

DSCB 230 - Aufgabenblatt 11

Die Aufgabenblätter werden in dieser Veranstaltung in Jupyter Notebooks veröffentlicht und bearbeitet. Diese finden Sie in der Github Organisation für Data Science 2 unter dem Repository *dscb230-tutorial* (<https://github.com/hka-mmvmv/dscb230-tutorial>). Die Musterlösung wird ebenfalls in Form eines Jupyter Notebook in Github hochgeladen.

Aufgabenteil 1

Heute lernt ihr Git. Erarbeitet folgende Arbeitsschritte in einem Tool (Terminal, Git-Desktop-UI Anwendung, ...) eurer Wahl. Die Musterlösung dieser Aufgabe beinhaltet die Lösung mittels dem Terminal.

1. Erstellt euch ein Github Profil (mit der HKA habt ihr Github Pro)
2. Erstellt euch ein Github Repository mit folgenden Eigenschaften:
 - Name: `HelloWorld`
 - Description: `Das Effizienteste Hello-World Programm dieser Erde`
 - Privat auswählen
 - README auswählen
 - .gitignore auswählen
 - Wähle `Python3` aus
 - Lizenz auswählen
 - Wähle `MIT` aus
3. Klont euer neues Repository auf eure Lokale Maschine
4. Navigiert mit einem Editor eurer Wahl in den neuen Ordner
5. Erstellt eine Python Datei
 - Name: `hello-world.py`
 - Inhalt: `print(„Hello, World!“)`
6. Pusht euren Inhalt in den Main Branch
7. Erstellt einen neuen Branch und wechselt da hinein
 - Name: `Optimized_HelloWorld`
8. Verändert nun die Datei wie folgt
 - Inhalt: `print(„Hello, World!\n“ * 100)`
9. Pusht nun auch diesen Inhalt in den entsprechenden Branch
10. Vollzieht nun einen Merge durch.
11. Verändert nun die `.gitignore` Datei sodass alle Python3 Dateien nicht mehr erkannt werden, denkt dran, alle bisherigen Dateien zu löschen.

Aufgabenteil 2

MERGE CONFLICT

Wenn ihr einen Branch in einen anderen merged, kann es unter Umständen zu einem sog. „merge-conflict“ kommen. Dies tritt auf, wenn eine (oder mehrere) Dateien gleichzeitig in verschiedenen Branches verändert wurde und danach zusammengeführt werden soll.

Um einen merge-conflict zu provozieren, habt ihr folgende Aufgaben:

1. Erstelle eine Datei („conflict_file.py“) in dem master Branch und schreibe etwas hinein.
2. Führe die notwendigen Schritte aus, um die Datei anschließend zu „commiten“
3. Erstelle einen neuen Branch („Feature-1“) und wechsle da hinein
4. Verändere „conflict_file.py“ (in „Feature-1“) und „commite“
5. Wechsle nun in den master Branch, verändere „conflict_file.py“ dort auch (in „master“).

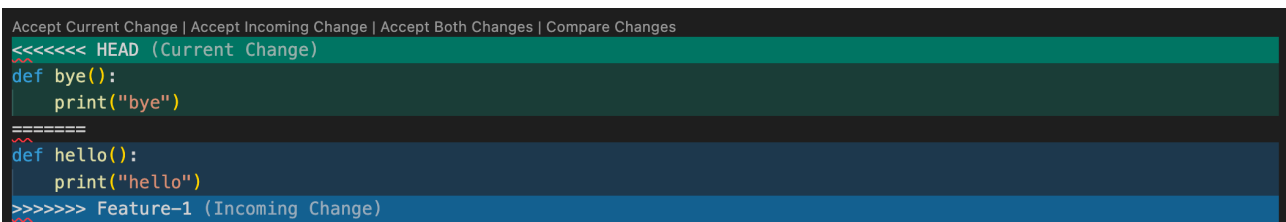
-> Die Veränderung sollte sich von der aus Schritt 4 unterscheiden

6. Versuche jetzt „Feature-1“ in master zu mergen

-> Jetzt sollte folgende Nachricht erscheinen

```
automatischer Merge von conflict_file.py
KONFLIKT (Inhalt): Merge-Konflikt in conflict_file.py
Automatischer Merge fehlgeschlagen; beheben Sie die
Konflikte und committen Sie dann das Ergebnis.
```

„conflict_file.py“ könnte z.B. so aussehen



```
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<<< HEAD (Current Change)
def bye():
    print("bye")
=====
def hello():
    print("hello")
>>>>>>> Feature-1 (Incoming Change)
```

Wenn man sich den „Git-status“ ansieht, kann man sehen, dass der merge noch nicht ausgeführt wurde und nun Handlungen gefordert werden.

