Collaborative Research Software Engineering @ WISJ 2025 [Part 1 of 2]

Instructors: Rousslan Dossa & Nadine Spychala (original author)

Greetings everyone!

This is a tutorial on best practices in research software engineering using Python as an example programming language:

- writing research software well and collaboratively,
- while explicitly taking into account software sustainability, open-source and reproducible science.

It is a **modified version** of the original Collaborative Research Software Engineering Tutorial by Nadine Spychala. It is based on Intermediate Research Software Development course from the Carpentries Incubator, so for anyone wanting to delve into more detail or get more practice... looking into that course is probably one of the best options.

Here, you'll get:

- a selection of the most relevant practices from most sections of the original course, with an emphasis on **version control, testing, some basic software design, and a collaborative development pipeline**,
- as well as some **new learning content, resources and tools** with deep learning, model training and experiment analysis.

We'll use GitHub CodeSpaces – a cloud-powered development environment that one can configure to one's liking.

- Everyone will instantiate a **GitHub codespace** within their GitHub account and **all coding will be done from there** - folks will be able to directly apply what is taught in their codespace, work on exercises, and implement solutions.
- Thus, the only thing you will need for this tutorial is an account on GitHub. More on GitHub CodeSpaces further below.

Table of Contents

0.1 Recap & motivation: why collaboration and best research software practices in the first place?

- In science, we often want or need to reproduce results to **build knowledge incrementally**.

    - If, for some reason, results can't be reproduced, we at least want to **understand the steps taken** to arrive at the results, i.e., have transparency on the tools used, code written, computations done, and anything else that has been relevant for generating a given research result.
- However, very often, the steps taken - and particularly the **code** written, for generating scientific results are **not available**, and/or **not readily implementable**, and/or **not sufficiently understandable**.
- The consequences are:
    - **Redundant**, or, at worst, **wasted work**, if reproduction of results is essential, but not possible. This, in the grand scheme of things, greatly slows down scientific progress.
    - Code that is not designed to be possibly re-used – and thus scrutinized by others – runs the risk of being flawed and therefore, in turn, produce, **flawed results**.
    - It **hampers collaboration** – something that becomes increasingly important as
        - people from all over the world become more inter-connected,
        - more diversified and specialized knowledge is produced (such that different "parts" need to come together to create a coherent "whole"),
        - the amount of people working in science increases,
        - many great things can't be achieved alone.
- To manage those developments well and avoid working in silos, it is important to have **structures** at place that **enable people to join forces**, and respond to and integrate each other's work well - we need more teamwork.
- **Why is it difficult to establish collaborative and best coding practices?** For cultural/scientific practice reasons, and the way academia or industry has set up its **incentives** (in terms of # of papers where authors are given credit as *individuals*, and prestige of journals plays a role), special value is placed on individual rather than collaborative research outputs. It also discourages doing things right rather than quick-and-dirty. This needs to change.

0.2 Difference between "coding" and research software engineering

The terms *programming* (or even *coding*) and *software engineering* are often used interchangeably, but those terms don't mean the same thing. Programmers or coders tend to focus on one part of software development which is implementation. Also, in the context of academic research, they often write software just for themselves and are the sole stakeholders.

Someone who is *engineering* software takes a broader view on code which also considers:

- **the lifecycle of software**: viewing software development as a process that proceeds from understanding software requirements, to writing the and and using/releasing it, to what happens afterwards,
- **who will (or may) be involved**: software is written for stakeholders - this may only be one researcher initially, but there is an understanding that others may become involved later on (even if that isn't evident yet),
- **the value of the software itself**: it is not merely a by-product, but may provide benefits in various ways, e.g., in terms of what it can do, the lessons learned throughout its development, and as an implementation of a research approach (i.e. a particular research algorithm, process, or technical approach),
- **its potential reuse**: there is an assumption that (parts of) the software could be reused in the future (this includes your future self!).

Bearing the difference between coding and software engineering in mind, how much do scientists actually need to do either of them? Should they rather code or write software, or do both (and if both, when do they do what)? This is a hard question and will very much depend on a given research project. In Scientific coding and software engineering: what's the difference?, it is argued that "*scientists want to explore, engineers want to build*". Both too little or too much of an engineering component in writing code can be a hindrance in the research process:

- Too much engineering right at the start can be a problem because unlike other professions, doing research often means to venture into the unknown, and to take/abandon many different paths and turns - overgeneralizing code can be wasted work, if the research underlying it hasn't gone beyond the exploratory stage (and if noone else except for oneself is engaging in this exploratory stage).
- Too little engineering can be a problem, too, as any code accompanying research that is beyond the very first exploratory stages, and where other people are involved in using it in some way will face problems.

To boil down the challenge in other words, when you start out writing code for your research, you need to ask yourself:

How much do you want to generalize and consider factors in the software lifecycle *upfront* in order to spare work at a later time-point **vs. stay specific and write single-use code** to not end up doing (potentially) large amounts of unnecessary work, if you (unexpectedly) abandon paths taken in your research?

While this is a question every coder/software engineer needs to ask themselves, it's a particularly important one for researchers.

It may not be easy to find a sweet spot, but, as a heuristic, you may err on the side of incorporating software engineering into your coding, as soon as

- you believe that there's a slight chance the code will be re-used by others (including yourself) now or in the future,
- you believe that there's a chance you are on a research trajectory that you will potentially, to some extent, communicate about in the future (in which case you'd need to provide the code alongside your research and have it ready for reuse, e.g., to explore your research and/or build incrementally on top of it).

More often than not, one or both points will apply fairly quickly.

0.3 What you'll learn

This tutorial equips you with a solid foundation for working on software development in a team, using practices that help you write code of higher quality, and that make it easier to develop and sustain code in the future – both by yourself and others. The topics covered concern core, intermediate skills covering important aspects of the software development life-cycle that will be of most use to anyone working collaboratively on code.

At the start, we'll address

- Integrated Development Environments,
- Git and GitHub,
- virtual environments and dependencies

Regarding testing software, we will se how

- ensure that results are correct by using unit testing and scaling it up,
- debug code & include code coverage,
- continuous integration for automated testing

Regarding software design, we will touch upon

- general principles to keep in mind for research software projects,
- a bit of object-oriented programming, and
- functional programming.

With respect to working on software with - and for - others, you'll hear about

- collaboratively developing software on GitHub (using pull requests),
- application programming interfaces,
- packaging code for release and distribution.

An opiniated section on collaborative research in machine learning

- sharing experiment results with team and the public, along with code and papers
- some tools for collaborative analysis

Some of you will likely have written much more complex code than the one you'll encounter in this tutorial, yet we call the skills taught "intermediate", because for code development in teams, you need more than just the right tools and languages – you need a strategy (best practices) for how you'll use these tools *as a team*, or at least for potential re-use by people outside your team (that may very well consist only of you). Thus, it's less about the complexity of the code as such within a self-contained environment, and more about the complexity that arises due to other people either working on it, too, or re-using it for their purposes.

0.4 Target audience

The best way to check whether this tutorial is for you is to browse its contents in this HackMD main document.

This tutorial is targeted to anyone who

- has some basic familiarity with Git/GitHub, and
- has basic programming skills in Python (or any other programming language),
- aims to learn more about best practices and new ways to tackle research software development (as a team).

It is suitable for all career levels – from students to (very) senior researchers for whom writing code is part of their job, and who either are eager to up-skill and learn things anew, or would like to have a proper refresh and/or new perspectives on research software development.

If you're keen on learning how to restructure existing code such that it is more robust, reusable and maintainable, automate the process of testing and verifying software correctness, and collaboratively work with others in a way that mimics a typical software development process within a team, then we're looking forward to you!

0.5 What you need to do before the event

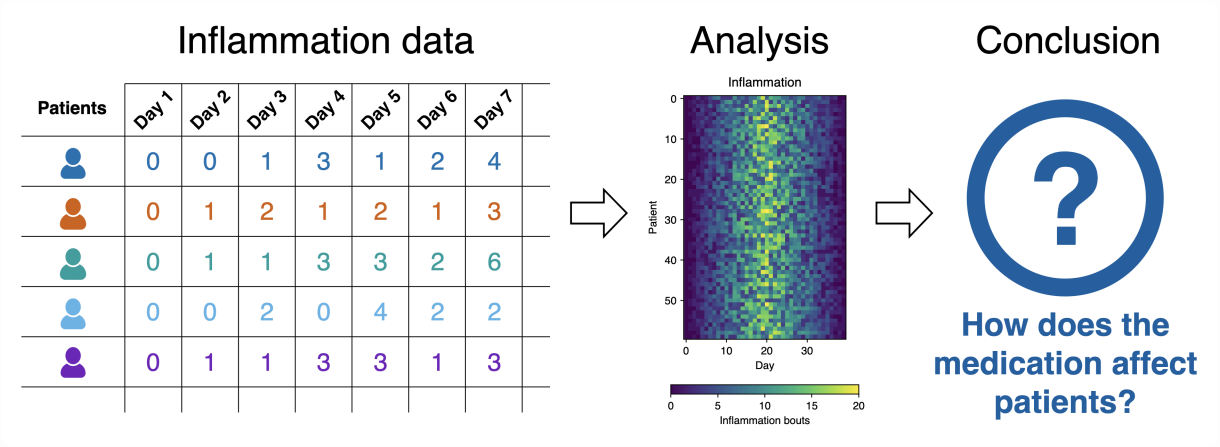
The only thing you need to before the event is to create an account on GitHub, if you haven't done so already.

