User Documentation

computor – manual for learners, tutors and lecturers

computor – netidee project 8012, TU Graz
Documentation of the current version · Licence CC BY 4.0

Overview **3**
What it is 3
Who it is for and how it helps 3
How it works 3
How this documentation is organised 3
Part A: For students **4**
Courses 4
Units and Readings 7
Assignments 8
Learn with Computor 11
Part B: For tutors and lecturers **13**
The Tutor View 13
Grading an Assignment 14
Teacher guide: course dashboard 16
Part C: Creating courses and assignments **20**
Computor Testing Framework - User Guide 20

Overview

What it is

computor is open-source software for programming courses. Students work on exercises in Visual Studio Code, either in the browser on computor.at or locally on their own computer, and have their solution checked by automated tests. For questions there is the AI tutor Lena, who is instructed to give hints rather than finished solutions. Lecturers and tutors see their students' progress, test attempts and common errors, and they grade submissions. This guide explains how to use computor in all three roles.

Who it is for and how it helps

Students get prompt feedback after every test run on whether their program does what was asked. Anyone who wants to try computor outside a course can read the three public courses of the "Data Science mit Python" series at computor.at/learn without an account, and work on them after registering. For tutors and lecturers there is no collecting of submissions, the automated tests take over part of the checking, and they can see early where students are struggling. Lecturers who want to set up their own courses will find in the third part of this documentation how exercises and their automated tests are described.

How it works

Every student gets their own version-controlled store (a Git repository) for each course, in which each assignment is a folder. Students open an assignment in the Computor view of VS Code, write their code and start a test run, which uploads the current state and checks it on the server. The test result appears right next to the assignment. To have their solution graded, students submit it explicitly in a separate step. Tutors open the same assignment in their tutor view, see the code, test results and messages, and give a grade. Lecturers find course statistics and the typical errors for each assignment in the computor web application.

How this documentation is organised

Part A describes working as a student, Part B working as a tutor or lecturer, and Part C how to create assignments. The complete reference manual of the VS Code extension is in the repository. The help pages for students and tutors are also available in the extension itself through the command **Computor: Welcome and Help** (German interface: *Willkommen und Hilfe*), which you run from the VS Code Command Palette (Ctrl+Shift+P). This documentation is also available in German. The guides describe the current version and are maintained together with the code in the repositories computor-vscode, computor-backend and data-science-python. This documentation is licensed under CC BY 4.0.

Part A: For students

Courses

Source: https://github.com/computor-org/computor-vscode/blob/main/docs/help/student-course.md

A course is everything you are enrolled in for one class: its units, its assignments, and the repository your work lives in. It is the top row in the Computor view, and most of what this page describes applies everywhere below it.

```
📖 Programming in Physics
├── ⚫ Week 1 — Introduction
│   ├── 🟦 Lab 1
│   └── 🟦 Homework 1
└── 🟦 Final Project
```

Click a row to expand it. The structure comes from your lecturer; you cannot change it.

Your work lives in one repository

Every assignment of a course is a folder in a **single repository** that belongs to you. This is the thing to understand first, because almost everything else follows from it:

- Saving a file changes it **on your computer only**.
- **Committing** records your changes in the repository and uploads them to the server. This is your backup, and it is what your lecturer can see.
- **Submitting** is a separate, deliberate step that hands one assignment in for grading. Committing is not submitting.

Because the repository holds the whole course, *Commit Course* on the course row saves everything you have worked on anywhere in the course, in one step. The commit action on a single assignment does the same for that folder alone.

You normally never set the repository up yourself. It is created and downloaded the first time you open the course, and again when the extension starts.

The marks on a row

Rows carry small marks that tell you where your work stands. They appear on assignments, on the units above them, and on the course row itself:

Mark	Meaning
`●`	You have changes that are not committed yet
`↑`	Committed, but not yet uploaded to the server
`⚠`	Uploading or fetching is failing — see *Fix Repository Authentication*

On the course row the `↑` carries a number (`↑3`), counting the commits waiting to be uploaded. A row with no marks is committed and uploaded — safe.

A 🔔 with a number means unread messages.

Actions on the course

Right-click the course row. The icons on the row are shortcuts to the same actions, and entries that cannot apply to your course are hidden rather than shown and then refused.

Commit Course

Saves **all** your work in the course and uploads it, in one step. Use it at the end of a working session, before closing the workspace, or whenever you want your work safely on the server.

Update Repository from Template

Brings your repository up to date with the course material, and **puts back any file you deleted**.

You rarely need to run it yourself: released updates arrive on their own when the workspace starts and when the course view refreshes. This action is the manual trigger for the same sync, for when you do not want to wait.

Two things happen:

1. New and changed material your lecturer has released is merged into your repository.
2. Every file the course material has and your folder does **not** is restored. Files you still have are left exactly as they are — your own work is never overwritten.

That second step is how you undo a mistake. If you have ruined a data file, a figure or a starter file, **delete it** and run this action: a fresh copy comes back.

> **Copy it first** if you want to keep what you wrote. Right-click the file → *Copy File…*, then delete the original.

Show Messages

Course announcements contain updates, deadlines and clarifications from lecturers. You can read them here, while posting announcements is reserved for lecturers and above.

