GIT en GITHUB

Status: doornemen en misschien splitsen in 2 modules?

1 – Wat is versiecontrole?

? Leerdoelen

- Je begrijpt wat versiecontrole is en waarom het essentieel is bij softwareontwikkeling.
- Je kunt een eenvoudig project opslaan in verschillende versies met Git.
- Je kunt teruggaan naar een vorige versie van je project.
- Je werkt samen in duo’s om de voordelen van versiecontrole te ontdekken.

? Uitleg

Stel: je werkt aan een Python-project en maakt elke dag kleine aanpassingen. Zonder versiecontrole heb je al snel tientallen bestanden, zoals *versie_definitief_v3.2_echt_laatste.py*. Wat als je collega per ongeluk iets verwijdert, of jijzelf iets overschrijft? Met **versiecontrole** kun je elk moment terug in de tijd — alsof je een tijdmachine voor je code hebt.

Versiecontrole betekent dat elke wijziging aan je project wordt opgeslagen, samen met de datum, auteur en beschrijving van de wijziging. Zo kun je altijd zien wie wat heeft gedaan, en kun je vorige versies herstellen.

Git: jouw lokale tijdmachine

Met **Git** wordt versiecontrole automatisch geregeld. Git bewaart ‘snapshots’ van je project, ook wel *commits* genoemd. Elke commit is een momentopname van de hele projectmap.

```
# Een nieuw Git-project starten
git init

# Bestanden toevoegen aan de versiecontrole
git add .

# Een nieuwe "snapshot" (commit) maken
```

```
git commit -m "Eerste versie van mijn programma"

# Aanpassingen maken in je code en een nieuwe commit maken
git commit -am "Functie toegevoegd voor berekeningen"
```

Teruggaan naar een eerdere versie

Je kunt met Git niet alleen zien wat je hebt veranderd, maar ook **terugrollen** naar een vorige versie van je project.

```
# Alle eerdere versies bekijken
git log

# Teruggaan naar een eerdere commit (tijdelijk)
git checkout commit_id

# Teruggaan naar de laatste versie (main branch)
git switch main
```

Zo kun je fouten herstellen zonder bestanden te verliezen. Dit maakt Git onmisbaar bij professioneel samenwerken aan software.

? Voorbeeld

```
# versie 1
print("Hallo wereld!")

# versie 2
naam = input("Wat is je naam? ")
print("Hallo", naam)

# versie 3
print("Welkom terug,", naam, "! Hoe gaat het vandaag?")
```

Elke versie kun je committen, zodat je later terug kunt naar een eerdere staat, bijvoorbeeld naar de versie zonder invoerfunctie.

?? Opdracht – Versiebeheer in actie (duo-opdracht)

1. Werk in duo’s.
2. Maak een nieuwe map op je computer, bijvoorbeeld `git_oefening`.
3. Maak een eenvoudig Python-script `groet.py` en voeg het toe aan een Git-repository.
4. Maak 3 commits waarin je het script uitbreidt met nieuwe functies (bijv. gebruikersinvoer, extra printregels).
5. Gebruik `git log` om alle versies te bekijken.
6. Kies een oudere commit en gebruik `git checkout` om tijdelijk terug te gaan in de tijd.
7. Controleer in VS Code of de oude versie echt terug is gezet.
8. Keer daarna terug naar de nieuwste versie met `git switch main`.

? Reflectie

- Wat ging er mis toen jullie probeerden samen te werken zonder Git?
- Wat vind je het grootste voordeel van kunnen teruggaan naar een vorige versie?
- Hoe helpt versiecontrole om fouten sneller te herstellen?
- Waarom denk je dat bedrijven *nooit* zonder versiecontrole werken?

? Inleveren

- Lever één screenshot in waarop te zien is dat jullie 3 commits hebben gemaakt en dat je via `git log` de versies kunt zien.
- Lever daarnaast een korte tekst in (`reflectie-versiecontrole-<jouwnaam>.txt`) waarin je beschrijft wat je geleerd hebt.
- Noem beide namen van het duo in het reflectiebestand.

2 – Introductie tot Git in Visual Studio Code

? Leerdoelen

- Je weet hoe je Git kunt gebruiken binnen Visual Studio Code.

- Je kunt wijzigingen vastleggen (committen) met een duidelijke beschrijving.
- Je kunt eerdere versies bekijken via de *Timeline* of *Source Control*-tab.
- Je leert samenwerken in duo's aan één projectmap met gedeelde commits.

? Uitleg

In de vorige les heb je geleerd wat versiecontrole is en hoe Git veranderingen opslaat via de opdrachtregel. In deze les gaan we Git gebruiken **via Visual Studio Code (VSC)**. VSC heeft Git standaard ingebouwd: je kunt commits maken, oude versies bekijken en zelfs terugrollen – allemaal met een paar klikken.

Stappenplan

1. Open Visual Studio Code.
2. Maak een nieuwe map aan, bijvoorbeeld `vsc_git_demo`.
3. Klik op **View → Source Control** of op het icoon met het tak-symbool links in de zijbalk.
4. Klik op **Initialize Repository** om Git aan te zetten in deze map.
   Source Control pane
5. Maak een nieuw bestand, bijvoorbeeld `app.py`, en voeg onderstaande code toe.

```
# app.py - versie 1
print("Welkom bij Git in VS Code!")
```

6. Sla het bestand op. In de Source Control-tab zie je nu dat er één wijziging is.
7. Typ een commitbericht, bijvoorbeeld `Eerste versie toegevoegd`, en klik op ✔ Commit.
8. Pas het bestand aan:

```
# app.py - versie 2
naam = input("Wat is je naam? ")
print(f"Hallo {naam}, welkom bij Git in VS Code!")
```

9. Commit opnieuw met een nieuw bericht, bijvoorbeeld `Gebruikersinvoer toegevoegd`.
10. Herhaal dit proces nog één keer met een kleine aanpassing (bijv. een extra printregel).