Let's start! 🚀💥

1. Introduction into the project & setting up the environment

1.1 The project

In this tutorial, we will use the Patient Inflammation Study Project which has been set up for educational purposes by the course creators, and is stored on GitHub. The project's purpose is to study the effect of a treatment for arthritis by analysing the inflammation levels in patients who have been given this treatment.



The data:

- each csv file in the `data` folder of the repository represents inflammation measurements from one separate clinical trial of the drug,
- each single csv file contains information for 60 patients whose inflammation levels had been recorded for 40 days whilst participating in the trial:
  - each row holds inflammation measurements for a single patient,
  - each column represents a successive day in the trial,
  - each cell represents an inflammation value on a given day for a given patient (in some arbitrary units of inflammation measurement).

The project as seen on the repository is not finished and contains some errors. We will work incrementally on the existing code to fix those and add features during the tutorial.

Goal: Write an application for the command line interface (CLI) to easily retrieve patient inflammation data, and display simple statistics such as the daily mean or maximum value (using visualization).

The code:

- a Python script `inflammation-analysis.py` which provides the main entry point in the application - this is the script we'll eventually run in the CLI, and for which we need to provide inputs (such as which data files to use),
- three directories - `inflammation` which contains collections of functions in `views.py` and `models.py`, `data` and `tests` which contains tests for our functions in `inflammation`,

- there is also a `README` file (describing the project, its usage, installation, authors and how to contribute).
- We'll get into each of these files later on.

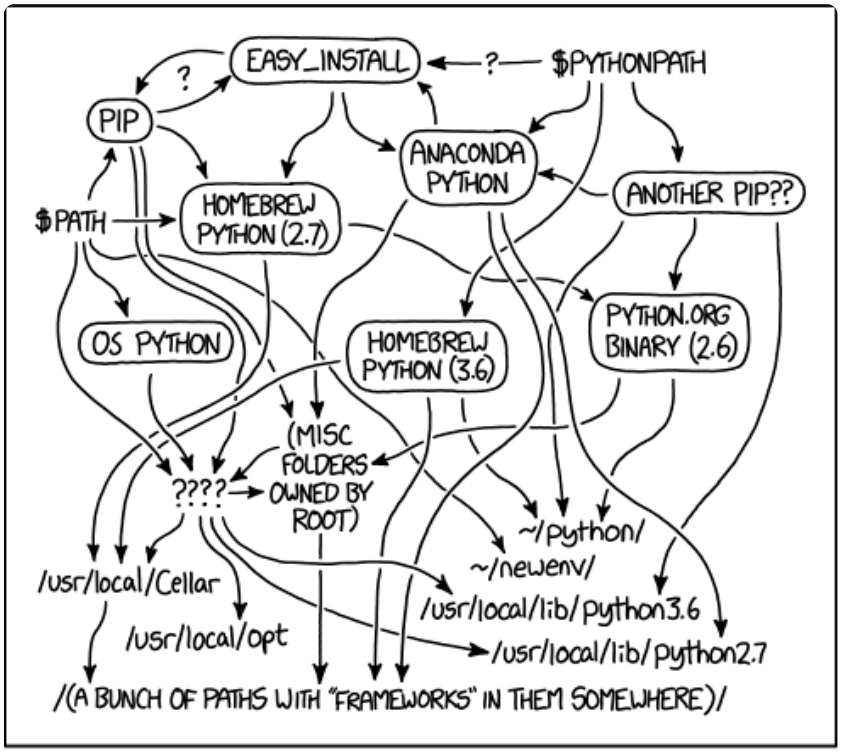
1.2 GitHub CodeSpaces

We will use GitHub CodeSpaces. A codespace is a cloud-powered development environment that you can configure to your liking. It can be accessed from:

- A web browser,
- Visual Studio Code,
- the JetBrains Gateway application (a compact desktop app that allows you to work remotely with a JetBrains IDE such as PyCharm without necessarily downloading one)

GitHub CodeSpaces' most appealing feature is that you can code from any device and get a standardized environment as long as you are connected to the internet.
This is ideal for our purposes (and maybe for some of yours in the future, too) as we'll avoid the hassle to install programs on your machines and copy/clone GitHub repositories remotely/locally before you will be able to code.
This spares us unexpected problems that would very likely occur when setting up the environment we need.

Python in particular can be a mess when it comes to dependencies between different components... see this XKCD webcomic for an insightful illustration:



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

> **! FOLLOW ALONG IN YOUR CODESPACE ! Let's instantiate a GitHub codespace**:

1. Log into your GitHub account, if not already
2. Go to this fork of the Patient Inflammation Study Project, click on the upper right green button `Code`, then choose `Create codespace in main`. Your codespace should load and be ready in a few seconds.
3. A GH Codespace can be intuited as a small virtual machine running Linux and having basic SWE utilities pre-installed
4. Let's inspect what we see...

1.3 Integrated Development Environments

In the cloud, you'll see that we use VSCode as an **Integrated Development Environment** (IDE), however, you could also use a codespace via your locally installed VSCode program by adding a GitHub Codespaces extension to it.

- What is an Integrated Development Environment? IDEs help manage all the arising complexities of the software development process, and normally consist of at least a source code editor, build automation tools and a debugger. They often provide support for syntax highlighting, code completion, code refactoring, and embedded version control via Git (we'll get to this below). As your codebase gets bigger and more complex, it will quickly involve many different files and external libraries.
- IDEs are extremely useful and modern software development would be very hard without them.
- VSCode comes with
    - **syntax highlighting**: makes code easier to read, and errors easier to find) code and finding errors easier,
    - **code completion**: speeds up the process of coding by, e.g., reducing typos, offering available variable names, functions from available packages, or parameters of functions,
    - **code definition & documentation references**: helps you get code information by showing, e.g., definitions of symbols (e.g. functions, parameters, classes, fields, and methods),
    - **code search**: helps you find a text string within a project by, e.g., using different scopes to narrow your search process,
    - **plethora of extensions** to fit a lot of scenarios (programming langauges)
    - **integrated debugger** for a smoother interactive debug experience, and
    - **version control** (more on that below) many other things ...

- We have the standard icons as displayed in VSCode on our local machines:
  - `Explorer`: browse, open, and manage all of the files and folders in your project,
  - `Run and Debug`: see all information related to running and debugging,
  - `Extensions`: add languages, debuggers, and tools to your installation,
  - `Source control` (or version control): to track and manage changes to code.
- We can use the `Terminal`: an interface in which you can type and execute text based commands - here, we'll use Bash which is both a Unix shell and command language,
  - A shell is a program that provides an entry point to/exposes an operating system's services to a human user or other programs.