Ask a course question to open your conversation with the teaching team about the course whose Help you opened. For a question about a particular exercise, use *Show Messages* on that assignment instead.

Export Course Examples

Packs the assignments you have on disk into a single ZIP, so you keep your work after the course ends and your access to the repository stops. You choose the layout:

- **Tree** — mirrors the course structure, folders named after the content titles
- **Flat** — one folder per assignment, named after its example identifier

In the browser the archive is saved in your workspace under `exports/` and then downloaded to your computer. If your browser blocks the download, use the *Download again* button on the page that opens, or right-click the file in the Explorer and choose *Download…*. In desktop VS Code you are asked where to save.

Working outside the workspace

The course page on the website (open the course, then *Check access* under *Your repository*) hands you a ready-to-paste `git clone` command with your personal access token in it. That gives you a **full git repository** on your own computer: you can pull updates and push your work with that token from any git client. The token works like a password, so treat it like one. It stays valid on its own — it is not the credential the workspace manages.

This is different from *Export Course Examples* above: the export is a plain ZIP of the files with no git in it, for keeping your work after the course. The clone stays connected to the course. Either way, the workspace remains the supported place to work — testing and submitting happen there.

Fix Repository Authentication

Renews the credentials your repository uses. Use it when uploading suddenly fails and the course row shows ⚠. It does not touch any of your files.

Set up Repository

Creates and downloads your repository. **You normally never need this** — it runs by itself. The entry only appears while the course has no local copy yet, for example after you moved to a different workspace folder or the first attempt failed.

Download Template

For courses handed out as a plain download instead of a repository: downloads the material as a ZIP, or extracts it into a folder you choose. The material is also fetched automatically when you first open such a course; use this if you want a second copy elsewhere.

When something looks wrong

- **A row shows ⚠** — run *Fix Repository Authentication* on the course.
- **New material has not arrived** — run *Update Repository from Template*.
- **You deleted something you needed** — see the same action; deleting the file is what brings a fresh copy back.
- **The tree looks out of date** — collapse and expand the course, which reloads it from the server.

Next

- Units — how a course is organized, and readings
- Assignments — working, testing, submitting, files

Units and Readings

Source: https://github.com/computor-org/computor-vscode/blob/main/docs/help/student-unit.md

A unit groups related content inside a course — usually a week, a topic or a chapter. Units hold assignments, readings and sometimes further units.

```
📖 Programming in Physics
├── ⚫ Week 1 — Introduction
│   ├── 📄 Lecture notes
│   ├── 🟦 Lab 1
│   └── 🟦 Homework 1
└── ⚫ Week 2 — Data Types
```

A unit is a circle, an assignment a square, and other material a document. The colour is the **content type** your lecturer defined — lab, homework, exam, project — so that the kind of work is visible at a glance. Which colour means what is set per course; hovering tells you the type by name.

Units are not work of your own: there is nothing to commit or submit on a unit row. What you can do is read what belongs to it.

Select a unit title, with the mouse or arrow keys followed by Enter, to read its own description beside your code. Select an exercise to read that exercise's README. Use the chevron to expand or collapse children. Descriptions reuse one preview tab, and selecting a unit does not set up a repository.

What a unit tells you

Hover over a unit for its title, how many items it contains, its type, and the number of unread messages if there are any.

On the row itself you may see:

- `🔔 3` — unread messages in this unit
- `● ↑ ⚠` — the state of the assignments underneath it, so a unit with uncommitted work is visible without expanding it. The marks mean the same everywhere; see Courses.

Actions on a unit

Show Description

Opens the text your lecturer wrote for this unit, rendered as a page. The entry appears only when there is a description.

If your lecturer wrote it in more than one language, the one you get follows your reading language (your profile language, otherwise the VS Code display language), with English as the fallback. On an assignment, **Preview Description in Another Language** lets you read the other versions.

Show Messages

Announcements and discussions for this unit — clarifications, hints, deadline changes. Narrower than the course-wide messages, wider than one assignment's.

Readings and other material

Not everything in a course is an assignment. Rows that are neither a unit nor an assignment are material to read or use: lecture notes, references, data. They have no repository, no tests and nothing to submit — so the actions on them are limited to opening the description and reading messages.

If a row looks like something you should be handing in but offers no *Submit*, it is one of these. Ask in the course messages if you expect otherwise; your lecturer may still be preparing it.

Hidden content

Your lecturer can hide a unit while it is being prepared, or after an exam. A hidden unit simply does not appear for you — you never see an empty or greyed row where one used to be. Content appearing later in the term is normal.

Next

- Assignments — working, testing, submitting, files
- Courses — the repository, commits and the row marks

Assignments

Source: https://github.com/computor-org/computor-vscode/blob/main/docs/help/student-assignment.md

An assignment is a piece of work you hand in: a folder in your course repository with the task description, the files you start from, and the files you are meant to write. This page covers working on one, handing it in, and everything you can do with the files inside it.