7. Passt „conflict_file.py“ nun so an, wie die Datei nach dem merge aussehen soll (arbeitet ihr im Team, ist jetzt der Moment, um sich mit der Person, die die Datei auch bearbeitet hat, zu beraten und sich auf eine Lösung zu einigen).
8. Stage und commite jetzt den merge

-> Jetzt sollte Folgendes erscheinen:

```
Merge branch 'Feature-1'

# Conflicts:
#   conflict_file.py
#
# Es sieht so aus, als committed Sie einen Merge.
# Falls das nicht korrekt ist, löschen Sie bitte die Datei
#   .git/MERGE_HEAD
# und versuchen Sie es erneut.

# Bitte geben Sie eine Commit-Beschreibung für Ihre Änderungen ein. Zeilen,
# die mit '#' beginnen, werden ignoriert, und eine leere Beschreibung
# bricht den Commit ab.
#
# Auf Branch master
# Alle Konflikte sind behoben, aber Sie sind immer noch beim Merge.
#
# Zum Commit vorgemerkte Änderungen:
#   geändert:      conflict_file.py
#
```

9. Schreibe nun einfach „:wq“ (write, quitt) und der Merge sollte vollzogen werden.

Push verweigert! Was tun?

(Preissel, R. and Stachmann, B. (2019) Git: dezentrale Versionsverwaltung im Team Grundlagen und Workflows.)

Ein Push wurde verweigert, weil auf dem gleichen Branch im anderen Repository ebenfalls Änderungen hinzugekommen sind. Der Konflikt muss lokal gelöst werden, bevor der Push durchgelassen wird.

1. Konflikt feststellen

Der push-Befehl meldet den Konflikt durch die folgende, etwas umständliche Meldung:

```
> git push

To/tmp/git-buch-klon.git
![rejected]                    alpha -> alpha (fetch first)
error: failed to push some refs to '/Users/stachi/Buch/'
hint: Updates were rejected because the remote contains work that
hint: you do not have locally. This is usually caused by another
hint: repository pushing to the same ref. You may want to first
hint: integrate the remote changes (e.g. "git pull ...") before
hint: pushing again. See the 'Note about fast-forwards' in
hint: 'git push --help' for details.
```

2. Pull durchführen

```
> git pull
```

3. Gegebenenfalls Merge-Konflikte bereinigen

```
> git mergetool
> git commit —all
```

4. Noch mal: Push

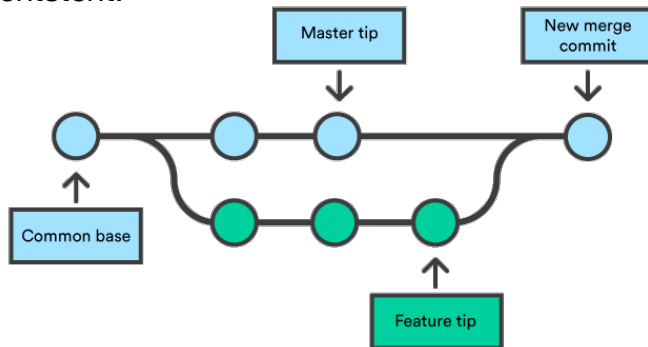
```
> git push
```

Aufgabenteil 3

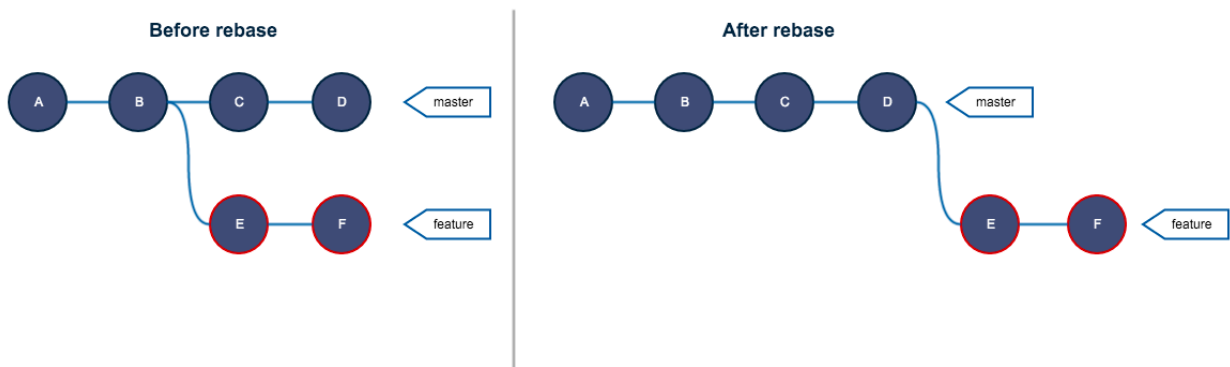
1. Unterschied zwischen git merge und git rebase:

Wird bei einem Projekt eine feature Branch basierend auf einer bestimmten Version der master Branch erstellt, so können master- und feature Branch, welche sich beide in der Version entwickelt haben, mithilfe von git merge oder git rebase zusammengeführt werden.