> **! FOLLOW ALONG IN YOUR CODESPACE ! Let's install the Python and Jupyter extension for VSCode**

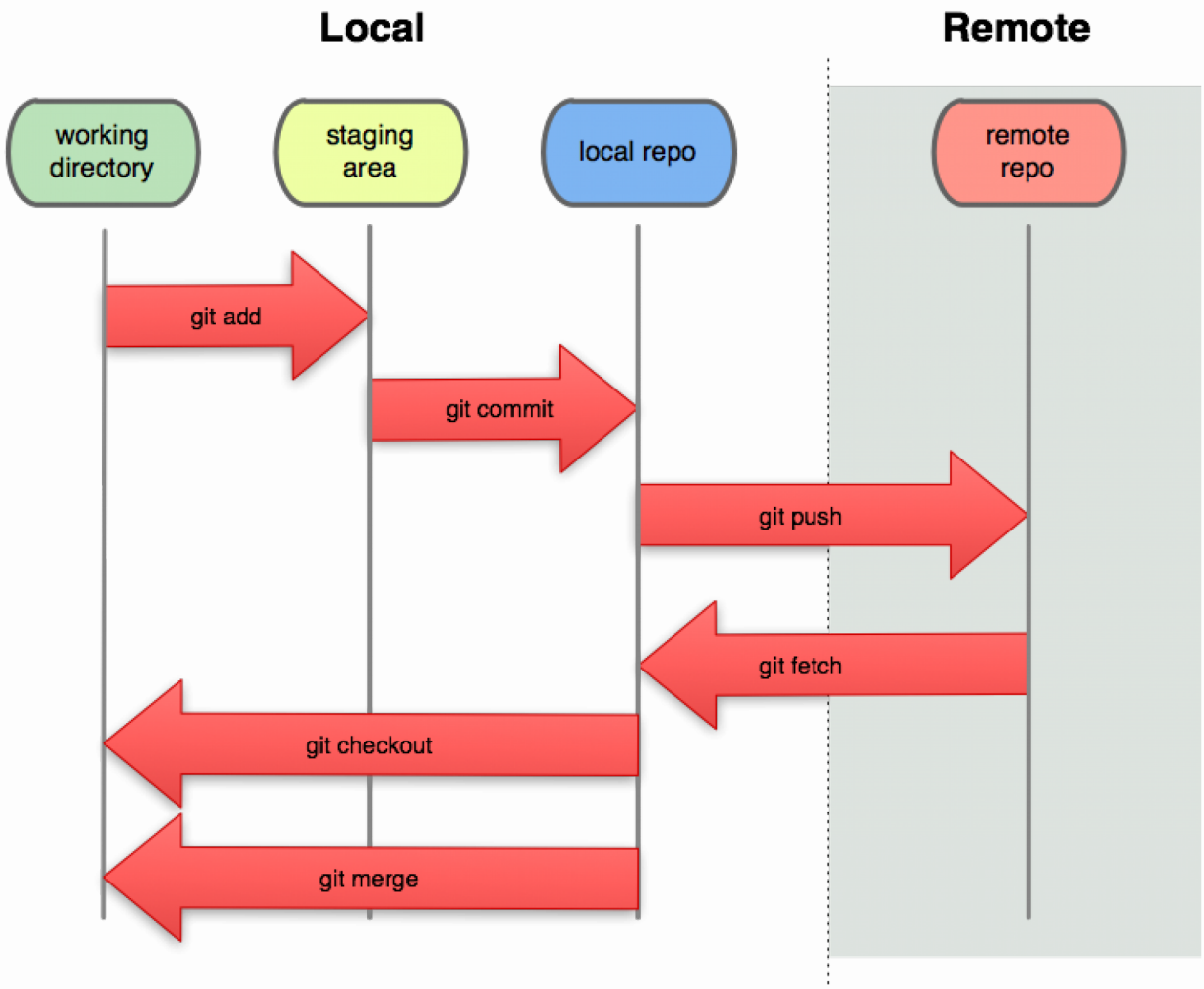
- Access the "Extension" menu in the left sidebar, search for each extension by its name, then install them.
- Jupyter will allow us to use the **IPython** console which is much more convenient for running and inspecting code outputs in a more interactive way.
- A Github Workspace **can be accessed from your native VSCode install with the extension of the same name**.

1.4 Git and GitHub

VSCode supports **version control** using Git version control: at the lower left corner, we can see which branch - something like a "container" storing a particular version of our code - in our version control system we're currently in (normally, if you didn't change into another branch, it's the one called `main`).

- What is version control? Version control allows you to track and control changes to your code. Its benefits can't be overstated:
  - it gives you and your team a single source of truth - one shared reality: everyone is working with the same set of data (files, code, etc.) and can access it in a common location,
  - it enables parallel development which becomes important as the need to manage multiple versions of code and files grows,
  - it maintains a full file history - nothing is lost, if mistakes happen, or older code regains importance,
  - it supports automation within the development process by automating tasks, such as testing and deployment,
  - it enables better team work by providing visibility into who is working on what,

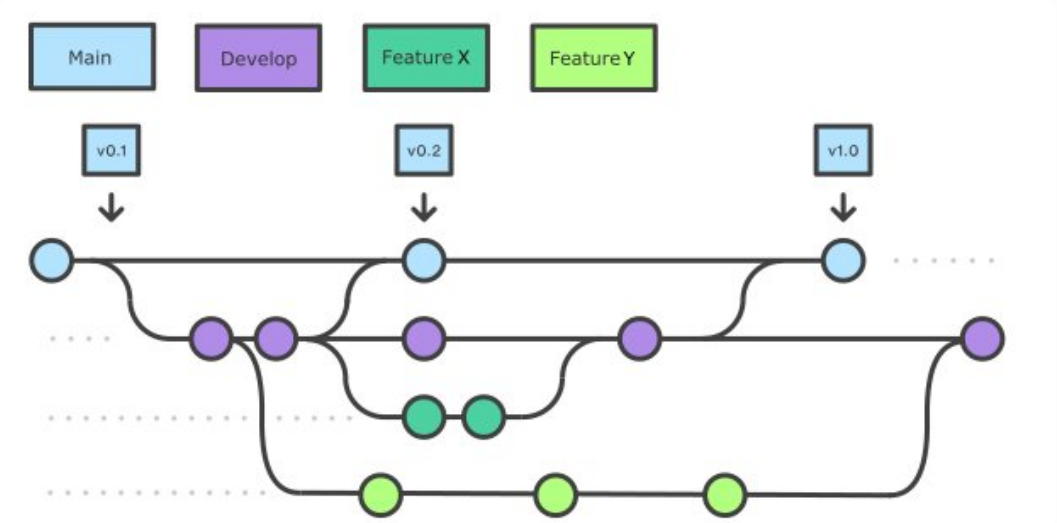
    - it is particularly effective for tracking text-based files (e.g. source code files, CSV, Markdown, HTML, CSS, Tex).
- Major components and commands used to interact with different parts in the Git infrastructure:
    - **working directory**: a directory (including any subdirectories) where your project files live and where you are currently working. This area is untracked by Git, if its not explicitly told to save changes which you do by using `git add filename`,
    - **staging area**: once you tell Git to start tracking changes to files, it saves those changes in the staging area. Each new change to the same file needs to be followed by another `git add filename` command to update it in the staging area,
    - **local repository**: this is where Git wraps together all your changes from the staging area, if you use the `git commit -m "some message indicating what you commit/had changed"` command. Each commit is a new, permanent "snapshot" (checkpoint, or record) of your project in time which you can share and get back to.
        - `git status` allows you to check the current status of your working directory and local repository, e.g., whether there are files which have been changed in the working directory, but not staged for commit (another command you'd probably use very often).
    - **remote repository** - this is a version of your project that is hosted somewhere on the Internet (e.g. on GitHub, GitLab or elsewhere). While you may version-control your local repository, you still run the risk of losing your work, if your machine breaks. When you use a remote repository, you'll need to push your changes from your local repository using `git push origin branch-name`, and, if collaborating with other people, pulling their changes using `git pull` or `git fetch` to keep your local repository in sync with others.
        - The key difference between `git fetch` and `git pull` is that the latter copies changes from a remote repository directly into your working directory, while `git fetch` copies changes only into your local Git repo.



Git workflow from PNGWing.

- Working with different **branches**: whenever you add a separate and self-contained piece of work, it's best practice is to use a new branch (called **feature branch**) for it, and merge it later on with the main branch as soon as it's safe to do so. Each feature branch should have a suitable name that conveys its purpose (e.g. “issue-fix23”).
  - Using different branches enables
    - the main branch to remain stable while you and the team explore and test the new (not-yet-functional) code on a feature branch,
    - you and other collaborators to work on several features at the same time without interfering with each other.
  - Normally, we'd have
    - a **main branch** (most often called `main`) which is the version of the code that is fully tested, stable and reliable,
    - a **development branch** (often called `develop` or `dev` by convention) that we use for work-in-progress code. Feature branches get first merged into `develop` after having been thoroughly tested. Once `develop` had been tested with the new features, it will get merged into `main`.

- Switching between/merging different branches:
  - `git branch develop` creates a new branch called `develop`,
  - `git merge branch-name` allows you to merge `branch-name` with the one you're currently in,
  - `git branch` tells you which branch you're currently in (something you'd check probably very frequently) as well as gives you a list of which branches exist (the one you're in is denoted by a star symbol),
  - `git checkout branch-name` allows you to switch from your current branch into `branch-name`.