Commits bekijken en terugrollen

Je kunt alle commits en hun verschillen bekijken via:

- **Source Control → View & History** of via de **Timeline**-tab in de zijbalk.
- Rechtsklik op een vorige commit en kies *Checkout* om tijdelijk terug te gaan.

```
# In terminal kan het ook:
git log              # toont alle commits
git checkout <commit_id>  # ga terug naar een vorige versie
git switch main      # terug naar de huidige versie
```

?? Opdracht – Visual Git in actie (duo-opdracht)

1. Werk in duo's. Eén student maakt de repository aan, de ander bekijkt mee.
2. Maak in Visual Studio Code een nieuw Python-bestand `vsc_demo.py`.
3. Voeg samen drie versies van de code toe (drie commits):

- v1 – print een begroeting;
- v2 – voeg een gebruikersinvoer toe;
- v3 – voeg een berekening of extra functie toe.
- Gebruik telkens een duidelijk commitbericht (bijv. "Input toegevoegd").
- Gebruik de Timeline om de vorige versies te bekijken.
- Ga terug naar een eerdere commit via **Checkout** en test of de oude versie wordt geladen.
- Keer terug naar de nieuwste versie met **Switch Branch → main**.

? Reflectie

- Hoe helpt de visuele Git-interface in VS Code om overzicht te houden?
- Wat vind je makkelijker: werken via de terminal of via de interface?
- Welke informatie is nuttig in een goed commitbericht?
- Hoe zou je dit proces gebruiken als je samenwerkt aan een groter project?

? Inleveren

- Lever een screenshot in van je Source Control-tab waarin minimaal drie commits zichtbaar zijn.
- Lever ook een korte toelichting in (`reflectie-vscode-git-<jouwnaam>.txt`) waarin je beschrijft:
  - Wat je hebt geleerd;
  - Wat het verschil is tussen terminal en VSC-interface;
  - Hoe jullie als duo hebben samengewerkt.

3 – Branches: experimenteren zonder risico

? Leerdoelen

- Je begrijpt wat een **branch** is en waarom branches gebruikt worden.
- Je kunt zelf een nieuwe branch maken in Visual Studio Code of via de terminal.
- Je kunt branches samenvoegen (mergen) zonder gegevens te verliezen.
- Je leert samenwerken in duo's waarbij ieder aan een eigen branch werkt.

? Uitleg

In een softwareproject wil je vaak experimenteren zonder de hoofdversie te breken. Daarvoor gebruik je **branches** (vertakkingen). Elke branch is een eigen versie van de code waarop je onafhankelijk kunt werken.

De hoofdversie heet meestal `main` of `master`. Je kunt een nieuwe branch maken (bijv. `feature-login`) om iets nieuws te proberen. Als het goed werkt, voeg je die branch later weer samen met de hoofdversie (dat heet **mergen**).

Voorbeeld

```
# Nieuwe branch aanmaken en erop overschakelen
git checkout -b feature-login

# Werken op deze branch
git add .
```

```
git commit -m "Loginfunctie toegevoegd"

# Terug naar main
git checkout main

# Branch samenvoegen met main
git merge feature-login
```

In Visual Studio Code

1. Open je project in VSC (bijv. `vsc_git_demo` van vorige les).
2. Open de **Source Control**-tab en klik onderaan in de blauwe balk op **main**.
3. Kies **Create new branch** en geef hem een naam, bijvoorbeeld `feature-welkom`.
4. Maak nu in de nieuwe branch een kleine aanpassing in je code, bijvoorbeeld:

```
# app.py - feature-welkom branch
naam = input("Wat is je naam? ")
print(f"Welkom {naam}, fijn dat je er bent!")
```

5. Sla de wijziging op en commit met het bericht: `Nieuwe welkomstboodschap toegevoegd`.
6. Ga terug naar de branch **main** via de blauwe balk onderaan.
7. Klik op het tandwiel in de Source Control-tab → **Merge Branch** → kies `feature-welkom`.

Je wijzigingen worden samengevoegd in de hoofdversie. Gefeliciteerd, je hebt je eerste merge uitgevoerd!

?? Opdracht – Branch en Merge (Duo)

Situatie: jullie werken samen aan één project. Jij werkt aan de begroeting, je partner aan een nieuwe functie.

1. Kies één laptop waarop het project van de vorige les staat.
2. Maak samen een plan:
   - Student A werkt op branch `feature-begroeting`
   - Student B werkt op branch `feature-berekening`
3. Beide studenten voegen hun eigen stuk code toe, bijvoorbeeld:

```
# feature-begroeting
print("Hallo, welkom terug bij ons programma!")

# feature-berekening
getal = int(input("Voer een getal in: "))
print(f"Het kwadraat van {getal} is {getal ** 2}.")
```

4. Beide branches worden apart gecommit.
5. Vervolgens worden beide branches één voor één gemerged in `main`.
6. Test of het gecombineerde programma werkt.
7. Simuleer nu een fout:
   - Pas op beide branches `print("Welkom!")` tegelijk aan met een andere tekst.
   - Merge opnieuw → er ontstaat een **merge conflict**.
8. Los het merge conflict op in Visual Studio Code:
   - Kies via “Accept Both Changes” of “Accept Current Change”.
   - Test de uiteindelijke code.

? Reflectie

- Waarom is het werken met branches veiliger dan direct werken op main?
- Wat gebeurt er bij een merge conflict, en hoe kun je dat oplossen?
- Wat zijn goede afspraken als je samenwerkt in één Git-project?
- Hoe zorg je ervoor dat commitberichten begrijpelijk zijn voor je partner?

? Inleveren

- Screenshot van je `Source Control`-tab met ten minste twee branches en één merge.
- Screenshot van de opgeloste merge-conflictweergave in Visual Studio Code.
- Korte reflectie in `reflectie-branches-<jouwnaam>.txt` met antwoorden op bovenstaande vragen.
- Lever dit in via Canvas per duo (één student levert in namens beiden).

4 – GitHub en samenwerken op afstand

? Leerdoelen

- Je begrijpt wat GitHub is en wat het verschil is met Git.
- Je kunt een lokale Git-repository koppelen aan GitHub.
- Je kunt code pushen (uploaden) en pullen (downloaden) via GitHub.
- Je kunt in duo's samenwerken via branches en pull requests.