Mit „git merge --squash feature“ aus der master Branch heraus wird master- und feature Branch zusammengeführt wodurch eine neue Version der master Branch entsteht.



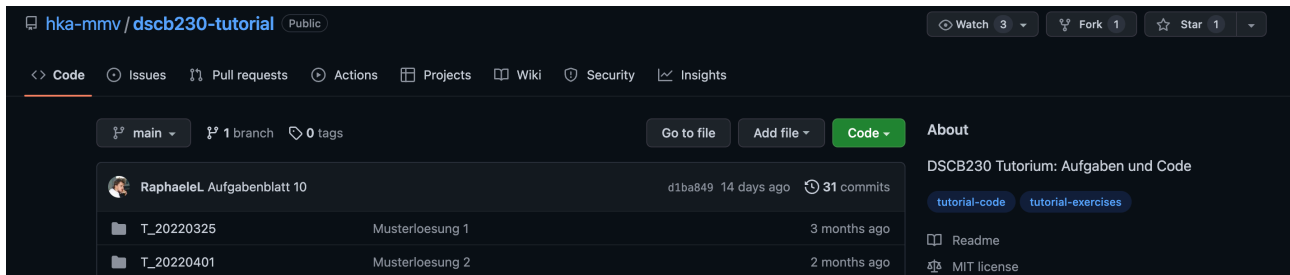
Wird hingegen aus der feature Branch, welche auf Basis der zweiten master Branch Version erstellt wurde, „git rebase master“ ausgeführt, so werden alle Veränderungen, die in den Commits der feature Branch getätigt wurden, gespeichert und auf den aktuellen Commit der master Branch ausgeführt. Somit wird im Endeffekt die feature Branch so abgeändert, sodass in der Form ist als wäre sie auf Basis der neusten master Branch Version erstellt worden.



Probiert den Command selbst aus, indem ihr das dargestellte Diagramm mithilfe von Beispiel commits in eurem repo umsetzt.

2. Ein Git repo forken

Geht auf die Github Page des Tutoriums (<https://github.com/hka-mmV/dscb230-tutorial>) und forkt das repo indem ihr auf „Fork“ oben rechts drückt.



Der Unterschied zu "git clone" ist, dass man bei einem geklonten repo seine Änderungen nicht direkt pushen kann, wenn der Autor des repo nicht speziell ausgewählt hat, dass man push Rechte hat. Durch das Forken des repo wird dieses quasi kopiert und in ein eigenes erstelltes repo geladen, an dem man dann alle Rechte hat und herumexperimentieren kann, ohne das originale Projekt zu beeinflussen.

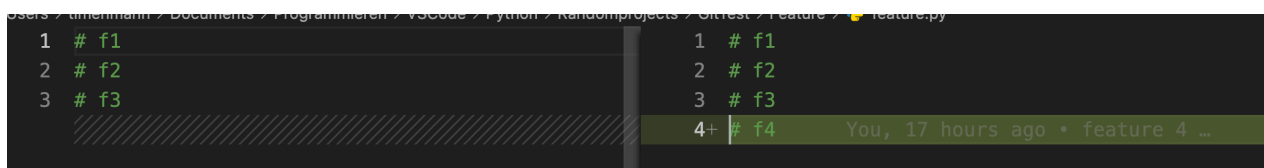
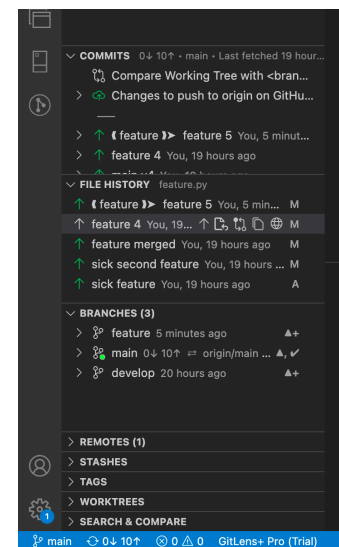
3. GitLens

GitLens ist eine open source Erweiterung für die Nutzung von Github in Visual Studio Code. Es bietet eine einfache und schnelle Navigation durch die Versionen der Dateien, zeigt Informationen zu Autor, Datum und Commitbeschreibung direkt in der ausgewählten Zeile an. Die Sidebar verleiht eine Übersicht über Branches, Commits, Dateihistorie, Contributors und vieles mehr.