Git feature branches, adapted by original course creators from Git Tutorial by sillevl (Creative Commons Attribution 4.0 International License)

> **! FOLLOW ALONG IN YOUR CODESPACE ! Ascertain the existence of the `main` and `develop` branches**

1.5 Creating virtual environments using `venv`

In `inflammation/models.py`, we see that we import two external libraries:

```
from matplotlib import pyplot as plt
import numpy as np
```

- In Python, we very often use external libraries that don't come as part of the standard Python distribution. Sometimes, you may need a specific version of an external library (e.g., because your code uses a feature, class, or function from an older version that has been updated since), or a specific version of Python interpreter.
- Consequently, each project may require a different setup and different dependencies, so it will be very useful to keep those different configurations separate to avoid

confusion between projects –> that's where a virtual environment comes in handy: a virtual environment **creates an isolated "working copy" of a given software project** that uses a specific version of Python interpreter together with specific versions of a number of external libraries installed into that virtual environment. You can create a self-contained virtual environment for any number of separate projects. The specifics of your virtual environment can be then reproduced by someone else using their own virtual environment.

- Virtual environments make it a lot easier for collaborators to use and work on your project's code, as potential installation problems and package version clashes are spared.
- Most modern programming languages are able to use virtual environments to isolate libraries for a specific project and make it easier to develop, run, test and share code with others; they are not a specific feature of Python.
- A list of commonly used Python virtual environment managers (we'll use `venv`):
  - `venv`,
  - `virtualenv`,
  - `pipenv`,
  - `conda`,
  - `poetry`.

> **! FOLLOW ALONG IN YOUR CODESPACE ! Let's create our virtual environment**

This can be achieved with the following commands:

```
$ python3 -m venv venv          # creating a new folder called "venv",
                                # and instantiating a virtual environment
                                # equally called "venv"

$ source venv/bin/activate      # activate virtual environment
(venv) $ which python3          # check whether Python from venv is used

Output:
/workspaces/python-intermediate-inflammation/venv/bin/python3

(venv) $ deactivate             # deactivate virtual environment
```

Our code depends on two external packages (`numpy`, `matplotlib`). We need to install those into the virtual environment to be able to run the code using a **package manager tool** such as `pip`:

```
(venv) $ pip3 install numpy matplotlib
```

When you are collaborating on a project with a team, you will want to make it easy for them to replicate equivalent virtual environments on their machines. With pip, virtual environments can be exported, saved and shared with others by creating a file called, e.g., `requirements.txt` (you can name as you like, but it's practice to label this file as "requirements") in your current directory, and producing a list of packages that have been installed in the virtual environment:

```
(venv) $ pip3 list                          # view packages installed
(venv) $ pip3 freeze > requirements.txt     # produce list of packages
```

If someone else is trying to use your library within their own virtual environment, instead of manually installing every dependency, they can just use the command below to install everything specified in the `requirements.txt` file.

```
(venv) $ pip3 install -r requirements.txt    # install packages from
                                             # requirements file
```

Let's check the status of our repository using `git status`. We get the following output:

```
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        requirements.txt
        venv/

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

While you do not want to commit the newly created directory `venv` and share it with others as it's is specific to your machine and setup only (containing local paths to libraries on your system specifically), you will want to share `requirements.txt` with your team as this file can be used to replicate the virtual environment on your collaborators' systems.

Checking the state of the codebase with `git status` and using `.gitignore`:

To tell Git to ignore and not track certain files and directories, you need to specify them in the `.gitignore` text file in the project root. You can also ignore multiple files at once that match a pattern (e.g. "*.jpg" will ignore all jpeg files in the current directory). Let's add the necessary lines into the `.gitignore` file:

```
# Virtual environments
venv/
.venv/
```

Checking the changes before committing

The most straightforward way to do so it to run `git diff` under the repository's tree.

- For each file that was modified, `git diff` will highlighted the parts that were removed and / or added
- VSCode has an integrated `diff` GUI tool that is more intuitive to use (left sidebar, third icon usually below the looking glass.)

Let's make a first commit to our local repository:

```
$ git add .gitignore requirements.txt
$ git commit -m "Initial commit of requirements.txt. Ignoring virtual env. folde
```

1.6 Virtual envs, dependencies and reproducibility

- Manually specified `requirement.txt` vs `pip freeze`:
  - the former will make `pip` fetch the latest build of the specified library during `pip install -r`,
  - while using the latter will force it to fetch the builds at version specified during export.
  - **Updated packages sometimes introduce breaking change compared to the older version used during development**.
  - For an easier re-use, it is desirable to be as specfic as possible with the versions of the package used.
- `(venv) pip` is tied to the version of native python that was used to created said env. As mentionned previously, new version of Python itself can also break working code from a previous version.
  - Package managers such as *Anaconda* or *Poetry* allows us to work with a specific version of `python`: (e.g. `conda create -n venv python==3.9 -y`), on top of having similair mechanics to `pip freeze`.
  - Some caveats of more advanced package managers:
    - a steeper learning curve for their usage,
    - Anaconda can take up a lot of time when resolving dependency conflicts
- Sometimes, even the remote files of a given dependencies can get removed, leaving some projects vulnerables to non-reproducibility.
  - Tools such as *Docker* takes dependency management one step further by creating a *virtual machine* with local copies of the environment and package required to run a specific, which an then be shared along with the code to increase the likelihood of reproducibility.

    - Docker takes some time to get used to, and might be too heavy handed in early phases of a project, is better left once a clean and working version that is candidate for open-sourcing / publication is available.
    - Cloud computing services like AWS, High-Performance Computer (clusters, supercomputers) usually rely on container based workflows due to the need to allow a large amount of users to use the same resources while also preserving privacy.

2. Testing Ensuring correctness of software (at scale)

Why is testing good?

- We want to increase chances that we're writing **correct code** for various possible inputs, and that, if it's a contribution to an existing codebase, it doesn't break it. (People often underestimate, or are not aware of, the risk of writing incorrect code.)
    - E.g., for the sake of argument, if each line we write has a 99% chance of being right, then a 70-line program will be wrong more than half the time.
- Debugging code is a hard reality - it will happen (a lot), no matter the coding skills. Our future selves and collaborators will likely have a much **easier time debugging**, if tests had been written beforehand.
- Obviously, writing code with tests takes some additional effort compared to writing code without tests. However, the effort pays off quickly, and likely **saves huge amounts of time in the medium to long term** by allowing us to more comprehensively and rapidly find errors, as well as giving us greater confidence in the correctness of our code.

We'll get into:

- **Automatically testing software** using a test framework called Pytest that helps us structure and run automated tests.
    - Manual and automatic code testing are not mutually exclusive, but complement each other. Manual testing is particularly good for checking graphical user interfaces and comparing visual outputs against inputs, but very time-consuming for many other things one would like to test for.
- **Scaling up unit testing** by making use of test parameterisation to increase the number of different test cases we can run.
- **Continuous integration** (CI) for automated using GitHub Actions - a CI infrastructure that allows us to automate tasks when some change happens to our code, such as running tests when a new commit is made to a code repository.

- **Debugging code** using the integrated debugger in VSCode which helps us locate an error in our code while it is running, and fix it.

2.1 Unit Testing

Let's create a new branch called `test-suite` where we'll write our tests. It is good practice to write tests at the same time when we write some new code on a feature branch. But since the code already exists, we're creating a feature branch just for writing tests this time. Generally, it is encouraged to use branches for even small bits of new work.

> **! FOLLOW ALONG IN YOUR CODESPACE ! Let's generate a new feature branch**:

```
$ git checkout develop
$ git branch test-suite
$ git checkout test-suite
```

Now let's look at the `daily_mean()` function in `inflammation/models.py`. It calculates the daily mean of inflammation values across all patients. Let's first think about how we could manually test this function.

One way to test whether this function does the right thing is to think about which output we'd expect given a certain input. We can **test** this **manually** by creating an input and output variable, and use, e.g., `npt.assert_array_equal()` to check whether the outcome of `daily_mean()` given the input variable matches the output variable.

- To use `daily_mean()`, we need to import it. To import it, we need to instantiate a directory for the codespace.
- Either do it using a Jupyter Notebook, or open the IPython console (right-clicking on `import numpy as np`, and choosing `Run in Interactive Window`, and `Run Selection/Line in Interactive Window`).