? Uitleg

Tot nu toe hebben we lokaal gewerkt met Git: al je versies staan op jouw computer. Met **GitHub** kun je jouw repository online opslaan zodat anderen eraan kunnen meewerken. GitHub is dus een *centrale opslagplaats* voor Git-projecten. Je kunt daar samen aan code werken, pull requests indienen en elkaars werk reviewen.

□□ Verschil tussen Git en GitHub

Git	GitHub
Versiebeheer op je eigen computer	Online platform om Git-projecten te delen
Werkt ook zonder internet	Vereist een account en internetverbinding
Gebruik via Terminal of VS Code	Gebruik via website of geïntegreerd in VS Code

? Repository koppelen aan GitHub

1. Maak een account aan op GitHub.com.
2. Klik op **New Repository** → geef je project een naam (bijv. `python-duo-project`).
3. Laat de opties "Public" en "Add a README" aanstaan.
4. Klik op **Create Repository**.
5. Kopieer de URL die GitHub toont, bijvoorbeeld:

   ```
   https://github.com/jouwnaam/python-duo-project.git
   ```

6. Open Visual Studio Code → open je projectmap.
7. Open de terminal in VSC en voer uit:

```
# Koppel je lokale map aan GitHub
git remote add origin https://github.com/jouwnaam/python-duo-project.git

# Upload de bestaande commits
git branch -M main
git push -u origin main
```

Je project staat nu online! Ga naar GitHub en vernieuw de pagina — je ziet je code, commits en branches verschijnen.

□□ Samenwerken

Je kunt nu andere studenten uitnodigen als **collaborator**:

1. Ga op GitHub naar **Settings → Collaborators**.
2. Klik op **Add people** en voeg het GitHub-account van je duo toe.
3. Je partner ontvangt een e-mail en kan daarna ook pushen en pullen.

□□ Werken in duo's (voorbereiding)

1. Student A maakt de repository aan en voegt student B toe als collaborator.
2. Student B klikt op "Code → HTTPS" en kiest **Clone** (kopieer de URL).
3. In Visual Studio Code → **Source Control → Clone Repository** en plak de URL.
4. Beide studenten hebben nu dezelfde code lokaal.

?? Opdracht – GitHub Push & Pull (Duo)

Situatie: jullie werken samen aan één Python-bestand dat een klein programma vormt. Jullie zullen elkaars wijzigingen om de beurt uploaden (push) en binnenhalen (pull).

1. Student A maakt het bestand `teamcode.py` met deze inhoud:

```
# teamcode.py - versie 1
print("Welkom bij het teamproject!")
```

2. Student A commit en pusht naar GitHub.
3. Student B doet `git pull` of gebruikt de knop **Pull** in VSC om de nieuwste versie binnen te halen.

4. Student B voegt code toe:

```
# teamcode.py - versie 2
naam = input("Wat is je naam? ")
print(f"Hallo {naam}, fijn dat we samenwerken!")
```

5. Student B commit en pusht zijn wijziging.
6. Student A doet opnieuw **Pull** om de update te ontvangen.
7. Beide studenten voegen nog één feature toe op hun eigen branch (zie les 3) en mergen die via GitHub.
8. Controleer in de GitHub-geschiedenis of beide namen zichtbaar zijn bij de commits.

? Reflectie

- Wat is het voordeel van werken met een gedeelde GitHub-repository?
- Wat gebeurt er als twee mensen tegelijk dezelfde regel code aanpassen?
- Wat is het verschil tussen `push` en `pull`?
- Waarom is het belangrijk om regelmatig te synchroniseren?

? Inleveren

- Link naar jullie GitHub-repository (zorg dat deze publiek of gedeeld is).
- Screenshot van:
  - de commitgeschiedenis met beide namen zichtbaar;
  - het Python-bestand in de browser met de laatste versie;
- Reflectie in `reflectie-github-<jouwnaam>.txt`.

? Verdieping (optioneel)

Voor studenten die verder willen:

- Leer werken met **Pull Requests** op GitHub. Maak een branch, push je wijziging, en open een Pull Request via de webinterface.
- Gebruik **Issues** om taken te verdelen in jullie duo-project.

- Bekijk onder *Insights → Network* hoe Git visualiseert hoe jullie branches zijn samengevoegd.

5 – Conflicten oplossen en samenwerken in grotere teams

? Leerdoelen

- Je begrijpt wat een **merge-conflict** is en waarom dat ontstaat.
- Je kunt een merge-conflict handmatig oplossen in Visual Studio Code.
- Je leert werken met **Pull Requests** op GitHub voor samenwerking in teams.
- Je kunt code van medestudenten reviewen en samenvoegen via GitHub.

? Uitleg

Wanneer meerdere mensen aan dezelfde code werken, kan het gebeuren dat twee personen dezelfde regel aanpassen. Git weet dan niet welke versie moet worden bewaard — dat noemen we een **merge-conflict**.

In deze les leer je hoe je zulke situaties oplost en hoe grotere teams hun werk organiseren met **Pull Requests**, zodat er meer controle is voordat nieuwe code wordt samengevoegd.

□□ Voorbeeld van een conflict

```
# main branch
print("Welkom bij ons teamproject!")


# feature branch
print("Welkom bij het nieuwe systeem!")
```

Als beide versies samengevoegd worden, weet Git niet welke te behouden. Je krijgt dan een conflict dat er zo uitziet:

```
<<<<<<< HEAD
print("Welkom bij ons teamproject!")
=======
print("Welkom bij het nieuwe systeem!")
```

```
>>>>>>> feature
```

In Visual Studio Code kun je dit eenvoudig oplossen:

- Klik op “**Accept Current Change**” om de versie van `main` te behouden.
- Klik op “**Accept Incoming Change**” om de versie van de branch te behouden.
- Klik op “**Accept Both Changes**” om beide regels te behouden.

Daarna commit je de oplossing en is het conflict verdwenen.