Außerdem erlaubt GitLens für viele Funktionen eine Terminalfreie Nutzung von Github, wie push, pull, commit, branches wechseln, branches erstellen, etc.

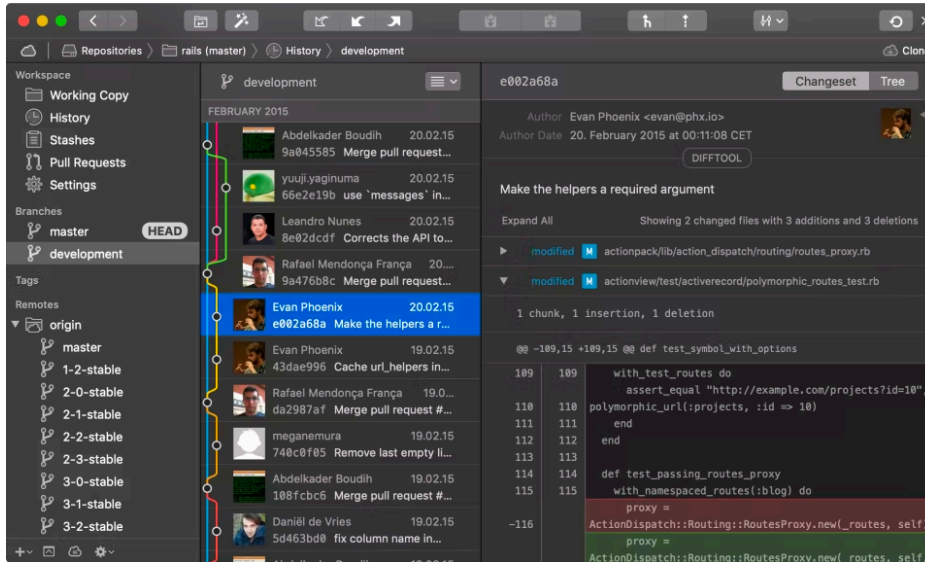


Auch kann mit nur 2 Clicks ein Überblick über die Veränderungen der aktuellen Datei durch die Commits gewonnen werden.



4. Git Tower

Git Tower hat wie GitLens die Funktion, einen besseren Überblick über die commit Historie sowie Autoren, Branches etc. eines repos zu gewährleisten und Git Commands wie Push, Pull, Branches erstellen und wechseln etc. direkt ins GUI zu integrieren. Git Tower ist jedoch nicht wie GitLens eine VSCode Erweiterung sondern ein eigenständiges Programm. Als Student kann man jedoch eine kostenfreie Lizenz hierfür beantragen, anderen.
Hier ein Beispiel aus dem GUI des Programmes.



5. Travis CI vs Jenkins

Travis CI und Jenkins sind beides CI/CD Plattformen, welche inkrementelle Codeänderungen, welche von den Entwicklern vorgenommen werden, verknüpfen und in eine fertige Software packen. Deren Funktionalität wird dann durch automatisierte Tests verifiziert (wie die automatisierten Tests der Github Abgaben der Aufgaben von Herrn Küppers). Ziel ist es, mögliche Fehler frühzeitig zu erkennen und dadurch Verzögerungen zu vermeiden, die Produktivität zu erhöhen und schnellere Release Zyklen zu ermöglichen.

Travis CI ist eine cloud basiertes CI/CD Plattform, welche für open Source Projekte gratis ist, für private Projekte jedoch nicht. Das Setup ist außerdem sehr einfach und Travis CI ist mit fast allen modernen Programmiersprachen kompatibel, jedoch nur mit den Versionsverwaltung Providern Github und Bitbucket. Travis CI ermöglicht außerdem, die geschriebene Software gleichzeitig mit verschiedenen Parametern zu testen wie verschiedene Backends, verschiedene Versionen der Programmiersprache oder libraries etc.

Jenkins ist eine open source CI/CD Plattform und daher gratis für alle Projekte. Jenkins ist nicht cloud basiert mit einer Vielzahl an Plugins, welche den CI/CD Workflow automatisieren, sodass diese scripts nicht selbst geschrieben werden müssen. Außerdem besteht eine Kompatibilität mit mehr Versionsverwaltungs Providern als nur Github und Bitbucket