```
1  import os
2
3  # get the current working directory
4  os.getcwd()
5
6  import numpy as np
7  import numpy.testing as npt
8  from inflammation.models import daily_mean
9
10 test_input = np.array([[1, 2], [3, 4], [5, 6]])
11 test_result = np.array([3, 4])
12 npt.assert_array_equal(daily_mean(test_input), test_result) # Runs without
13 # print(daily_mean(test_input)) # confirm that it matches `test_result
```

We can think about multiple pairs of expected output given a certain input:

```
test_input = np.array([[2, 0], [4, 0]])
test_result = np.array([2, 0])
npt.assert_array_equal(daily_mean(test_input), test_result)

test_input = np.array([[0, 0], [0, 0], [0, 0]])
test_result = np.array([0, 0])
npt.assert_array_equal(daily_mean(test_input), test_result)
```

However, we get a mismatch between input and output for the first test:

```
...
AssertionError:
Arrays are not equal

Mismatched elements: 1 / 2 (50%)
Max absolute difference: 1.
Max relative difference: 0.5
 x: array([3., 0.])
 y: array([2, 0])
```

The reason here is that one of our specified outputs is wrong - which reminds us that tests themselves can be written in a wrong way, so it's good to keep them as simple as possible so as to minimize errors.

We could put these tests in a separate script to automate running them. However, a Python script stops at the first failed assertion, so if we get one no matter why, all subsequent tests wouldn't be run at all –> this calls for a **testing framework** such as `Pytest` where we

- define our tests we want to run as **functions**,
- are able to run a **plethora of tests** at the same time regardless of whether test failures had occured, and
- get an **output summary** of the test functions.

Unit Testing

Let's look at `tests/test_models.py` where we see one test function called `test_daily_mean_zeros()`:

```
def test_daily_mean_zeros():
    """Test that mean function works for an array of zeros."""
    from inflammation.models import daily_mean

    test_input = np.array([[0, 0],
                           [0, 0],
                           [0, 0]])
    test_result = np.array([0, 0])

    # Need to use NumPy testing functions to compare arrays
    npt.assert_array_equal(daily_mean(test_input), test_result
```

Generally, each test function requires

- inputs, e.g., the `test_input` `NumPy` array,
- execution conditions, i.e., what we need to do to set up the testing environment to run our test, e.g., importing the `daily_mean()` function so we can use it (we only import the necessary library function we want to test within each test function),
- a testing procedure, e.g., running `daily_mean()` with our test_input array and using `np.assert_array_equal()` to test its validity,
- expected outputs, e.g., our test_result NumPy array that we test against,
- if using `PyTest`, the letters 'test_' at the beginning of the function name.

> **! FOLLOW ALONG IN YOUR CODESPACE ! Let's install `PyTest` :**

```
pip install pytest
```

We can then run `PyTest` in the CLI...

```
python -m pytest tests/test_models.py # or
# pytest tests/test_models.py
```

... and get the following output:

```
============================================================ test session starts ====
platform linux -- Python 3.10.8, pytest-7.3.2, pluggy-1.2.0
rootdir: /workspaces/python-intermediate-inflammation
collected 2 items

tests/test_models.py ..

============================================================= 2 passed in 1.06s =====
```

We can also test single testing functions in our `test_models.py` file. To do that, we need to configure our testing set up by clicking on the testing icon, choosing `Configure Python Test`, then `Pytest`, and then the folder the tests are in.

- You will now be able to see a green arrow to the right of any testing function. You can click on it, and the test will be run.

> **! TASK 1 ! Write a new test case that tests the `daily_max()` function**, adding it to `test/test_models.py`.

> **Ascertain we are are under the `test-suite` branch, which was created from `develop`**

- You could choose to write your functions very similarly to `daily_mean()`, defining input and expected output variables followed by the equality assertion.
- Use test cases that are suitably different.
- Once added, run all the tests again with `python -m pytest tests/test_models.py`, and have a look at your new tests pass.

▶ Example solution

Now that we have added a new test, we would like to commit this new changes from the `test-suite` branch to the `develop` branch. Namely:

- regenerate the `requirement.txt` since we added new dependencies (`pip freeze`)
- track the new changes with `git add` (staging phase)
- commit the new changes with `git commit`
- use `git merge` to incorporate the changes from `test-suite` to `develop`

▶ Example solution

In case of mistakes

- Use `git log` to check the history of commits and changes
- Use `git reset --hard <commit hash>` to revert to previous state of the code

2.2 Scaling up unit tests

We had used two different testing functions to distinguish between integer and string inputs. Writing a separate test functions to test the same function for different cases is quite inefficient - that's where **test parameterisation** comes in handy.

Instead of writing a separate function for each different test, we can parameterise the tests with multiple test inputs, e.g., in `tests/test_models.py`, we can rewrite the `test_daily_mean_zeros()` from above and `test_daily_mean_integers()`

```
@pytest.mark.parametrize(
    "test, expected",
    [
        ([ [0, 0], [0, 0], [0, 0] ], [0, 0]),
        ([ [1, 2], [3, 4], [5, 6] ], [3, 4]),
    ])
def test_daily_mean(test, expected):
    """Test mean function works for array of zeroes and positive integers."""
    from inflammation.models import daily_mean
    npt.assert_array_equal(daily_mean(np.array(test)), np.array(expected))
```

- We need to provide input and output names - e.g., `test` for inputs, and `expected` for outputs -, as well as the inputs and outputs themselves that correspond to these names. Each row within the square brackets following the `"test, expected"` arguments corresponds to one test case. Let's look at the first row:
  - `[ [0, 0], [0, 0], [0, 0] ]` would be the input, corresponding to the input name `test`,
  - `[0, 0]` would be the output, corresponding to the output name `expected`,
- The `parameterize()` function is a **Python decorator**: A Python decorator is a function that takes as an input a function, adds some functionality to it, and then returns it (more about this in the section on **functional programming**).
  - `parameterize()` is a decorator in that it takes as an input the respective testing function, adds functionality to it by specifying multiple input and expected output test cases, and calling the function over each of these inputs automatically when this test is called.

! TASK 2 ! Rewrite your test functions for `daily_max()` using test parameterisation.

- Make sure you're back in the `test-suite` branch.
- Once added, run all the tests again with `python -m pytest tests/test_models.py`, and have a look at your new tests pass.
- Commit your changes, and merge the `test-suite` branch into the `develop` branch.

▶ Example solution

2.3 Debugging code & code coverage

Debugging code

We can find problems in our code conveniently in VScode using **breakpoints** (points at which we want code execution to stop) and our testing functions.

> **! FOLLOW ALONG IN YOUR CODESPACE !**

- Let's choose a function we want to debug, e.g., `daily_max()`, and set a breakpoint somewhere within that function by left-clicking the space to the left of the line numbers,
- we then click on the testing icon in the VSCode IDE, look for the respective testing function, e.g., `test_daily_max()`, then choose `Debug Test`.
- We can now see the local and global variables in the upper left space of the IDE, and play around with those in the `DEBUG CONSOLE` to check whether our function does what it's supposed to do, e.g., run `np.max(data, axis=0)` and see whether it's giving the expected output (i.e., `array([0, 0])`).
- We can also watch specific variables,
- We can click on the red rectangle at the top to stop the debugging process.

Code coverage

While Pytest is an indispensable tool to speed up testing, it can't help us decide *what* to test and *how many* tests to run.

As a heuristic, we should try to come up with tests that test

- as many functions as possible, with test cases as different from each other as possible,
- rather than, let's say, an endless number of test cases for the same function, and using rather redundant test cases.

This ensures a high degree of **code coverage**. A Python package called `pytest-cov` that is used by Pytest gives you exactly this - the degree to which you've covered your code w. r. t. tests.

> **! FOLLOW ALONG IN YOUR CODESPACE ! Let's install `pytest-cov` and assess code coverage:**

```
$ pip3 install pytest-cov
$ python -m pytest --cov=inflammation.models tests/test_models.py
```

- `--cov` is an additional named argument to specify the code that is to be analysed for test coverage.

- We also specify the file that contains the test for the code to be analysed.

Output:

```
============================================== test session starts ==========
platform linux -- Python 3.10.8, pytest-7.3.2, pluggy-1.2.0
rootdir: /workspaces/python-intermediate-inflammation
plugins: cov-4.1.0
collected 7 items

tests/test_models.py .......

---------- coverage: platform linux, python 3.10.8-final-0 -----------
Name                      Stmts   Miss  Cover
---------------------------------------------
inflammation/models.py        9      2    78%
---------------------------------------------
TOTAL                         9      2    78%


============================================== 7 passed in 1.25s ===========
```

- 78% of our statements in `inflammation.models` are tested. To see which ones have not yet been tested, we can use the following line in the terminal:

```
python -m pytest --cov=inflammation.models --cov-report term-missing tests/test_m
```

Output:

```
============================================== test session starts ==========
platform linux -- Python 3.10.8, pytest-7.3.2, pluggy-1.2.0
rootdir: /workspaces/python-intermediate-inflammation
plugins: cov-4.1.0
collected 7 items

tests/test_models.py .......