? Pull Requests

In grotere teams wil je niet dat iedereen zomaar rechtstreeks naar `main` pusht. In plaats daarvan werk je met **Pull Requests (PR’s)**:

1. Je maakt een nieuwe branch (bijv. `feature-login`).
2. Je commit en pusht je wijzigingen naar GitHub.
3. Op GitHub klik je op **Compare & Pull Request**.
4. Een teamgenoot bekijkt jouw code, stelt eventueel vragen of keurt de PR goed.
5. Na goedkeuring wordt de code samengevoegd met `main`.

Pull Request voorbeeld

?? Opdracht – Samenwerken met Pull Requests (team van 3)

Situatie: jullie werken met z’n drieën aan één Python-project. Iedere student werkt op een eigen branch en dient daarna een Pull Request in.

1. Student A maakt een nieuwe GitHub-repository aan: `teamproject`.
2. Voeg student B en C toe als *collaborators*.
3. Ieder clonet de repository op zijn eigen computer.
4. Ieder maakt een eigen branch:
   - Student A → `feature-begroeting`
   - Student B → `feature-berekening`
   - Student C → `feature-menu`

5. Iedere student maakt zijn eigen functie:

```
# feature-begroeting
def begroet():
    naam = input("Wat is je naam? ")
    print(f"Welkom {naam}!")

# feature-berekening
def berekening():
    getal = int(input("Voer een getal in: "))
    print(f"Het kwadraat van {getal} is {getal ** 2}.")

# feature-menu
def menu():
    print("1. Begroeting")
    print("2. Berekening")
```

6. Ieder pusht zijn branch naar GitHub.
7. Iedere student opent een **Pull Request** via de GitHub-interface.
8. Student A (projectbeheerder) controleert en accepteert de PR's één voor één.
9. Controleer daarna de **commit history** en **Network graph** op GitHub: alle branches zijn samengevoegd.

□□ Extra oefening: Merge-conflict simulatie

1. Laat student A en B tegelijk dezelfde regel in `menu()` wijzigen.
2. Push beide versies.
3. Bij de tweede merge verschijnt een conflict.
4. Los dit conflict op via Visual Studio Code of via de GitHub webinterface.
5. Commit de oplossing en controleer dat de code correct werkt.

? Reflectie

- Wat is het voordeel van werken met Pull Requests?
- Wanneer ontstaat een merge-conflict precies?

- Hoe los je een conflict op zonder werk van anderen te verliezen?
- Waarom is duidelijke communicatie binnen een team belangrijk bij versiebeheer?

? Inleveren

- Link naar jullie GitHub-repository (teamproject).
- Screenshot van minimaal drie Pull Requests (1 per student).
- Screenshot van een opgelost merge-conflict in Visual Studio Code of GitHub.
- Korte reflectie in `reflectie-teamsamenwerking-<jouwnaam>.txt`.

? Verdieping (optioneel)

Voor studenten die verder willen:

- Gebruik **GitHub Issues** om taken te verdelen (zoals “maak loginfunctie”).
- Voeg labels, beschrijvingen en reviewers toe aan je Pull Requests.
- Leer hoe je **protected branches** instelt zodat alleen goedgekeurde code in `main` terechtkomt.
- Onderzoek het gebruik van **GitHub Actions** om automatisch tests uit te voeren bij nieuwe commits.

6 – Versiebeheer en herstel

? Leerdoelen

- Je begrijpt wat **versiebeheer** betekent en waarom het belangrijk is bij softwareontwikkeling.
- Je kunt eerdere versies van je project bekijken, herstellen en vergelijken.
- Je leert hoe je veilig kunt experimenteren zonder werk te verliezen.
- Je kunt werken met de belangrijkste Git-commando’s voor herstel (`log`, `checkout`, `revert`, `reset`).

? Uitleg

Git is niet alleen handig om samen te werken — het is ook een **tijdmachine** voor je code. Elke keer dat je commit, maak je een momentopname van je project. Als er later iets misgaat, kun je eenvoudig terug naar een eerdere staat.

Deze les leert je hoe je commits kunt bekijken, oudere versies kunt herstellen en zelfs fouten kunt terugdraaien.

□□ Voorbeeld: commitgeschiedenis bekijken

Gebruik het volgende commando om te zien wat je eerder hebt opgeslagen:

```
git log --oneline
```

Voorbeeldoutput:

```
a4c9e21  Voeg begroetingsfunctie toe
a3e11bf  Maak hoofdmenu aan
b21d9a7  Eerste commit - projectstructuur
```

Je ziet dat elke commit een unieke code (de “hash”) heeft, zoals `a4c9e21`. Die code kun je gebruiken om terug te gaan naar dat specifieke moment in de tijd.

? Teruggaan naar een vorige versie

□ `git checkout` (tijdelijk terugkijken)

Gebruik dit commando om tijdelijk een oude versie van je code te bekijken:

```
git checkout a3e11bf
```

Je project staat nu in de toestand van die commit. Wil je weer terug naar de nieuwste versie?

```
git checkout main
```

□ `git revert` (een wijziging ongedaan maken)

Als een fout in een eerdere commit zit, kun je die ongedaan maken met:

```
git revert a4c9e21
```

Git maakt dan automatisch een *nieuwe* commit die de fout ongedaan maakt, zonder dat je geschiedenis wordt gewist.

⏪ `git reset` (permanent terugdraaien)

Met `git reset` kun je echt terug in de tijd — maar pas op, want dit verwijdert commits:

```
git reset --hard a3e11bf
```

Gebruik dit alleen als je zeker weet dat je die commits niet meer nodig hebt.

? Experimenteren zonder risico

Wil je iets proberen zonder je hoofdproject te breken? Maak dan een nieuwe branch:

```
git checkout -b testversie
```

Zo kun je experimenteren in een veilige kopie van je project. Als het goed werkt, merge je het later terug in `main`. Als het niet werkt, verwijder je gewoon de branch:

```
git branch -d testversie
```

?? Opdracht – De Git Tijdmachine

Doel: leer hoe je commits kunt terugzien, herstellen en veilig experimenteren met branches.