What the row tells you

```
🟦 Lab 1: Hello World   ● 🔔 1 (2/5) (1/3) 85% 90%
```

Reading that from the left:

Part	Meaning
🟦	The content type your lecturer defined — lab, homework, exam. A tick or cross on the icon shows whether the last test passed
● ↑ ⚠	Your work's state: uncommitted, not uploaded, upload failing. See Courses
🔔 1	Unread messages on this assignment
(2/5)	Test runs: two used of five allowed. Just (2) when there is no limit

Part	Meaning
`(1/3)`	Submissions: one made of three allowed
`85%`	The **test result** — what the automated tests scored
`90%`	The **grade** — your lecturer's official mark, once graded

The two percentages always keep their places: the test result first, the grade second. If you have a grade but no test result, the first slot reads `-%`, so the second number is always the grade and never has to be guessed at.

Hovering gives the same information in words, plus your team members if the assignment is a team one.

Working on an assignment

1. Read the task. Click the exercise title, or use the arrow keys followed by Enter to select it. Its README opens beside your code without setting up a repository. The preview icon and right-click → *Show README Preview* also remain available. Use the chevron to expand the exercise's files when ready to work.

2. Write your solution. Expand the assignment to see its files and click one to open it. Keep the files the task asks for under the names it asks for — renaming a file the tests look for makes them fail.

3. Test. Click the beaker icon, or right-click → *Test Assignment*.

> Testing is not a local dry run. Computor saves your open files, commits them, uploads them, and runs the tests **on the server** — so a test run always uploads your current work, and it always costs one of your test runs if the assignment has a limit. Results come back into the Test Results panel; click a failed test to jump to the code it is about.

4. Commit as you go. The commit icon on the assignment saves and uploads that folder; *Commit Course* on the course row does the whole course at once. Commit early and often — it is your backup, and it costs nothing. It is not a submission and nobody grades it.

5. Submit when you are ready. The upload icon, or right-click → *Submit*. This is the deliberate act of handing in, it is limited, and the count only goes up.

Computor submits the commit that was **tested**, not necessarily your newest one. If those differ — because the last test could not run, for instance — you are told before anything is handed in, so your newest work never gets graded silently or left out silently.

6. Read the feedback. *Show Test Results* for the automated run, *View Details* for the submission history and grading, *Show Messages* for what your lecturer or tutor wrote. A status of *Correction Necessary* means changes are expected; *Improvement Possible* means you may revise if submissions remain.

Test runs and submissions are limited

When your lecturer sets a limit, the row shows how much you have left, and Computor tells you before an action would exceed it rather than after.

Spend test runs deliberately: read the error, reason about the code, then test. Save submissions for work you believe in — once the count is reached you cannot hand in again, and no amount of remaining time helps.

Working with files

Expand an assignment to see its files. Right-click a file, a folder, or the assignment itself. Everything here happens **on your computer only** — nothing reaches the server until you commit.

Creating. *New File…* and *New Folder…* on the assignment or on a folder. The new file opens straight away.

Copying and moving. *Copy File…* and *Move File…* offer only folders inside the same assignment, so work cannot land in a different exercise by mistake. If the name is taken you are asked to *Keep Both* or *Overwrite*. *Cut*, *Copy* and *Paste* are the two-step alternative; *Paste* appears once something is waiting.

Renaming and duplicating. *Rename…* gives a new name; *Duplicate* makes a copy beside the original.

Deleting. *Delete File* and *Delete Folder*, always after a confirmation. If the file was part of an assignment you already submitted, the confirmation says so — you may still delete it, but not without knowing. The same warning appears when you move or rename such a file.

Deleting a file elsewhere — in the Explorer, or with `rm` in a terminal — cannot be caught beforehand. Computor notices it immediately afterwards and, if the file was part of a submission, offers to bring your last committed version back with one click.

Paths. *Copy Path* for the full path, *Copy Relative Path* for its path inside the repository, *Reveal in File Explorer* to show it in the file manager — or in the VS Code Explorer when you work in the browser. If the browser refuses Computor the clipboard, the path is shown in a box to copy by hand.

What you cannot change. A few things are managed for you: `.git`, and at the top of an assignment the `README.md`, its translations, and `mediaFiles` — that is the task description, which your lecturer maintains. READMEs in sub-folders are yours to write.

Getting a file back

If you have ruined a file that came with the assignment — data, a figure, the starter code — you can have a fresh copy:

1. **Copy anything you want to keep** (right-click → *Copy File…*).
2. **Delete** the file.
3. Right-click the **course** → *Update Repository from Template*.

Every file the course material has and your folder does not is restored, and files you still have are left untouched. That is why deleting first is a required step rather than a shortcut: it is what makes the restore safe for everything else you have written.

Team assignments

Some assignments are handed in by a group. You share one folder and one submission count with your teammates: any of you can commit, and any of you can submit for all of you. The hover text lists who is on the team.

Agree on who submits and when. A submission spent by one member is spent for everyone.

When something goes wrong

"I deleted a file I needed." See Getting a file back. If you had committed the file, Computor may also have offered to restore your own version right after the deletion.

"Tests will not run." The assignment has to be set up locally and your work has to be uploadable — a `⚠` on the row means uploading is failing, so run *Fix Repository Authentication* on the course first. If the run starts but every test fails, read the output in Test Results before spending another attempt.

"Submitting failed." Check the submission count on the row. If the count is not the problem, look for `⚠` — a submission needs your work on the server.

