming and recording?

- Streaming is mass communication: one to many
  - Interaction audience→presenters is more limited (but different)
- Using consumer-grade tools, normal people can reach huge audiences by Twitch/YouTube/etc.
- This isn't actually that hard: people with much less training than us do it all the time.
- They reach huge audiences and maintain engagement for long events.

Recording and rapid video editing is useful even without streaming.

How did we get to the current state

- In-person workshops
  - 3 × full day, required travel, infrequent, one-shot
- Covid and remote teaching
  - Traditional "Zoom" teaching several times
- Mega-CodeRefinery workshop
  - 100-person Zoom teaching
  - Emphasis on teams
- Research Software Hour
  - Livestream free-form discussions on Twitch
- Streamed "HPC Kickstart" courses

Benefits and disadvantages

Benefits:

- Larger size gives more (but different) interaction possibility
  - "Notes" for async Q&A
- Recording (with no privacy risk) allows instant reviews
- Stream-scale allows for many of the things you have learned about in days 1-3.

Disadvantages:

- Requires training for using the tools
- Requires a certain scale to be worth it
- Coordination is much harder for big events

Future prospects (briefly)

- Streaming probably stays as a CodeRefinery tool
- We *can* scale our courses much larger than they are now. Why don't we, together with others?

Q&A

Keypoints

- Streaming isn't that hard to understand
- There are benefits and disadvantages

Behind the stream

Objectives

- Get to know what happens "behind the stream" of a workshop
- See what the "broadcaster" sees and what they need to do.

Instructor note

- Teaching: 20 min
- Q&A 10 min

In this episode, you'll see an end-to-end view of streaming from the broadcaster's point of view. It's a tour but not an explanation or tutorial.

Who does what

We have certain role definitions:

- **Broadcaster**: Our term for the person who manages the streaming.
- **Director**: Person who is guiding the instructors to their sessions, changing the scenes, calling the breaks, etc.
  - Could be the same as broadcaster.
- **Instructor**: One who is teaching. They don't have to know anything else about how streaming works.

This lesson describes what the Broadcaster/Director sees.

Window layouts

What does the broadcaster see on their screen?

- What are the main windows you see?
- What do each of them do?
- Which ones do you need to focus on?
- How do you keep all this straight in your head?

How scenes are controlled

What has to be done during a course?

- How do you start the stream?
- How do you change the view?
- How do you adjust things based on what the instructors share?
- How do you coordinate with the instructors?
- How do you know when to change the view?

Getting it set up

- How hard was it to figure this out?
- How hard is it to set it up for each new workshop?

What can go wrong

- What's the worst that has happened?
- What if you need to walk away for a bit?

Q&A

Q&A from audience

Keypoints

- The broadcaster's view shouldn't be so scary.
- There is a lot to manage, but each individual part isn't that hard.

Video editing

Objectives

- Get to know ways of quick video editing to be able to provide accessible videos
- Learn how video editing can be distributed and done the same day.

Instructor note

- Teaching: 20 min
- Exercises: 20 min

Video recordings could be useful for people watching later, but also are (perhaps more) useful for **immediate review and catching up after missing a day in a workshop**. For this, they need to be released immediately, within a few hours of the workshop. CodeRefinery does this, and you can too.

Hypothesis: videos must be processed the same evening as they were recorded, otherwise (it may never happen) or (it's too late to be useful). To do that, we have to make processing *good enough* (not perfect) and *fast* and *distributeable*.

Primary articles

- Video editor role description: https://coderefinery.github.io/manuals/video-editor/
- ffmpeg-editlist: the primary tool: https://github.com/coderefinery/ffmpeg-editlist
  - Example YAML editlists: https://github.com/AaltoSciComp/video-editlists-asc

Summary

- Basic principle: privacy is more important than any other factor. If we can't guarantee privacy, we can't release videos at all.
  - Disclaimers such as "if you don't want to appear in a recording, leave your video off and don't say anything", since a) accidents happen especially when coming back from breakout rooms. b) it creates an incentive to not interact or participate in the course.
- Livestreaming is important here: by separating the instruction from the audience audio/video, there is no privacy risk in the raw recording. They could be released or shared unprocessed.
- Our overall priorities
  1. No learner (or anyone not staff) video, audio, names, etc. are present in the recordings.
  2. Good descriptions.
  3. Removing breaks and other dead time.
  4. Splitting videos into useful chunks (e.g. per-episode), perhaps equal with the next one:
  5. Good Table of Contents information so learners can jump to the right spots (this also helps with "good description".)
- ffmpeg-editlist allows us to define an edit in a text file (crowdsourceable on Github), and then generate videos very quickly.