1. Maak een nieuw project aan in Visual Studio Code en initialiseer Git:
   ```
   git init
   ```
2. Maak een bestand `team.py` met deze inhoud:
   ```
   print("Versie 1 – Hallo team!")
   ```
3. Voer de volgende stappen uit:
   1. Commit de eerste versie: `git add .` en `git commit -m "Versie 1"`
   2. Verander de tekst naar: `print("Versie 2 – Nieuwe begroeting!")`
   3. Commit opnieuw met `git commit -am "Versie 2"`
   4. Bekijk de geschiedenis met `git log --oneline`
4. Gebruik `git checkout` om tijdelijk terug te gaan naar versie 1. Controleer in VS Code wat er verandert.
5. Gebruik `git revert` om de laatste wijziging ongedaan te maken.

6. Maak daarna een nieuwe branch `testversie` en voeg daar extra code aan toe:

```
print("Testmodus actief!")
```

7. Controleer met `git branch` welke branches er zijn, en verwijder daarna de testversie met `git branch -d testversie`.

? Reflectie

- Wat is het verschil tussen `checkout`, `revert` en `reset`?
- Waarom is het veiliger om met `revert` te werken dan met `reset`?
- Wat is het voordeel van werken met branches bij experimenteren?
- Heb je ooit code verloren? Hoe zou Git je daartegen kunnen beschermen?

? Inleveren

- Lever één screenshot in van de `git log` output (met minimaal 2 commits).
- Lever één screenshot in waarin je `git revert` gebruikt en het effect zichtbaar is.
- Lever je map met Git-project als ZIP-bestand in met de naam `versiebeheer-<jouwnaam>.zip`.
- Voeg een kort reflectiebestand toe: `reflectie-versiebeheer-<jouwnaam>.txt`.

? Verdieping (optioneel)

- Gebruik **git diff** om te vergelijken wat er tussen twee commits is veranderd.
- Probeer **git tag** om belangrijke versies te markeren, bijvoorbeeld: `git tag v1.0`.
- Onderzoek het gebruik van **git restore** om specifieke bestanden te herstellen zonder de hele commit terug te draaien.
- Bekijk in GitHub Desktop of VS Code de visuele commitgeschiedenis via de "Timeline"-weergave.

Les 7 – Praktijkopdracht: Samenwerken met Git en GitHub in een teamproject

? Leerdoelen

- Je kunt als team samenwerken aan één Python-project met behulp van Git en GitHub.
- Je kunt branches gebruiken om parallel te werken zonder elkaars werk te overschrijven.
- Je begrijpt hoe je Pull Requests gebruikt om wijzigingen samen te voegen.
- Je kunt conflicten oplossen, commits terugdraaien en de commitgeschiedenis analyseren.
- Je kunt reflecteren op de voordelen van versiebeheer bij teamwork.

? Inleiding

In deze praktijkles brengen we alles samen wat je tot nu toe hebt geleerd over Git en GitHub. Je werkt in een klein team (2 of 3 personen) aan één gezamenlijk project. Het doel is niet alleen een goed werkend programma, maar ook een **duidelijke samenwerking via versiebeheer**.

Je zult merken dat goed versiebeheer het verschil maakt tussen chaos en controle. Jullie gaan werken alsof jullie een echt ontwikkelteam zijn dat samen code schrijft, reviewt en samenvoegt.

? Het project

Het team bouwt een klein Python-project genaamd **“TeamPlanner”** — een programma dat afspraken en taken van teamleden toont. Elke student krijgt een onderdeel:

- **Student A:** maakt de functie `voeg_taak_toe()` die taken toevoegt.
- **Student B:** maakt de functie `toon_todo_lijst()` die taken weergeeft.
- **Student C:** maakt de functie `bewaar_data()` die taken opslaat in een JSON-bestand.

Voorbeeld van het eindresultaat:

```
# teamplanner.py

import json

taken = []

def voeg_taak_toe():
    taak = input("Voer een taak in: ")
    taken.append(taak)
```

```
def toon_todo_lijst():
    print("\nJe takenlijst:")
    for t in taken:
        print("-", t)

def bewaar_data():
    with open("taken.json", "w") as f:
        json.dump(taken, f)
    print("Taken opgeslagen in taken.json")

# Hoofdmenu
while True:
    print("\n1. Taak toevoegen\n2. Lijst tonen\n3. Opslaan\n4. Stoppen")
    keuze = input("Kies een optie: ")

    if keuze == "1":
        voeg_taak_toe()
    elif keuze == "2":
        toon_todo_lijst()
    elif keuze == "3":
        bewaar_data()
    elif keuze == "4":
        break
    else:
        print("Ongeldige keuze")
```

?? Opdracht – Teamplanner bouwen

1. **□□ Repository opzetten**
   Eén student maakt een nieuwe GitHub-repository aan met de naam `teamplanner` en voegt de anderen toe als *collaborators*.

2. **□□ Clone het project**
   Ieder teamlid klonet de repository in Visual Studio Code:
   ```
   git clone https://github.com//teamplanner.git
   ```

3. **□□ Branches maken**
   Ieder maakt een eigen branch:

- `feature-taken-toevoegen`
- `feature-lijst-tonen`
- `feature-data-opslaan`

4. **Code schrijven**
   Iedereen schrijft zijn deel van de code in zijn eigen branch en commit regelmatig:

   ```
   git add .
   git commit -m "Voeg functie voeg_taak_toe toe"
   ```

5. **Pushen naar GitHub**
   Ieder student pusht zijn branch naar GitHub:

   ```
   git push -u origin feature-taken-toevoegen
   ```

6. **Pull Requests aanmaken**
   Op GitHub maakt ieder teamlid een Pull Request. Een ander teamlid controleert en keurt de PR goed (*reviewen*).

7. ⚔ **Conflicten oplossen**
   Als er conflicten ontstaan (bv. iedereen bewerkt `teamplanner.py`), los ze samen op via Visual Studio Code. Gebruik “Accept Both Changes” als jullie beide code willen behouden.

8. **Versies beheren**
   Na het mergen van alle functies:
   - Gebruik `git log --oneline` om alle commits te bekijken.
   - Gebruik `git revert` om een foutieve commit ongedaan te maken.
   - Gebruik `git checkout -b testversie` om een nieuwe testversie te maken en iets te proberen.