"No test runs left." Ask your lecturer in the course messages. Meanwhile your last successful result still stands, and you can still submit.

"My newest work was not graded." Submitting hands in the *tested* commit. Run a test on your latest commit, then submit again if you have one left.

Next

- Courses — the repository, commits and the row marks
- Units — how a course is organized, and readings

Learn with Computor

Source: https://github.com/computor-org/data-science-python/blob/main/docs/GETTING_STARTED.md

Read the three course manifests in the course map, then open an example's `content/index_de.md` or `content/index_en.md`. Reading the public material needs no account. Instructor solutions and private grading tests are kept separately.

Desktop VS Code

Install Python 3.12 and VS Code. Clone this repository, open its folder, and install the **Computor** extension from publisher `computor-org`. In a Python virtual environment, run `python -m pip install --require-hashes -r requirements.lock`. Copy any supplied student template and input files into a working folder and run your code locally. Some exercises begin from an empty file. Check the environment with `python -m pytest tests`.

GitHub Codespaces

Create a Codespace. A GitHub account is required; GitHub's quota and billing apply. The devcontainer installs the pinned Python dependencies and the Computor extension. Code runs in your Codespace, not on Computor's servers. No AI model is installed.

Courses and Lena

Sign into Computor to join a course, keep progress, submit your work, and ask Lena for hints or feedback in the course messages. A hosted workspace is optional: you can continue in desktop VS Code or Codespaces when hosted capacity is full. Lena processes questions using TU Graz's private model service and cannot run code. Your account's course history contains the messages and submitted work according to the platform privacy notice.

To ask Lena, type `@lena` in an assignment's course message, or type `@` and choose Lena from the suggestions. The older `@luna` and the short `@ai` reach the same tutor; choosing a suggestion is optional.

Part B: For tutors and lecturers

The Tutor View

Source: https://github.com/computor-org/computor-vscode/blob/main/docs/help/tutor-course.md

As a tutor you review what students hand in: you look at their work, test it, grade it, and tell them what to change. Computor gives you two views for that in the sidebar:

- **Courses & Members** — where you pick the course and the student to review.
- **Courses** — that student's course content, with everything you need to inspect, test and grade an assignment.

This page explains finding your way around and finding the work that is waiting for you. Grading an assignment covers the work on a single assignment.

Picking a student

Courses & Members lists your courses under their organization and course family. Click a course to select it, then:

- **Course Groups** narrows the member list to one group. Pick *All groups* to see everyone, *No Group* for members without one.
- Click a member to load their content into the **Courses** view.

Two badges behind a member's name tell you where work is waiting:

Badge	Meaning
📝 3	Three submitted assignments you have not reviewed yet
🔔 2	Two unread messages from this student

Work through the members with a 📝 badge and it disappears as you grade — that badge *is* your to-do list.

Reading the content tree

The **Courses** view starts with three rows repeating your selection — course, group, member — followed by the units and assignments of the course, in the order students see them.

Units show how many items they contain, and carry the same `[📝]` and `[🔔]` markers as their assignments, so a collapsed unit never hides waiting work.

Assignments are the rows you act on. The icon's corner shows the grading status you set (green for corrected, red for correction necessary, amber for improvement possible); the badge on it shows the student's last test result and whether they submitted. The numbers behind the name are the student's budgets — `(2/5)` means two test runs used of five allowed, and the second pair counts submissions — followed by the test result and the grade as percentages. Hover for the same information in words, including who graded it.

Clicking an assignment downloads the student's latest submission and the reference solution, and opens the test results if there are any. See Grading an assignment.

Hidden content

Rows marked 👁 `invisible` are content the lecturer has hidden from students — usually staged material or a closed exam. You still see these rows, and you can still grade them: an exam is often hidden right after it ends, and its submissions still need marks.

Messages and comments

Show Messages on an assignment opens your conversation with the student about that assignment — this is where you tell them what to fix. When you set a grading status of *Correction Necessary* or *Improvement Possible*, this conversation opens by itself, because the student cannot act on the verdict until you say what to change.

Show Comments on a member is different: comments are notes about the student that only course staff can read. The student never sees them. Use them for anything the next tutor or the lecturer should know.

Next

- Grading an assignment — checkout, testing, comparing and grading a single piece of work

Grading an Assignment

Source: https://github.com/computor-org/computor-vscode/blob/main/docs/help/tutor-assignment.md

One assignment, one student: this page walks through reviewing a submission from opening it to the grade and the message that goes with it. How to find the student and the assignment is covered in The Tutor View.

What is under an assignment

Click the assignment row (or right-click → *Checkout*) and Computor downloads what you need. Expanding the row then shows:

- **Submissions** — the files of the student's latest submission. This is what you grade.
- **References** — the reference solution for the deployed version of the assignment, so you can see what was expected.
- **History** — earlier submissions, when there is more than one. Each entry can be expanded, tested and downloaded like the latest one.

Checkout from the context menu asks before re-downloading files you already have; clicking the row just fetches what is missing.

Reviewing the work

Read the task first. The preview icon on the row (or *Show README Preview*) opens the assignment description the student worked from.

Open the files. Expand *Submissions* and click a file to read it. To see how it differs from the sample solution, right-click a submission file → *Compare with Reference* — a diff opens against the same file in the reference.