How we do it

The full explanation is in the form of the exercises below. As a summary:

- Record raw video (if from a stream, audience can't possibly be in it)
- Run Whisper to get good-enough subtitles. Distribute to someone for checking and improving.
- Define the editing steps (which segments become which videos and their descriptions) in a YAML file.
- Run ffmpeg-editlist, which takes combines the previous three steps into final videos.

Exercises

Exercise A

These exercises will take you through the whole sequence.

✍ Editing-1: Get your sample video

Download a sample video:

- Video (raw): http://users.aalto.fi/~darstr1/sample-video/ffmpeg-editlist-demo-kickstart-2023.mkv
- Whisper subtitles (of raw video): http://users.aalto.fi/~darstr1/sample-video/ffmpeg-editlist-demo-kickstart-2023.srt
- Schedule of workshop (day 1, 11:35–12:25) - used for making the descriptions. :::::

✍ Editing-2: Run Whisper to generate raw subtitles and test video.

First off, install Whisper and generate the base subtitles, based on the. Since this is probably too much to expect for a short lesson, they are provided for you (above), but if you want you can try using Whisper, or generating the subtitles some other way.

You can start generating subtitles now, while you do the next steps, so that they are ready by the time you are ready to apply the editlist. ffmpeg-editlist can also slice up the subtitles from the main video to make subtitles for each segment you cut out.

Whisper is left as an exercise to the reader.

✔ Solution

Example Whisper command:

```
$ whisper --device cuda --output_format srt --initial_prompt="Welcome to
CodeRefinery day four." --lang en --condition_on_previous_text False
INPUT.mkv
```

An initial prompt like this make Whisper more likely to output full sentences, instead of a stream of words with no punctuation.

✍️ Editing-3: Create the basic editlist.yaml file

Install ffmpeg-editlist and try to follow its instructions, to create an edit with these features:

- The input definition.
- Two output sections: the “Intro to the course”, and “From data storage to your science” talks (Remember it said the recording started at 11:35… look at the schedule for hints on when it might start!). This should have a start/end timestamp from the *original* video.

A basic example:

```
- input: day1-raw.mkv

# This is the output from one section.  Your result should have two of these
sections.
- output: part1.mkv
  title: something
  description: >-
    some long
    description of the
    segment
  editlist:
    - start: 10:00    # start timestamp of the section, in *original* video
    - end: 20:00      # end timestamp of the section, in the *original* video
```

✔ Solution

This is an excerpt from our actual editlist file of this course

```yaml
- input: day1-obs.mkv

- output: day1-intro.mkv
  title: 1.2 Introduction
  description: >-
    General introduction to the workshop.

    https://scicomp.aalto.fi/training/kickstart/intro/

  editlist:
  - start: 00:24:10
  - end: 00:37:31


- output: day1-from-data-storage-to-your-science.mkv
  title: "1.3 From data storage to your science"
  description: >-
    Data is how most computational work starts, whether it is
    externally collected, simulation code, or generated. And these
    days, you can work on data even remotely, and these workflows
    aren't obvious. We discuss how data storage choices lead to
    computational workflows.

    https://hackmd.io/@AaltoSciComp/SciCompIntro

  editlist:
  - start: 00:37:43
  - end: 00:50:05
```

💬 Discussion: what makes a video easy to edit?

- Clear speaking and have high audio quality.
- For subtitle generation: Separate sentences cleanly, otherwise it gets in a “stream of words” instead of “punctuated sentences” mode.
- Clearly screen-sharing the place you are at, including section name.
- Clear transitions, “OK, now let’s move on to the next lesson, LESSON-NAME. Going back to the main page, we see it here.”
- Clearly indicate where the transitions are
- Hover mouse cursor over the area you are currently talking about.
- Scroll screen when you move on to a new topic.
- Accurate course webpage and sticking to the schedule

All of these are also good for learners. By editing videos, you become an advocate for good teaching overall.

✍️ Editing-4: Run ffmpeg-editlist

Install ffmpeg-editlist: `pip install ffmpeg-editlist[srt]` (you may want to use a virtual environment, but these are very minimal dependencies).