9. **Project afronden**
   Voeg samen nog een afsluitende commit toe met een bericht als:

   ```
   git commit -am "Eindversie TeamPlanner met alle functies"
   ```

? Samenwerken en communiceren

Verdeel de taken duidelijk en gebruik GitHub-berichten of een gezamenlijke chat om af te stemmen wie aan welk onderdeel werkt.

Tip: spreek af dat niemand direct pusht naar `main`, alleen via Pull Requests.

? Reflectie

- Wat ging goed in jullie samenwerking met GitHub?
- Waar liepen jullie tegenaan (conflicten, miscommunicatie, verkeerde merges)?
- Hoe hielp versiebeheer bij het terugvinden van fouten?
- Wat zou je de volgende keer anders doen in je teamworkflow?

? Inleveren

- Link naar jullie GitHub-repository (`teamplanner`).
- Screenshot van:
  - De commitgeschiedenis (`git log` of GitHub History).
  - Een Pull Request (met review).
  - Een opgelost merge-conflict.
- Reflectieverslag in `reflectie-teamplanner-<jouwnaam>.pdf` (max. 1 A4).

? Verdieping (optioneel)

- Gebruik **GitHub Issues** om taken te verdelen en voortgang te volgen.
- Gebruik **GitHub Projects** (Kanban-bord) om jullie workflow visueel te beheren.
- Maak gebruik van **tags** om releases te markeren, bijvoorbeeld `v1.0`.
- Gebruik **git blame** om te zien wie welke regel heeft geschreven.
- Probeer een automatische check toe te voegen via **GitHub Actions** om code automatisch te testen bij elke Pull Request.

8 – Codekwaliteit en Best Practices

? Leerdoelen

- Je begrijpt waarom duidelijke commit-berichten belangrijk zijn.
- Je kunt goede branch-namen en commit-boodschappen schrijven.

- Je kunt een `.gitignore`-bestand gebruiken om onnodige bestanden uit je repository te houden.
- Je kunt basisregels toepassen om code leesbaar, veilig en onderhoudbaar te houden.
- Je kunt samenwerken volgens professionele ontwikkelstandaarden.

? Inleiding

In de vorige les heb je samengewerkt via Git en GitHub. Maar samenwerken is pas écht effectief als je code **duidelijk, netjes en consistent** is. Goede codekwaliteit voorkomt verwarring, fouten en discussies binnen teams. Vandaag leer je hoe professionele ontwikkelaars werken met Git: met duidelijke namen, overzichtelijke commits, en schone repositories.

? 1. Structuur en namen

Gebruik bij elke project een duidelijke mappenstructuur en logische namen:

```
project/
│
├── src/           # broncode
├── data/          # JSON, CSV, of testdata
├── tests/         # testbestanden
├── .gitignore     # bestanden die niet in Git moeten
├── README.md      # uitleg over het project
└── main.py        # startbestand
```

Gebruik leesbare namen voor branches:

- `feature-login-systeem`
- `fix-bug-dataverwerking`
- `update-readme`

? 2. Commit-berichten

Een commit-bericht is een korte samenvatting van *wat* en *waarom* je iets hebt aangepast. Schrijf ze kort, duidelijk en in de tegenwoordige tijd.

 Goed voorbeeld:

```
git commit -m "Voeg validatie toe aan loginformulier"
```

Slecht voorbeeld:

```
git commit -m "fix"
git commit -m "update 3"
git commit -m "werkt nu"
```

Gebruik eventueel een langere commitbeschrijving met `git commit` zonder `-m`:

```
Voeg foutafhandeling toe bij het laden van JSON-bestand

- voorkomt crash bij ontbrekende data.json
- toont nu een foutmelding aan de gebruiker
```

? 3. .gitignore

Met een `.gitignore`-bestand geef je aan welke bestanden of mappen Git moet overslaan. Zo voorkom je dat tijdelijke of persoonlijke bestanden worden meegestuurd.

```
# .gitignore
__pycache__/
.vscode/
*.log
*.env
*.sqlite
```

Tip: gebruik github.com/github/gitignore voor kant-en-klare voorbeelden.

?? 4. Codekwaliteit

Netjes programmeren betekent:

- Duidelijke functienamen (`bereken_gemiddelde()` in plaats van `bg()`).
- Consistente inspringing en witregels.
- Korte functies – elke functie doet één ding goed.
- Commentaar alleen als iets niet vanzelfsprekend is.
- Gebruik type hints als dat helpt:

```
def bereken_gemiddelde(getallen: list[float]) -> float:
    return sum(getallen) / len(getallen)
```

Bij teamwork kun je ook een `README.md` gebruiken met uitleg:

```
# Teamplanner
Kleine applicatie om taken te beheren.
- Voeg taken toe
- Bekijk de takenlijst
- Sla taken op in JSON
```

? 5. Versiebeheer en teruggaan in tijd

Met Git kun je altijd terug naar een vorige versie van je project. Dat is de kracht van versiebeheer.

Bekijk je commitgeschiedenis:

```
git log --oneline
```

Terug naar een vorige commit (tijdelijk):

```
git checkout 3a7f4c2
```

Terugdraaien van een foutieve commit (blijvend):

```
git revert 3a7f4c2
```

Een nieuw test-branch aanmaken vanuit oude versie:

```
git checkout -b bugfix-test 3a7f4c2
```

Zo kun je veilig experimenteren, fouten herstellen en versies vergelijken zonder werk kwijt te raken.

?? Opdracht – Codekwaliteit in jouw project

1. Kies je `TeamPlanner`-repository of een ander project van jou.
2. Maak een `.gitignore`-bestand aan en voeg daarin ten minste 4 regels toe.
3. Schrijf 3 commits met duidelijke, professionele commit-berichten.

4. Maak een fout en herstel die met `git revert`.
5. Maak een nieuwe branch `cleanup-code` waarin je de code leesbaarder maakt (namen, witregels, comments).
6. Voeg een `README.md` toe met een korte beschrijving van je project.

? Samenwerken

Werk in duo's: ieder voert een code-review uit bij de ander. Controleer op leesbaarheid, netheid en begrijpelijkheid van commits.