Look at their test result. Clicking the row opens the student's own last test run in the Test Results panel. Entries under *History* offer *Show Test Results* for that particular submission.

Run your own test. *Run Test* on the assignment or a submission packages the downloaded files and runs the course's tests on them. This is your test run, not the student's — it does not use up their budget. On *References* the same command checks that the reference itself still passes.

Setting the grade

Click the grading icon on the assignment row, or right-click → *Grading…*. You are asked for two things:

1. **The grade** — a number between 0.00 and 1.00. This is the official mark, shown to the student as a percentage next to their test result.
2. **The status** — what should happen next:

Status	Meaning
corrected	Done. The work is reviewed and the grade stands
correction_necessary	The student must revise and submit again
improvement_possible	Good enough to stand, but the student may improve it
not_reviewed	Take the review back, e.g. after grading the wrong student

The grade applies to the latest submitted version. If the student submits again afterwards, the assignment shows up as unreviewed again.

After a correction verdict, say what to change. When you pick *correction_necessary* or *improvement_possible*, the message conversation for this assignment opens by itself. The student only sees the status and the percentage — without a message they cannot know what was wrong, so write one before moving on.

An assignment marked `👁 invisible` is graded exactly the same way — hidden only means students cannot see the content any more, not that its submissions stop needing marks.

Granting extra attempts

When a student is out of test runs or submissions and asks for another attempt, right-click the assignment → *Set Max Test Runs for Student…* or *Set Max Submissions for Student…*. The number you enter applies to this student on this assignment only; leaving the field empty puts them back on the assignment's normal limit.

Next

- The Tutor View — finding students, badges, hidden content, messages and comments

Teacher guide: course dashboard

Source: https://github.com/computor-org/computor-backend/blob/main/docs/teacher-guide.md

This guide is for lecturers and tutors who use the Computor web app to follow their students' work. It explains what each page shows, where the numbers come from, and what they do not tell you. No technical knowledge is needed.

Everything on these pages is computed from what the platform already records when students work on assignments:

- **Test runs**: each time a student runs the automatic tests, the time, the score (0–100 %) and the test report are stored.
- **Submissions**: when a student officially submits an assignment, the time is stored.
- **Grades and feedback**: when a tutor or lecturer grades a submission, the grade, status, written feedback, grader and time are stored.

The platform does not record keystrokes, screen time or time spent in the editor. Anything that looks like "time spent" is an estimate and is labelled as one.

Where to find the pages

Open a course and use the sidebar:

Sidebar entry	Who sees it	What it is for
Lecturer → Grading	Lecturers and above	Course statistics, the student list with flags, and each student's detail page
Lecturer → Typical errors	Lecturers and above	The test failures that many students share
Tutor	Tutors and above	How much work is waiting for review
Tutor → Submissions	Tutors and above	The review queue
Tutor → Grading	Tutors and above	Grading one student at a time

Tutors see only the Tutor pages. The Lecturer pages show data about the whole course and are reserved for lecturers, course maintainers, course owners and administrators.

Course statistics (Lecturer → Grading)

The top of the page summarises the course:

- **Students**, **Average progress** and **Completed**: progress is the share of submittable assignments a student has officially submitted.
- **At risk**: students below 25 % progress.
- **Inactive**: students without a submission in the last 14 days.
- **Progress distribution** and the content-type chart show how the course is spread across progress levels and assignment types.

Flags

Below the charts, the **Flags** panel lists simple, fixed rules that point out students you may want to look at. A flag is a prompt to look, not a verdict: each one says which rule fired and the numbers that triggered it. Open **How flags are computed** on the page to see every rule with its threshold.

Flag	When it appears
Many failed attempts	10 or more test runs on one assignment without reaching the full score. Shown as a warning while the full score is still missing, as information once it was reached.
Sudden jump to full score	A test run reached 100 % directly after a run at or below 30 % on the same assignment. Often a student simply found the bug; it is not evidence of copying on its own.
Inactive	No test run and no submission in the last 14 days, or none at all.
Low progress	Fewer than 25 % of the submittable assignments submitted.
Last-minute submission	The last submission came less than 2 hours before the due date.
Submission after the due date	At least one submission after the due date.

The platform does not store deadlines per assignment. The two deadline rules use the **Due date** you enter in the Flags panel (or on a student's page). It is remembered in your browser for this course and applies to all assignments, so set it to the deadline you are interested in. Without a due date the two deadline rules are skipped.

The student table has a **Flags** column. Hover over a flag to read its reason, and tick **Only flagged** to list just the flagged students. Click a student to open their detail page.

A single student (Lecturer → Grading → a student)

The detail page shows, from top to bottom:

1. **Average grade, progress and submission counts.**
2. **Submission timeline**: the student's submissions accumulated over time, with the start and due date you set at the top of the page.
3. **Flags**: the same rules as above, for this student, each with its reason.
4. **Attempt timeline** (in German: *Bearbeitungsverlauf*): one row per assignment the student has worked on.
    - *Attempts over time*: dots are test runs, coloured by score (green: full score, amber: at least 50 %, red: below 50 %); bars are submissions; a dashed line marks the due date. All rows share one time axis, so you can see which assignment was worked on when.
    - *Runs* and *Submissions*: how many of each.
    - *First / last activity*: the first and last test run or submission.
    - *Full score reached*: when a test run first reached 100 %, or "not yet".
    - *Best / latest*: the best and the most recent test score.
    - *Days*: on how many different days the student worked on it.
    - *Median gap*: the typical time between two consecutive test runs.