The `ffmpeg` command line tool must be available in your `PATH`.

✓ Solution

It can be run with (where `.` is the directory containing the input files):

```
$ ffmpeg-editlist editlist.yaml .
```

Just running like this is quick and works, but the stream may be garbled in the first few seconds (because it's missing a key frame). (A future exercise will go over fixing this. Basically, add the `--reencode` option, which re-encodes the video (this is **slow**). Don't do it yet.

Look at the `.info.txt` files that come out.

✍ Editing-5: Add more features

- Several chapter definitions.(re-run and you should see a `.info.txt` file also generated). Video chapter definitions are timestamps of the *original* video, that get translated to timestamps of the *output* video.

  ```
  - output: part1.mkv
    editlist:
    - start: 10:00
    - -: Introduction    #  <-- New, `-` means "at start time"
    - 10:45: Part 1      #  <-- New
    - 15:00: Part 2      #  <-- New
    - end: 20:00
  ```

  Look at the `.info.txt` files that come out now. What is new in it?

- Add in "workshop title", "workshop description", and see the `.info.txt` files that come out now. This is ready for copy-pasting into a YouTube description (first line is the title, rest is the description).

  Look at the new `.info.txt` files. What is new?

✓ Solution

- This course actually didn't have chapters for the first day sessions, but you can see chapters for day 2 here, for example.
- Example of the workshop description for this course
- Example info.txt file for the general introduction to the course. The part after the `-----` is the workshop description.

```
1.2 Introduction - HPC/SciComp Kickstart summer 2023

General introduction to the workshop.

https://scicomp.aalto.fi/training/kickstart/intro/

00:00 Begin introduction       <-- Invented for the exercise demo, not real
03:25 Ways to attend           <-- Invented for the exercise demo, not real
07:12 What if you get lost     <-- Invented for the exercise demo, not real

-----

This is part of the Aalto Scientific Computing "Getting
started with Scientific Computing and HPC Kickstart" 2023
workshop.  The videos are available to everyone, but may be
most useful to the people who attended the workshop and want
to review later.

Playlist:
https://www.youtube.com/playlist?list=PLZLVmS9rf3nMKR2jMglaN4su3ojWtWMVw

Workshop webpage:
https://scicomp.aalto.fi/training/scip/kickstart-2023/

Aalto Scientific Computing: https://scicomp.aalto.fi/
```

✍️ Editing-6: Subtitles

Re-run ffmpeg-editlist with the `--srt` option (you have to install it with `pip install ffmpeg-editlist[srt]` to pull in the necessary dependency). Notice how `.srt` files come out now.

Use some subtitle editor to edit the *original* subtitle file, to fix up any transcription mistakes you may find. You could edit directly, use `subtitle-editor` on Linux, or find some other tool.

What do you learn from editing the subtitles?

✔ Solution

```
$ ffmpeg-editlist --srt editlist.yaml
```

There should now be a `.srt` file also generated. It generated by finding the `.srt` of the original video, and cutting it the same way it cuts the video. Look and you see it aligns with the original.

This means that someone could have been working on fixing the Whisper subtitles while someone else was doing the yaml-editing.

✍️ Editing-6: Subtitles

Re-run ffmpeg-editlist with the `--srt` option (you have to install it with `pip install ffmpeg-editlist[srt]` to pull in the necessary dependency). Notice how `.srt` files come out now.

Use some subtitle editor to edit the *original* subtitle file, to fix up any transcription mistakes you may find. You could edit directly, use `subtitle-editor` on Linux, or find some other tool.

What do you learn from editing the subtitles?

✍️ Editing-7: Generate the final output file.

- Run ffmpeg-editlist with the `--reencode` option: this re-encodes the video and makes sure that there is no black point at the start.
- If you re-run with `--check`, it won't output a new video file, but it *will* re-output the `.info.txt` and `.srt` files. This is useful when you adjust descriptions or chapters.

💬 Discussion: how to distribute this?

Create a flowchat of all the parts that need to be done, and which parts can be done in parallel. Don't forget things that you might need to do before the workshop starts.

How hard was this editing? Was it worth it?

Exercise B

This is a more limited (and older) version of the above exercise, using an synthetic example video.

✍️ Use ffmpeg-editlist to edit this sample video

Prerequisites: `ffmpeg` must be installed on your computer outside of Python. Be able to install ffmpeg-editlist. This is simple in a Python virtual environment, but if not the only dependency is `PyYAML`.