Gebruik eventueel GitHub-comments of een kort reviewformulier om feedback te geven.

? Reflectie

- Wat heb je geleerd over het belang van duidelijke commit-berichten?
- Wat vind jij het belangrijkste aspect van goede codekwaliteit?
- Hoe helpt Git om samen te werken zonder code te verliezen?

? Inleveren

- Link naar je GitHub-repository.
- Screenshot van je `.gitignore`.
- Screenshot van `git log --oneline` met minstens 3 commits.
- Reflectieverslag (max. 1 A4) over codekwaliteit en versiebeheer.

? Verdieping (optioneel)

- Gebruik **pre-commit hooks** om automatisch codechecks uit te voeren (met tools zoals `black` of `flake8`).
- Probeer **GitHub Actions** om tests automatisch uit te voeren bij elke commit.
- Onderzoek wat **Semantic Versioning** is en hoe het wordt toegepast in open-source projecten.
- Gebruik **git tag** om versies te markeren, zoals `v1.0`.

9 – Branching en Merge-strategieën

? Leerdoelen

- Je begrijpt wat branches zijn en waarom ze belangrijk zijn bij samenwerking.
- Je kunt een nieuwe branch aanmaken, wisselen en samenvoegen (mergen).
- Je kunt een merge-conflict herkennen en oplossen.
- Je leert de basis van merge-strategieën (zoals main/develop-feature aanpak).
- Je werkt samen met een teamlid via GitHub op een gecontroleerde manier.

? Inleiding

Tot nu toe heb je geleerd hoe je commits maakt en versies beheert. Maar in echte projecten werken vaak meerdere programmeurs tegelijk aan verschillende onderdelen. Hoe voorkom je dat iedereen in dezelfde code schrijft en elkaars werk overschrijft? Dat doe je met **branches** – aparte "takken" van je project.

Een branch is als een zijspoor: je kunt experimenteren, nieuwe functies bouwen of bugs oplossen zonder het hoofdproject (de `main`-branch) te breken.

? 1. Wat is een branch?

Een **branch** is een aparte ontwikkellijn binnen een Git-repository. Elke branch kan eigen commits bevatten, en uiteindelijk samengevoegd worden met de hoofdbranch.

```
main
 ├── commit A
 ├── commit B
 │
 └── feature-login
      ├── commit C
      └── commit D
```

Branches helpen je om veilig te werken aan nieuwe features zonder dat de hoofdversie kapotgaat.

? 2. Branch aanmaken en wisselen

Bekijk bestaande branches:

```
git branch
```

Maak een nieuwe branch:

```
git branch feature-login
```

Ga naar die branch:

```
git checkout feature-login
```

Of in één stap:

```
git checkout -b feature-login
```

Wanneer je nu commits maakt, komen die alleen in `feature-login` terecht, niet in `main`.

? 3. Samenvoegen van branches (merge)

Als je klaar bent met werken op een branch, wil je de aanpassingen terugbrengen naar `main`.

Stap 1 – Ga naar main:

```
git checkout main
```

Stap 2 – Merge de feature-branch:

```
git merge feature-login
```

Git probeert de aanpassingen automatisch samen te voegen. Soms gaat dat goed, maar soms niet...

? 4. Merge-conflicten oplossen

Een **merge-conflict** ontstaat als twee mensen dezelfde regels code hebben aangepast in verschillende branches.

Bijvoorbeeld:

```
<<<<<<< HEAD
print("Welkom bij de webshop!")
=======
print("Welkom bij de loginpagina!")
```

```
>>>>>>> feature-login
```

De markeringen `<<<<<<<` en `>>>>>>>` laten zien waar Git niet weet welke versie de juiste is.

Oplossen:

1. Kies de juiste tekst (of combineer ze).
2. Verwijder de conflict-markeringen.
3. Voeg het bestand opnieuw toe: `git add bestandsnaam`
4. Maak een nieuwe commit: `git commit -m "Los merge-conflict op"`

?? 5. Branch-strategieën

In professionele projecten gebruikt men vaak vaste afspraken over branches:

- **main** – de stabiele versie van het project (productie).
- **develop** – de werkversie waarin nieuwe features worden samengebracht.
- **feature/naam** – individuele ontwikkeltakken voor nieuwe functies.
- **hotfix/naam** – snelle bugfixes op de main-branch.

Zo houd je overzicht, en weet iedereen waar welke code thuishoort.

```
main ──► develop ──► feature-login
              │
              └────► feature-dashboard
```

?? Opdracht – Samenwerken met branches

Werk in duo's. Je gaat samen een klein Python-project uitbreiden met branches en merges.

1. Student A maakt een nieuwe repository op GitHub en deelt die met Student B (collaborator).
2. Student A maakt de branch `feature-login`.
3. Student B maakt de branch `feature-dashboard`.
4. Beide voegen in hun branch een apart Python-bestand toe (bijv. `login.py` en `dashboard.py`).

5. Push de branches naar GitHub met `git push origin feature-login` en `git push origin feature-dashboard`.
6. Maak via GitHub een **Pull Request** om beide branches samen te voegen.
7. Als er een conflict is: los dit op, commit en merge opnieuw.

? Reflectie

- Wat zijn de voordelen van werken met branches?
- Wat ging er mis bij het samenvoegen en hoe heb je dat opgelost?
- Waarom is het belangrijk om kleine, logische commits te maken bij samenwerking?

? Inleveren

- Link naar je GitHub-repository.
- Screenshot van de twee branches in GitHub.
- Screenshot van het oplossen van een merge-conflict.
- Kort reflectieverslag (`reflectie-branches-<jouwnaam>.txt`).

? Verdieping (optioneel)

- Onderzoek de verschillen tussen **merge** en **rebase**.
- Experimenteer met `git stash` om tijdelijk werk op te slaan zonder te committen.
- Gebruik `git log --graph --oneline --all` om de branch-structuur visueel te zien.
- Lees over de **Git Flow**-methode, een populaire workflow met vaste regels voor branches.