    - *Est. active time*: an **estimate**. It adds up the gaps between consecutive test runs that are at most 30 minutes apart. Work done without running the tests is invisible to the platform, so read this as a lower bound, never as measured working time.
    - *Last submission* (only with a due date): how long before or after the due date the last submission came.
5. **History of runs, submissions and feedback**: everything in time order, newest first. A *Submission run* is a test run on an official submission; a *Grading* row shows the grade, its status, the written feedback and who gave it. Use the two filters to show one assignment or one kind of entry.
6. **Course content progress**: every assignment with its status and number of attempts.

Use **Print** to get a paper or PDF copy of the page, for example for a consultation with the student.

Typical errors (Lecturer → Typical errors)

This page collects the failing tests from the students' test runs and groups those that fail for the same reason. It helps you spot problems that many students share, for example a misunderstood task or a missing hint.

For each group you see the assignment, the failing test, the error message, how many **students** are affected and how many failing **runs** there were. Groups affecting the most students come first. Click a row to see up to three examples from different students with the original message and a link to that student's page.

How messages are grouped: only the first line that names the error is kept, and details that differ from student to student are replaced by placeholders: numbers by `<n>`, quoted values by `<str>`, file paths by `<path>` and memory addresses by `<addr>`. Plain names in quotes, such as `'np'` in "name 'np' is not defined", are kept because they usually name the actual problem. So "assert 3 == 4" and "assert 7 == 2" become one group, `assert <n> == <n>`.

Things to know:

- Only test runs by students count; lecturers' and tutors' own test runs are left out.
- The page reads the newest failing test runs, 300 by default. If more exist, a notice says so; choose a larger number or a single assignment to look further back.
- *Runs without report* counts runs for which no detailed test report was stored. They cannot contribute to any group.

Tutor pages

Tutor (overview)

Shows how many students you have, how many have ungraded submissions, the number of ungraded submissions and unread messages, with links to the two pages below.

Submissions

The review queue: every student or team with submissions, by default only those with ungraded work, oldest waiting first. Filter by assignment, or untick **Only ungraded** to see everything. Click a row to open that student's grading page at the right assignment.

Grading

Choose a student from the list (tick **Only with ungraded submissions** to narrow it down). The student page lists their assignments on the left; pick one to see:

- the latest test result and the number of submissions and test runs,
- a form to **grade the latest submission**: a grade in percent, a status (*Corrected*, *Correction necessary*, *Improvement possible* or *Not reviewed*) and written feedback for the student,
- the earlier grades and feedback for this assignment,
- the details of the latest test run.

The grade field is pre-filled with the latest automatic test score; change it as needed. When a grade refers to an older submission than the latest one, the page says so.

What the dashboard does not do

- It does not judge students automatically. Flags are fixed, visible rules with stated thresholds; there is no hidden score.
- It does not measure time spent working. The only time figure is the labelled estimate in the attempt timeline.
- It does not know assignment deadlines unless you enter a due date.

Part C: Creating courses and assignments

Computor Testing Framework - User Guide

Source: https://github.com/computor-org/computor-backend/blob/main/computor-testing/docs/USER_GUIDE.md

This guide is for **exercise authors** who write test.yaml files and use the framework to test student submissions.

Quick Start

1. Install

From the root of a `computor-backend` checkout (`computor-types` is not on PyPI):

```
pip install -e ./computor-types -e ./computor-testing
```

2. Check your environment

```
computor-test check --all
```

3. Create a test.yaml

```
# Use the blocks CLI to generate a starter file
blocks init -l python -n "My Exercise Tests"

# Or create manually (see examples below)
```

4. Run tests

```
computor-test python run -t ./student -T test.yaml
```

Directory Structure

Your exercise directory should look like:

```
my-exercise/
├── test.yaml                 # Test definitions (required)
├── meta.yaml                 # Metadata (required for upload to the platform)
├── content/
│   └── index_en.md           # Assignment text, one file per language
├── reference/                # Reference solution
│   └── solution.py
├── localTests/
│   ├── correctSolution/      # Should pass all tests
│   │   └── solution.py
│   └── wrongSolution/        # Should fail some tests
│       └── solution.py
└── output/
    └── testSummary.json      # Generated test results
```

Writing test.yaml

Basic Structure

```
name: Exercise Name
description: What this exercise tests
type: python           # python, octave, r, julia, c, fortran, document
version: "1.0"

properties:
  relativeTolerance: 1e-12    # Default for numeric comparisons
  absoluteTolerance: 0.0001
  timeout: 180                # Seconds

  tests:
    - type: exist             # Test collection
      name: File Check
      tests:
        - name: solution.py   # Individual test

    - type: variable
      name: Variable Tests
      entryPoint: solution.py
      tests:
        - name: result
          value: 42
```

Property Cascade

Properties set at the `properties:` level serve as defaults. They can be overridden at the test collection level, which can be overridden at the individual test level.