---------- coverage: platform linux, python 3.10.8-final-0 -----------
Name                      Stmts   Miss  Cover   Missing
-----------------------------------------------------
inflammation/models.py        9      2    78%   18, 32
-----------------------------------------------------
TOTAL                         9      2    78%


============================================== 7 passed in 0.29s ===========
```

- We'd need to look at lines 18 and 32 to check for the yet untested code.

! TASK 3 ! Clean up and final commit of our `test-suite` changes

```
$ pip3 freeze > requirements.txt
$ git status
# TODO: Update .gitignore with __pycache__ and other indesirable, etc...
$ git add ./
$ git commit -m "Add coverage support"
$ git checkout develop
$ git merge test-suite
```

What is Test Driven Development?
In test-driven development, we first write the tests, and then the code, i.e., the thinking process would go from

- defining the feature we want to implement, writing the tests, and writing only as much code as needed to pass the test,
- rather than defining the feature, writing the code, and then writing the tests.

This way, the set of tests act like a specification of what the code does. The main advantages are:

- writing tests can't be avoided/not prioritized,
- we may get a better idea of how our code will be used before writing it,
- we may refrain from adding things into our code that eventually turn out unnecessary anyway.

Final notes on testing

- A complex program requires a much higher investment in testing than a simple one –> find a good trade-off between code complexity and test coverage.
- Tests can not find every bug that may exist in the code - manual testing will remain a crucial component.
- If using data, try to use as many different datasets as possible to test your code, thereby increasing confidence that it does the correct thing.
- Most software projects increase in complexity as they develop - using automated testing can save us a lot of time, particularly in the long term.

2.4 Continuous Integration

If we're collaborating on a software project with multiple people who push a lot of changes to one of the major repositories, we'd need to constantly pull down their changes to our local machines, and do our tests with the newly pulled down code - this would result in a lot of back and forth, slowing us down quite a bit. That's where **Continuous integration** (CI) comes in handy:

- It is the practice of merging all developers' working copies to a shared main working copy several times a day. When a new change is committed to a repository, CI clones

the repository, builds it if necessary, and runs any tests. Once complete, it presents a report to let you see what happened.

- It thereby gives *early* hints so as to whether there are important incompatibilities between a given working copy and the shared main working copy.
- It also allows us to test whether our code works on different target user platforms.

There are many CI infrastructures and services. We'll be looking at GitHub Actions - which, unsurprisingly, is available as part of GitHub.

GitHub Actions

`YAML` (a recursive acronym which stands for "YAML Ain't Markup Language") is a text format used by GitHub Action workflow files. `YAML` files use

- key-value pairs (values can also be arrays), e.g.,

```yaml
name: Kilimanjaro
height_metres: 5892
first_scaled_by: Hans Meyer

first_scaled_by:
  - Hans Meyer
  - Ludwig Purtscheller
```

- maps which allow us to define nested, hierarchical data, e.g.,

```yaml
height:
  value: 5892
  unit: metres
  measured:
    year: 2008
    by: Kilimanjaro 2008 Precise Height Measurement Expedition
```

Let's set up CI using GitHub Actions: with a GitHub repository, there's a way we can set up CI to run our tests automatically when we commit changes.
To do this, we need to add a new file in a particular directory of our repository (make sure you're on the `test-suite` branch).

> **! TASK 4 ! Adding a basic testing workflow with Continuous Integration using Github Actions**

Let's create a new directory `.github/workflows` which is used specifically for GitHub Actions, as well as a new file called `main.yml` :

```
$ mkdir -p .github/workflows
$ cd .github/workflows
```

```
$ vim main.yml # emphasis on the .yml, not .yaml
```

In the `main.yml`, we'll write the following:

```
name: CI

# We can specify which Github events will trigger a CI build
on: push

jobs: # Can have multiple jobs, run in parallel

  build:

    # we can also specify the OS to run tests on
    runs-on: ubuntu-latest

    # a job is a seq of steps
    steps:

    # Next we need to checkout out repository, and set up Python
    # A 'name' is just an optional label shown in the log - helpful to clar
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Set up Python 3.9
      uses: actions/setup-python@v2
      with:
        python-version: "3.9"

    - name: Install Python dependencies
      run: |
        python3 -m pip install --upgrade pip
        pip3 install -r requirements.txt

    - name: Test with PyTest
      run: |
        python -m pytest --cov=inflammation.models tests/test_models.py
```

- `name: CI`: name of our workflow
- `on: push`: indication that we want this workflow to run when we push commits to our repository.
- `jobs: build:` the workflow itself is made of a single job named `build`, but could contain any number of jobs after this one, each of which would run in parallel.
- `runs-on: ubuntu-latest`: statement about which operating systems we want to use, in this case just Ubuntu.
- `steps:` the steps that our job will undertake in turn to 1) set up the job's environment (think of it as a freshly installed machine, albeit virtual, with very little installed on it) and 2) run our tests. Each step has a name (which you can choose to your liking) and a way to be executed (as specified by `uses` / `run`).

- `name: Checkout repository for the job`: use a GitHub Action called `checkout`
- `name: Set up Python 3.9`: here, we use the `setup-python` Action, indicating that we want Python version 3.9.
- `name: Install Python dependencies`: install latest version of `pip`, dependencies, and our `inflammation` package: In order to locally install our inflammation package, it's good practice to upgrade the version of pip that is present first, then we use pip to install our package dependencies.
- `name: Test with PyTest`: finally, we let PyTest run our tests in `tests/test_models.py`, including code coverage.

Finally, lets commit the changes and push to trigger the actions
First, let's run the following commands:

```
$ git add .github/workflows
$ git commit -m "Added GH CI support"
$ git push
```

Then head under the "Actions" tab of your repository to check the build results.

Scaling up testing using build matrices

To address whether our code works on different target user platforms (e.g., Ubuntu, Mac OS, or Windows), with different Python installations (e. g., 3.8, 3.9 or 3.10), we can use a feature called **build matrices**. Doing our tests across all these platforms and program versions would take *a lot* of time - that's where a build matrix comes in handy.

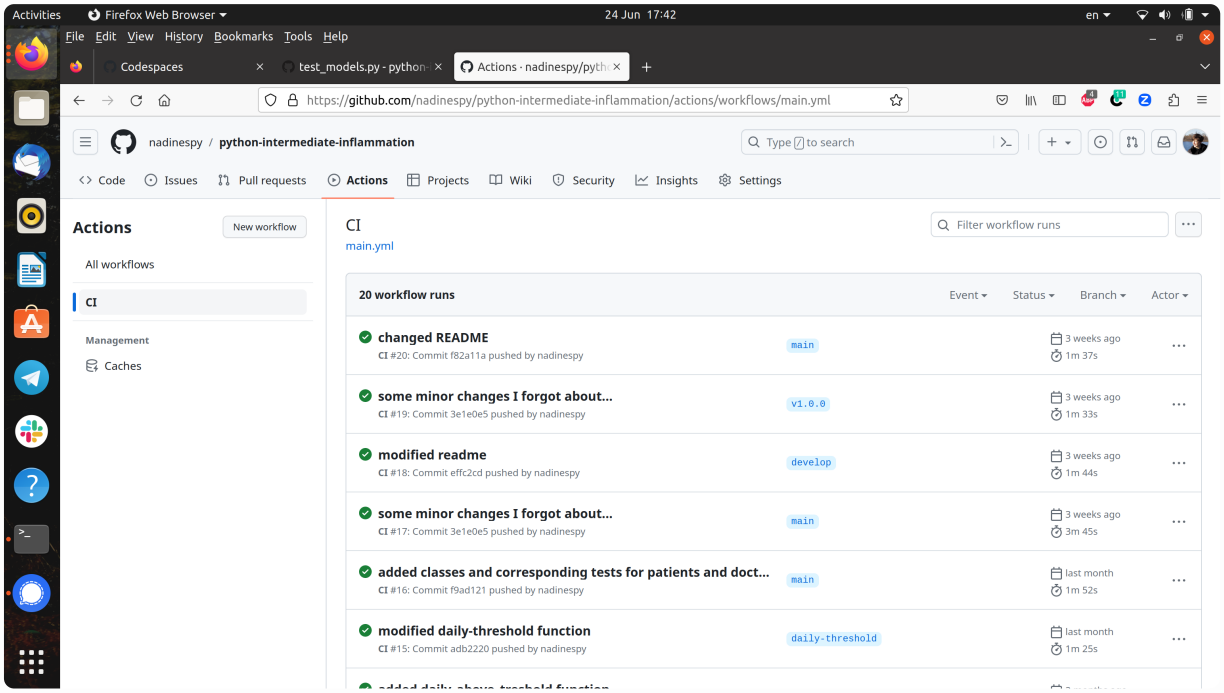
- Using a build matrix inside of our `main.yml`, we can specify environments (such as operating systems) and parameters (such as Python versions), and new jobs will be created that run our tests for each permutation of these.
- We first define a `strategy` as a `matrix` of operating systems and Python versions within `build`. We then use `matrix.os` and `matrix.python-version` to reference these configuration possibilitiesjob

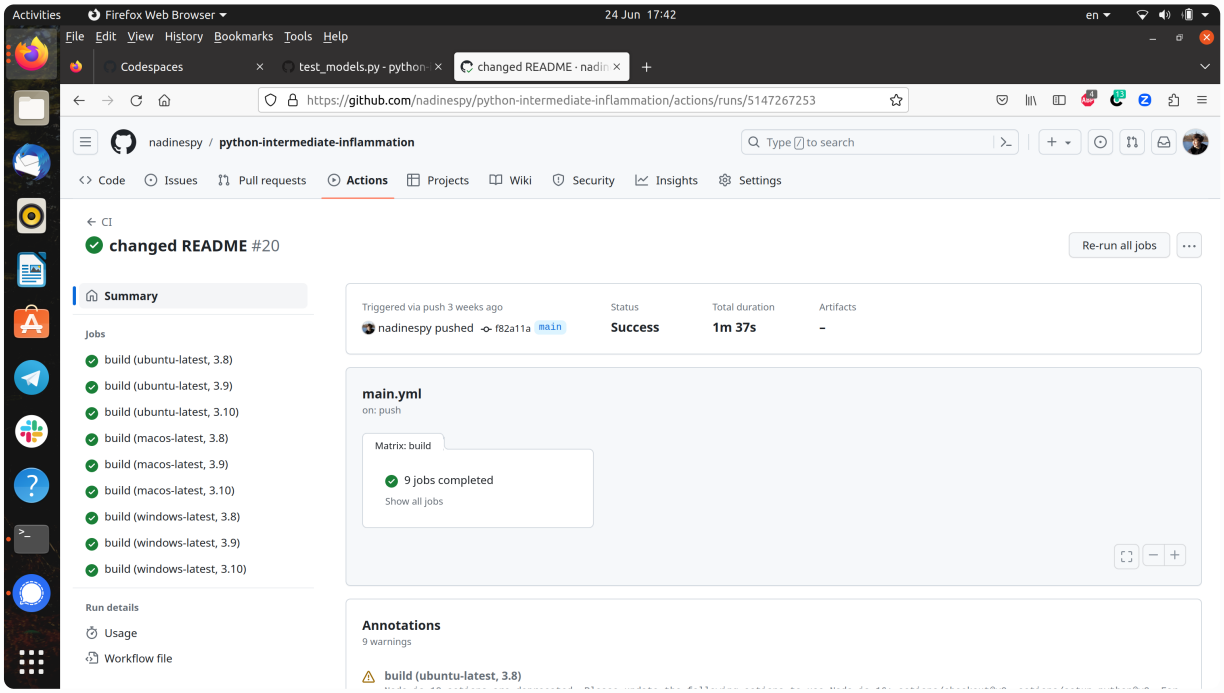
```yaml
name: CI

# We can specify which Github events will trigger a CI build
on: push

# now define a single job 'build' (but could define more)
jobs:

  build:

    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
        python-version: ["3.8", "3.9", "3.10"]