10 – GitHub Collaboration & Pull Requests

? Leerdoelen

- Je begrijpt hoe samenwerken via GitHub werkt met forks, branches en pull requests (PR’s).
- Je kunt bijdragen aan andermans project via een fork en een pull request.
- Je kunt code van anderen beoordelen (code review) en verbeteringen voorstellen.
- Je weet hoe je als projectbeheerder bijdragen van anderen kunt samenvoegen.
- Je kunt professioneel samenwerken in een gedeelde GitHub-repository.

? Inleiding

Je weet nu hoe je met Git kunt werken, branches maakt en samenvoegt. In deze les leer je hoe ontwikkelaars wereldwijd samenwerken aan hetzelfde project — via **GitHub** en het mechanisme van *Pull Requests* (PR’s). Dat is hoe open-source software zoals Python, React en Linux worden gebouwd.

Met een pull request zeg je eigenlijk: **“Ik heb iets verbeterd — willen jullie dit toevoegen aan het hoofdproject?”**

? 1. Forken van een project

Wanneer je niet direct schrijfrechten hebt op een repository (bijv. een open-source project), maak je een **fork** — een eigen kopie van de repository op jouw GitHub-account.

1. Ga naar het GitHub-project van je klasgenoot of docent.
2. Klik op **Fork** (rechtsboven).
3. GitHub maakt een kopie van het project in jouw eigen account.
4. Kloon je fork naar je lokale computer:
   ```
   git clone https://github.com/jouwnaam/projectnaam.git
   ```

Je kunt nu in je eigen kopie werken zonder de originele repository te veranderen.

? 2. Wijzigingen maken in een fork

Je maakt altijd een aparte branch voor je aanpassingen:

```
git checkout -b verbeterde-documentatie
```

Voeg een nieuw bestand toe of pas iets aan:

```
echo "Dit is een verbetering" >> README.md
git add README.md
git commit -m "Verbeter documentatie in README"
```

Push je branch naar GitHub:

```
git push origin verbeterde-documentatie
```

? 3. Een Pull Request maken

Een **Pull Request** (PR) is een verzoek aan de eigenaar van het originele project om jouw code te “mengen” in hun repository.

1. Ga op GitHub naar jouw repository (de fork).
2. Je ziet een melding: *“Compare & Pull Request”* — klik daarop.
3. Voeg een titel en beschrijving toe waarin je uitlegt wat je hebt gewijzigd en waarom.
4. Klik op **Create Pull Request**.

De eigenaar van het originele project kan je wijziging nu bekijken, opmerkingen maken of deze samenvoegen.

? 4. Code Review

In professionele teams bekijkt altijd iemand anders jouw code voordat deze wordt samengevoegd. Dat heet een **code review**.

Tijdens een review controleer je:

- Is de code duidelijk en goed leesbaar?
- Worden er geen fouten geïntroduceerd?
- Zijn de commit-berichten logisch?
- Houdt de code zich aan de afspraken (naming, structuur)?

Voorbeeld van feedback op GitHub:

```
□□"Goede toevoeging! Overweeg om de functienaam duidelijker te maken."
□□"Hier kun je beter een f-string gebruiken in plaats van concatenatie."
```

De eigenaar kan na de review kiezen om de wijziging te accepteren (`Merge Pull Request`), of eerst nog om verbeteringen vragen.

?? 5. Een Pull Request samenvoegen (merge)

Als jij de eigenaar bent van het project, kun je binnenkomen PR's beheren:

1. Bekijk de PR op GitHub.
2. Controleer de wijzigingen via het tabblad *Files changed*.
3. Laat een review achter (approve, comment of request changes).
4. Klik op **Merge Pull Request** → **Confirm**.

Na het mergen is de code van de andere ontwikkelaar onderdeel van jouw hoofdbranch. Dat is samenwerking op professioneel niveau.

? 6. Conflicten bij Pull Requests

Soms wijzigt iemand dezelfde regel code als jij. GitHub toont dan een melding: **"This branch has conflicts that must be resolved."**

Je kunt het conflict online oplossen:

1. Klik op **Resolve conflicts**.
2. Kies de juiste regels of combineer beide versies.
3. Klik op **Mark as resolved** → **Commit merge**.

?? Opdracht – Samenwerken via Pull Requests

Werk in duo's. Jullie gaan elkaars code uitbreiden via forks en pull requests.

1. Student A maakt een GitHub-repository (`duo-project`) en zet er een simpel Python-programma in (bijv. `main.py` met een print-statement).
2. Student B **forkt** de repository van Student A.
3. Student B maakt een branch `feature-uitbreiding` en voegt iets toe (bijv. een functie of nieuw bestand).
4. Student B maakt een Pull Request naar de originele repository van Student A.
5. Student A beoordeelt de code, geeft feedback en merge de PR.

6. Beide studenten bespreken wat er goed ging en wat beter kan bij het reviewen.

? Samenwerken

Gebruik duidelijke commit-boodschappen en beschrijf je wijzigingen in de PR. Gebruik GitHub-comments om feedback te geven en om verbeteringen te bespreken.

? Reflectie

- Wat vond je van het werken met forks en pull requests?
- Wat heb je geleerd over het reviewen van elkaars code?
- Waarom zijn duidelijke PR-beschrijvingen belangrijk?
- Wat zou je volgende keer anders doen in de samenwerking?

? Inleveren

- Link naar het GitHub-project van Student A (met samengevoegde PR).
- Link naar het GitHub-project van Student B (fork en eigen branch).
- Screenshots van de Pull Request en de review-reacties.
- Kort reflectieverslag (`reflectie-pullrequest-<jouwnaam>.txt`).

? Verdieping (optioneel)

- Onderzoek hoe grote open-source projecten PR’s beheren (bijv. op GitHub bij Django of Flask).
- Gebruik **Draft Pull Requests** om feedback te vragen vóór je klaar bent.
- Experimenteer met **Code Owners** en **Reviewers** in GitHub-instellingen.
- Probeer een workflow met **GitHub Projects** of **Issues** voor taakverdeling.

Revision #4
Created 2025-10-19 15:41:33 UTC by Max
Updated 2025-10-19 15:49:34 UTC by Max