The cascade applies in all testers to `qualification`, `relativeTolerance`, `absoluteTolerance`, `typeCheck`, `shapeCheck`, `ignoreClass` and `verbosity`, and from `properties:` to the collections also to `timeout`, `storeGraphicsArtifacts` and `competency`. `allowedOccuranceRange` and `occuranceType` are not inherited; set them on each test that needs them.

Test Types

exist - File Existence

Available for ALL languages. Checks that required files exist.

```
- type: exist
  name: Required Files
  tests:
    - name: solution.py          # Exact filename
    - name: data/*.csv           # Glob pattern
    - name: config.json
      allowEmpty: true           # Allow 0-byte files (default: false)
```

variable - Variable Comparison

Available for: Python, Octave, R, Julia

Runs a script and compares workspace variables against expected values.

```
- type: variable
  name: Computation Tests
  entryPoint: solution.py          # Script to execute
  tests:
    - name: result                 # Variable name
      value: 42                    # Expected value

    - name: matrix
      value: [[1, 2], [3, 4]]     # Arrays/matrices

    - name: pi_approx
      value: 3.14159
      relativeTolerance: 1e-4     # Override default tolerance

    - name: greeting
      value: "Hello, World!"
```

With Setup/Teardown Code

```
- type: variable
  name: Advanced Tests
  entryPoint: solution.py
  setUpCode:
    - "x = 10"                     # Python and Octave: runs after the entry point
    - "y = compute(x)"
  tearDownCode:
    - "cleanup()"                  # Runs after setUpCode, before variables are read
  tests:
    - name: y
      value: 100
```

With Input (stdin)

```
- type: variable
  name: Interactive Tests
  entryPoint: solution.py
  inputAnswers:
    - "5"                          # First input() call gets "5"
    - "hello"                      # Second input() call gets "hello"
  tests:
    - name: result
      value: 5
```

stdout - Output Comparison

Available for: C, Fortran, Python, R, Julia

```
- type: stdout
  name: Output Tests
  entryPoint: main.c
  tests:
    - name: greeting
      qualification: contains      # Check output contains this
      pattern: "Hello"

    - name: exact
      qualification: matches       # Exact match
      value: "Hello, World!\n"

    - name: pattern
      qualification: regexp        # Regex match
      pattern: "Result: \\d+"

    - name: prefix
      qualification: startsWith
      pattern: "Program"

    - name: suffix
      qualification: endsWith
      pattern: "Done."
```

Without a `qualification`, the output is compared with `value` or, if no `value` is given, with the output of the reference solution. C and Fortran trim the output before comparing and also accept `value` in place of `pattern`. Python compares the untrimmed output and takes the expected text for every qualification, including `matches`, from `pattern`. R and Julia support `contains` and `regexp` (and treat `matches` as a regular expression).

structural - Code Structure

Available for ALL languages. Checks for keywords/patterns in source code.

```
- type: structural
  name: Code Structure
  file: solution.py               # File to analyze
  tests:
    - name: no_eval               # Descriptive name
      pattern: "\\beval\\b"       # Regex pattern to search
      allowedOccuranceRange: [0, 0]  # Must appear 0 times (forbidden)

    - name: uses_function
      pattern: "\\bdef\\b"
      allowedOccuranceRange: [1, 0]  # At least once, no upper limit

    - name: for_loop
      pattern: "\\bfor\\b"
      allowedOccuranceRange: [2, 5]  # Must appear 2-5 times
```

Note on `allowedOccuranceRange`: The format is `[min, max]`:

Value	Meaning
`[0, 0]`	The item must **not** occur at all
`[n, 0]` (n > 0)	At least `n` occurrences, no upper limit
`[min, max]`	Between `min` and `max` occurrences, inclusive

Alternatively, `countRequirement: n` asserts an exact count. It applies only when `allowedOccuranceRange` is not set; a sub-test with neither is skipped.

For Python, `name` is matched against the token stream, so dotted names such as `np.pi` are counted as a unit and matches inside comments and strings are ignored. Supplying a `pattern` instead switches to a regex over the raw file text, which does include comments and strings.

R and Julia count `pattern` (or `name`) in the same way. Octave ignores `pattern` and counts the keyword or function given as `name`. C and Fortran apply the range to known keywords given as `name` (such as `for` or `malloc`); for any other test they only check that `pattern` (or `name`) occurs in the source.

exitcode - Exit Code Check (C, Fortran)

```
- type: exitcode
  name: Exit Code
  entryPoint: main.c
  tests:
    - name: success
      expectedExitCode: 0

    - name: error_case
      expectedExitCode: 1
```

compile - Compilation Test (C, Fortran)

```
- type: compile
  name: Compilation
  entryPoint: main.c
  compiler: gcc
  compilerFlags: ["-Wall", "-Wextra", "-Werror"]
  tests:
    - name: compiles_clean         # Should compile without errors
```

graphics - Plot Testing (Python, Octave)