    runs-on: ${{ matrix.os }}

    # a job is a seq of steps
    steps:

    # Next we need to checkout out repository, and set up Python
    # A 'name' is just an optional label shown in the log - helpful to clar
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: ${{ matrix.python-version }}
```

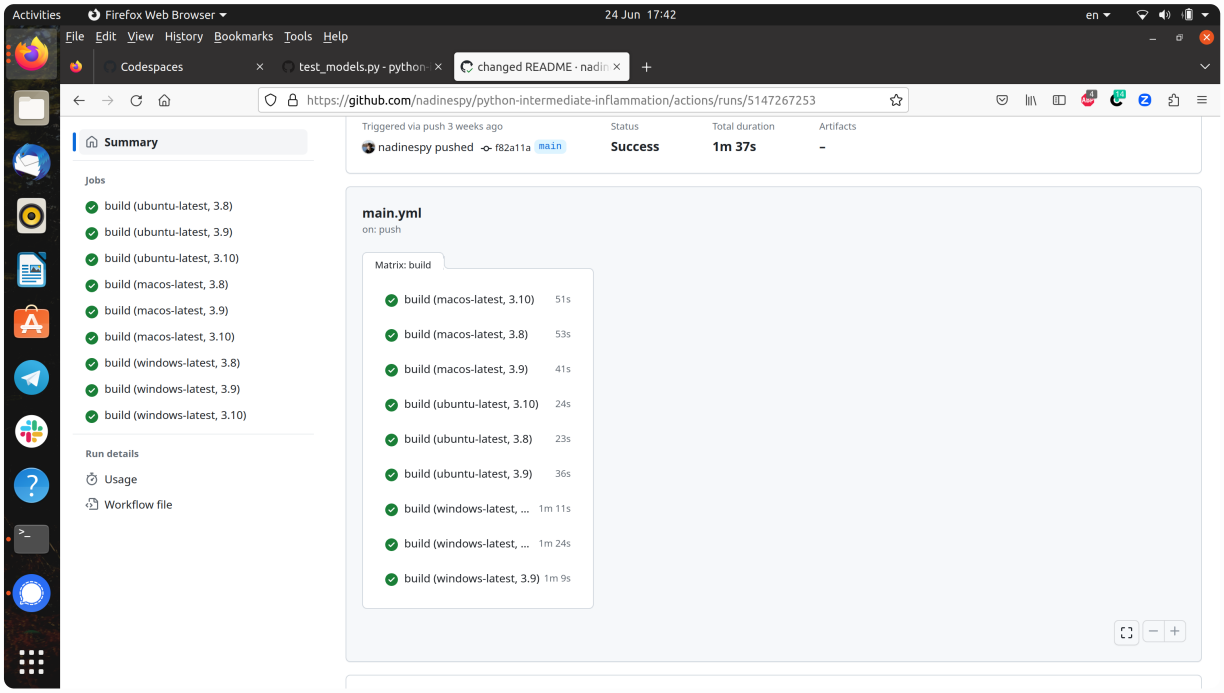
Let's add the new folder/file to the local repository and merge with `develop`:

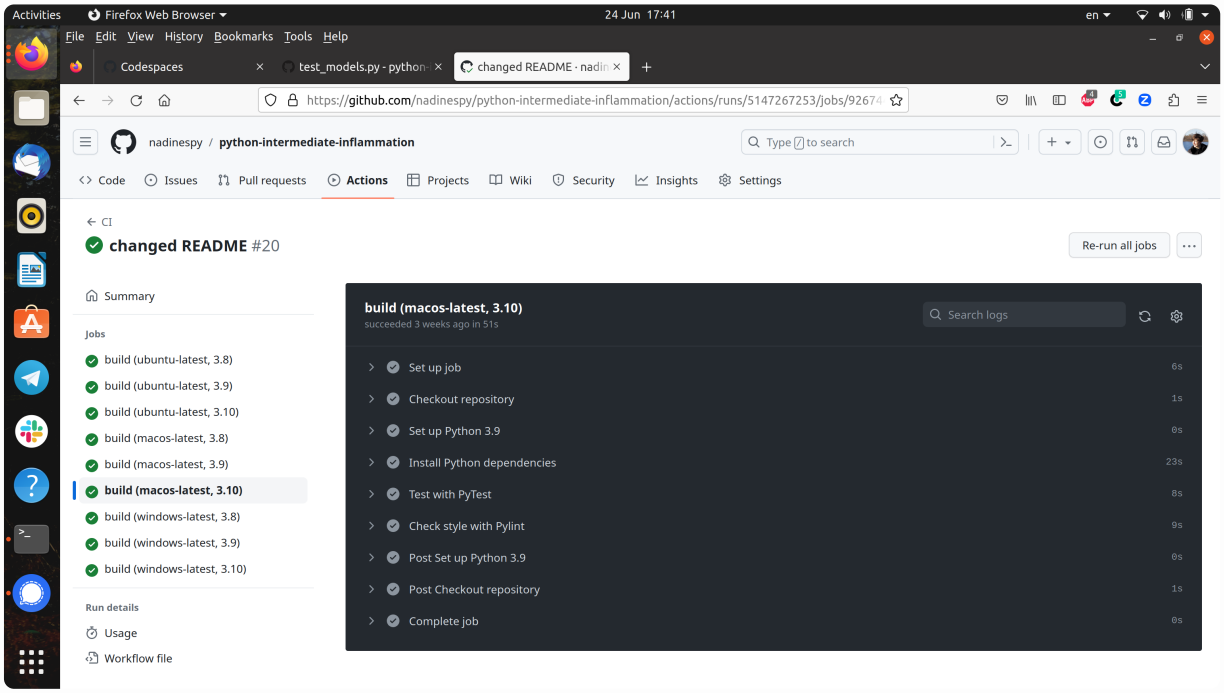
```
$ git add .github
$ git commit -m "Add GitHub Actions configuration & build matrix for os and Pyth
$ git switch develop
$ git merge test-suite
```

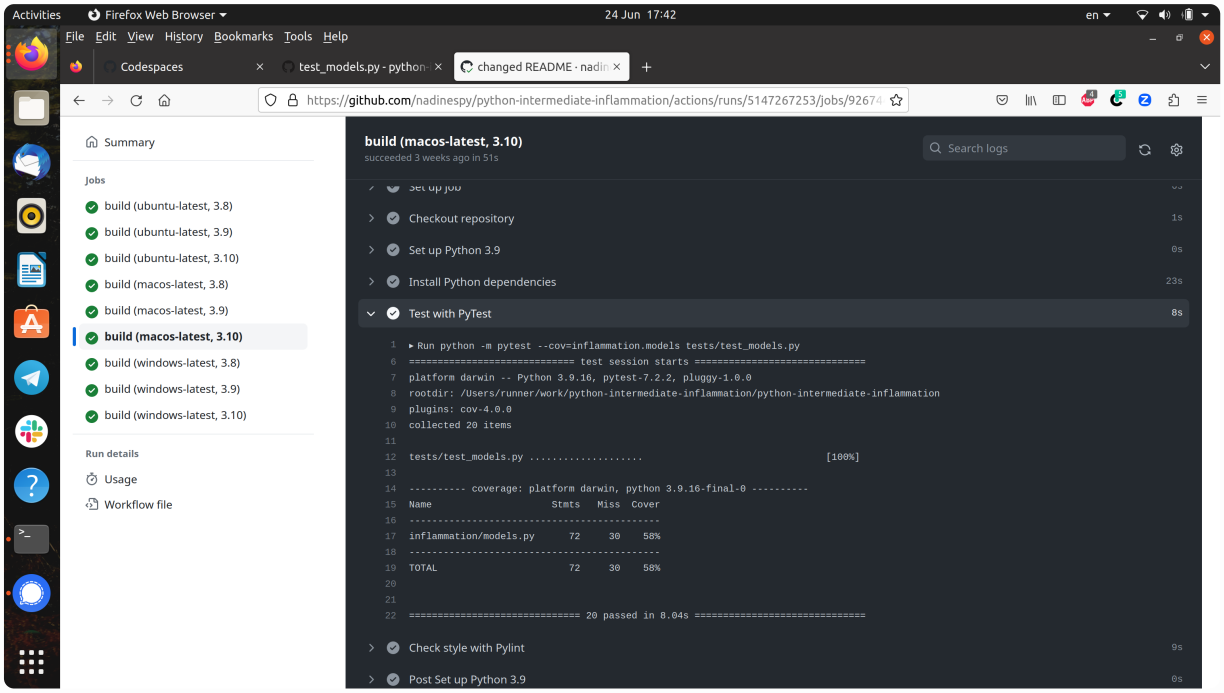
Whenever you push your changes to a *remote* repository, GitHub will run CI as specified by the `main.yml`.
You can check its status on the website of your remote repository under `Actions`.
For each push, you'd get a report about which of the steps have been successfully/unsuccessfully taken.


Activities
Firefox Web Browser
24 Jun 17:42
en
File Edit View History Bookmarks Tools Help
Codespaces
test_models.py - python-
Actions · nadinespy/pyth
https://github.com/nadinespy/python-intermediate-inflammation/actions/workflows/main.yml
nadinespy / python-intermediate-inflammation
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Actions
New workflow
All workflows
CI
Management
Caches
CI
main.yml
Filter workflow runs
20 workflow runs
Event
Status
Branch
Actor
changed README
CI #20: Commit f82a11a pushed by nadinespy
main
3 weeks ago
1m 37s
some minor changes I forgot about...
CI #19: Commit 3e1e0e5 pushed by nadinespy
v1.0.0
3 weeks ago
1m 33s
modified readme
CI #18: Commit effc2cd pushed by nadinespy
develop
3 weeks ago
1m 44s
some minor changes I forgot about...
CI #17: Commit 3e1e0e5 pushed by nadinespy
main
3 weeks ago
3m 45s
added classes and corresponding tests for patients and doct...
CI #16: Commit f9ad121 pushed by nadinespy
main
last month
1m 52s
modified daily-threshold function
CI #15: Commit adb2220 pushed by nadinespy
daily-threshold
last month
1m 25s


Activities
Firefox Web Browser
24 Jun 17:42
en
File Edit View History Bookmarks Tools Help
Codespaces
test_models.py - python-
changed README · nadin
https://github.com/nadinespy/python-intermediate-inflammation/actions/runs/5147267253
nadinespy / python-intermediate-inflammation
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
CI
changed README #20
Re-run all jobs
Summary
Jobs
build (ubuntu-latest, 3.8)
build (ubuntu-latest, 3.9)
build (ubuntu-latest, 3.10)
build (macos-latest, 3.8)
build (macos-latest, 3.9)
build (macos-latest, 3.10)
build (windows-latest, 3.8)
build (windows-latest, 3.9)
build (windows-latest, 3.10)
Run details
Usage
Workflow file
Triggered via push 3 weeks ago
nadinespy pushed f82a11a main
Status
Success
Total duration
1m 37s
Artifacts
-
main.yml
on: push
Matrix: build
9 jobs completed
Show all jobs
Annotations
9 warnings
build (ubuntu-latest, 3.8)


Activities
Firefox Web Browser
24 Jun 17:42
en
File Edit View History Bookmarks Tools Help
Codespaces
test_models.py - python
changed README · nadin
https://github.com/nadinespy/python-intermediate-inflammation/actions/runs/5147267253
Summary
Jobs
build (ubuntu-latest, 3.8)
build (ubuntu-latest, 3.9)
build (ubuntu-latest, 3.10)
build (macos-latest, 3.8)
build (macos-latest, 3.9)
build (macos-latest, 3.10)
build (windows-latest, 3.8)
build (windows-latest, 3.9)
build (windows-latest, 3.10)
Run details
Usage
Workflow file
Triggered via push 3 weeks ago
nadinespy pushed f82a11a main
Status
Success
Total duration
1m 37s
Artifacts
-
main.yml
on: push
Matrix: build
build (macos-latest, 3.10) 51s
build (macos-latest, 3.8) 53s
build (macos-latest, 3.9) 41s
build (ubuntu-latest, 3.10) 24s
build (ubuntu-latest, 3.8) 23s
build (ubuntu-latest, 3.9) 36s
build (windows-latest, ... 1m 11s
build (windows-latest, ... 1m 24s
build (windows-latest, 3.9) 1m 9s


Activities
Firefox Web Browser
24 Jun 17:41
en
File Edit View History Bookmarks Tools Help
Codespaces
test_models.py - python
changed README · nadin
https://github.com/nadinespy/python-intermediate-inflammation/actions/runs/5147267253/jobs/92674
nadinespy / python-intermediate-inflammation
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
CI
changed README #20
Re-run all jobs
Summary
Jobs
build (ubuntu-latest, 3.8)
build (ubuntu-latest, 3.9)
build (ubuntu-latest, 3.10)
build (macos-latest, 3.8)
build (macos-latest, 3.9)
build (macos-latest, 3.10)
build (windows-latest, 3.8)
build (windows-latest, 3.9)
build (windows-latest, 3.10)
Run details
Usage
Workflow file
build (macos-latest, 3.10)
succeeded 3 weeks ago in 51s
Search logs
Set up job 6s
Checkout repository 1s
Set up Python 3.9 0s
Install Python dependencies 23s
Test with PyTest 8s
Check style with Pylint 9s
Post Set up Python 3.9 0s
Post Checkout repository 1s
Complete job 0s



You may also look into these resources on unit testing, scaling it up, and continuous integration:

- An introduction to unit testing
- Scaling up unit testing using parameterisation
- Automating unit testing with Continuous Integration

3. Software Design Continued here