- Download the sample video: http://users.aalto.fi/~darstr1/sample-video/sample-video-to-edit.raw.mkv
- Copy a sample editlist YAML
- Modify it to cut out the dead time at the beginning and the end.

- If desired, add a description and table-of-contents to the video.
- Run ffmpeg-editlist to produce a processed video.

✓ Solution

```
- input: sample-video-to-edit.raw.mkv
- output: sample-video-to-edit.processed.mkv
  description: >
  editlist:
    - start: 00:16
    - 00:15: demonstration
    - 00:20: discussion
    - stop: 00:25
```

```
$ ffmpeg-editlist editlist.yaml video/ -o video/
```

Along with the processed video, we get `sample-video-to-edit.processed.mkv.info.txt` ::

```
This is a sample video


00:00 Demonstration
00:04 Discussion
```

See also

- ffmpeg-editlist demo: https://www.youtube.com/watch?v=thvMNTBJg2Y
- Full demo of producing videos (everything in these exercises): https://www.youtube.com/watch?v=_CoBNe-n2Ak
- Example YAML editlists: https://github.com/AaltoSciComp/video-editlists-asc

Keypoints

- Video editing is very useful for learning
- Set your time budget and make it good enough in that time
- Reviewing videos improves your teaching, too.

Open Broadcaster Software (OBS) introduction

Objectives

- Understand that OBS is a video mixer.

- Understand the basic controls and features of OBS.

Instructor note

- Teaching: 15 min
- Q&A 5 min

- OBS theory in CodeRefinery manuals
- The next episode Open Broadcaster Software (OBS) setup

In this episode, you'll get more familiar with the OBS portion of the streaming setup. You'll see how it's used, but not yet how to configure it from scratch. You'll learn how to be a "director".

What is OBS?

- Formally "OBS Studio"
- Cross-platform, easy to use screencasting and streaming app.
- Real-time video mixer.

OBS user interface

- What does each view do?
- Let's click through the buttons
- Let's see the important config options

OBS during a course

- What management is needed
- The control panel

Hardware requirements

See also

- The next episode Open Broadcaster Software (OBS) setup which is about configuring OBS.

Keypoints

- OBS may seem complicated, but it's a graphical application and most pieces make sense once you know how it works.

Open Broadcaster Software (OBS) setup

Objectives

- See how to configure OBS using the pre-made CodeRefinery scene collections
- Modify the collections to suit your needs.

Instructor note

- Teaching: ?? min
- Hands-on: ?? min
- Q&A: ?? min

- The previous episode Open Broadcaster Software (OBS) introduction
- OBS theory in CodeRefinery manuals

CodeRefinery OBS configs

- https://github.com/coderefinery/obs-config

Installing the OBS config

Initial setup

Setup before each course

Keypoints

- OBS may seem complicated, but it's a graphical application and most pieces make sense once you know how it works.

What's next?

Objectives

- Know next steps if you want to do streaming

Instructor note

- Teaching: 5 min
- Q&A 5 min

What comes next?

- We talked a lot about theory, and gave demonstrations.
- Hands-on is very different. We recommend working with someone to put it in practice.
-

- Work with someone who can show you the way
- Use it for smaller courses with a backup plan

See also

Keypoints

- These lessons about streaming have been the theoretical part of streaming training.

Instructor guide

Target audience

- Everyone teaching online workshops about computational topics or interested in teaching.
- Previous and future instructors and helpers of CodeRefinery workshops.

Timing

Session 1

- 15 min: Intro
- 45 min: Lesson design and development
- 15 min: Break
- 45 min: Lesson template
- 15 min: Break
- 30 min: How we collect feedback and measure impact

Session 2

- 15 min: Intro
- 45 min: Workshop overview, roles, onboarding/installation, helpers
- 15 min: Break
- 30 min: Collaborative notes and interaction
- 15 min: Sound
- 15 min: Break
- 45 min: Screenshare

Session 3

- 15 min: Intro

- 30 min: Computational thinking
- 15 min: Break
- 30 min: Co-teaching
- 30 min: Teaching philosohies
- 15 min: Break
- 45 min: Sharing teaching tips, tricks, tools etc

Session 4

- 15 min: Intro
- 45 min: ..
- 15 min: Break
- 45 min: ..
- 15 min: Break
- 45 min: ..