Each `name` is an expression evaluated after the script has run. In Python it is prefixed with `plt.`; in Octave it is evaluated as written. The result is compared with `value` or with the reference solution.

```
- type: graphics
  name: Plot Tests
  entryPoint: plot_script.py
  tests:
    - name: gcf().axes[0].get_xlabel()
      value: "Time"
    - name: gcf().axes[0].get_title()
      value: "Data Plot"
```

```
- type: graphics
  name: Plot Tests
  entryPoint: plot_script.m
  tests:
    - name: get(gca, 'XLim')
      value: [0, 10]
```

Document Test Types

Available for: Document

```
type: document

properties:
  tests:
    - type: wordcount
      name: Word Count
      file: essay.md
      tests:
        - name: minimum_words
          allowedOccuranceRange: [500, 0]   # At least 500 words

    - type: section
      name: Required Sections
      file: essay.md
      tests:
        - name: Introduction
        - name: Methodology
        - name: Conclusion

    - type: keyword
```

```
      name: Keywords
      file: essay.md
      tests:
        - name: algorithm
          allowedOccuranceRange: [3, 0]     # At least 3 occurrences

    - type: linecount
      name: Line Count
      file: essay.md
      tests:
        - name: minimum_lines
          allowedOccuranceRange: [50, 0]
```

Success Dependencies

You can make test collections depend on previous ones:

```
properties:
  tests:
    - type: exist
      name: File Check
      id: file_check              # Give it an ID
      tests:
        - name: solution.py

    - type: variable
      name: Variable Tests
      entryPoint: solution.py
      successDependency: file_check   # Only runs if File Check passed
      tests:
        - name: result
          value: 42
```

Token Exchange

In Octave tests, files found by earlier `exist` tests can be referenced as `#file_N#` (file name, counted from 1) in the `file` of a structural test:

```
- type: exist
  name: Script
  tests:
    - name: "*.m"                 # Whatever .m file the student submitted

- type: structural
  name: Loops
  file: "#file_1#"                # Replaced with the file found above
  tests:
    - name: for
      allowedOccuranceRange: [1, 0]
```

---

meta.yaml

Metadata of the example, required when it is uploaded to the platform. The tester itself reads only `properties.testDependencies`.

```
slug: physics.python.hello_world  # Unique, dot-separated identifier
version: '1.0'
title: Hello World in Python
description: A simple Hello World program
language: en
license: CC BY-NC 4.0
keywords: [python, basics]         # Shown as tags in the example library
category: Basics

authors:
  - name: Your Name
    email: your.email@example.org
    affiliation: Your Institute

properties:
  studentSubmissionFiles:
    - solution.py
  executionBackend:
    language: python
    version: "3.12"
```

`executionBackend` takes either a `language` (python, octave, r, julia, c, cpp, fortran, document, matlab), matched against the test runners of the installation, with an optional `version`, or a `slug` that pins one specific runner. Its `settings` are stored but not applied by the testers; set `timeout` and compiler options in `test.yaml`.

CLI Reference

Unified CLI

```
# Run tests
computor-test <language> run -t <student_dir> -T <test.yaml> [-v 0-3]

# Local testing (test.yaml in the parent directory or in <solution_dir>)
computor-test <language> local -d <solution_dir>

# Check runtime
computor-test check --all
computor-test check python

# List available testers
computor-test list
```

Blocks CLI (test.yaml generation)

```
# List available test types
blocks list -l python

# Initialize a new test.yaml
blocks init -l python -n "My Tests"

# Generate a test snippet
blocks generate -l python -t variable -n result

# Browse templates
blocks templates list
blocks templates show "Variable Check" -l python

# Export for VSCode extension
blocks export ./generated
```

Sandbox CLI

```
# Check backends
sandbox check -v

# Run a command in sandbox
sandbox run python3 -c "print('hello')"

# Show configuration
sandbox config --show

# Test sandbox functionality
sandbox test
```

Dependency Installer

```
# Install all dependencies
deps-installer install -f dependencies.yaml

# Generate Dockerfile
deps-installer dockerfile -f dependencies.yaml -o Dockerfile

# Validate dependencies file
deps-installer validate -f dependencies.yaml
```

Troubleshooting

"Runtime not found"

```
# Check which runtimes are available
computor-test check --all

# For R, make sure jsonlite is installed:
Rscript -e "install.packages('jsonlite', repos='https://cloud.r-project.org/')"

# For Julia, make sure JSON is installed:
julia -e 'import Pkg; Pkg.add("JSON")'
```

Tests pass locally but fail on server

- Check timeout values - server may be slower
- Ensure no hardcoded absolute paths in test.yaml
- Check that the packages your code imports are installed on the platform's test runner; the platform does not read `dependencies.yaml`
- Verify `executionBackend` in meta.yaml names a language or runner the installation provides

"Execution failed" with no further message

Each student process runs with resource limits. `COMPUTOR_JOB_MAX_PROCESSES` (default 512) counts all processes and threads of your user, so on a busy workstation local runs can fail to start the interpreter. Set `COMPUTOR_JOB_MAX_PROCESSES=0` for local runs.

"Path validation failed"

The framework prevents directory traversal attacks. Ensure:

- No `..` in file paths
- All paths are relative to the test root directory
- Entry points reference files within the student/reference directories

Output not captured

For interpreted languages, make sure the entry point script actually produces output. For compiled languages, make sure `printf`/`cout` flushes output (use `\n` or `fflush(stdout)`).