Programming & Software versioning

D. Schneider

Programming & Software versioning

Daniel Schneider

- PhD Student
  - Big Data & Objective Evaluation
- TU Graz, Austria
- Institute for Driver Assistance and Connected Mobility
- AVL Deutschland, Karlsruhe
  - Objective Evaluation of lateral controlling ADAS
  - Fleet data evaluation
  - Knowledge Discovery

Motivation

Job perspective in the field of ADAS & AD

- Tense market situation (job market)
- This makes it more important to fulfill the requirement profiles

- Requirements:
  - Programming (Python, C, C++, MATLAB, …)
  - Libraries/Tools (Numpy, Pandas, PyTorch, Scikit, …)
  - Sensor knowledge (Camera, RADAR, LiDAR, Fieldbus, …)
  - Machine Learning (Mathematics)
  - Visualization (Tableau, plotly, matplotlib, …)
  - Agile processes
  - Docker
  - Often Linux knowledge

Agenda

Theory

1. Programming – Python3
2. Software versioning

Practical demo

1. Getting ready
2. Git & Python basics
3. Real world example

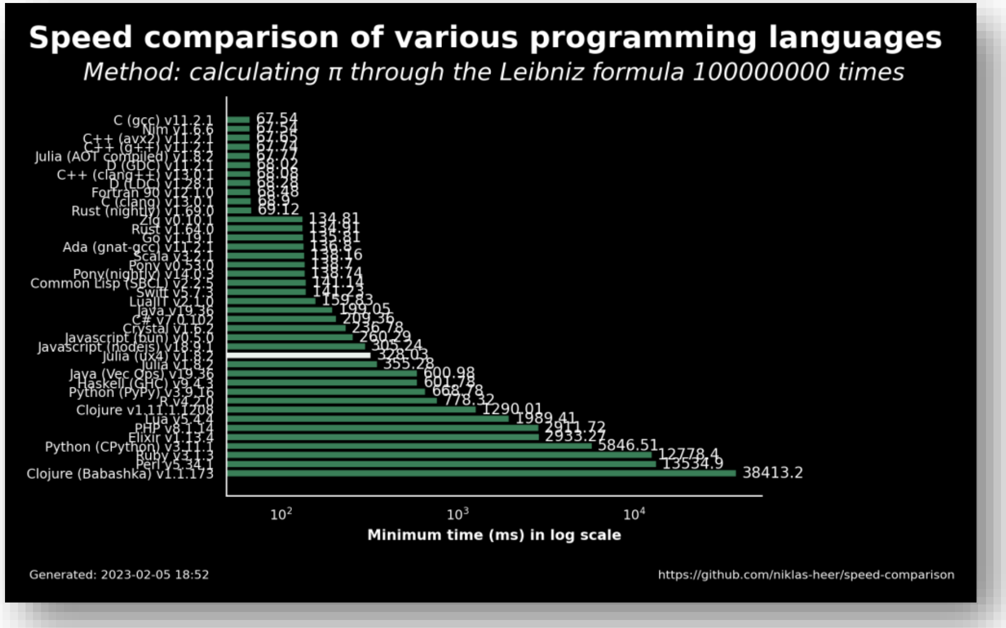
Chacon, Scott, and Ben Straub. *Pro git*. Springer Nature, 2014. Download

1. Programming – Python3

- Developed 1991 by Guido van Rossum
- Easy to learn high-level programming language
- Good readable due to indentation
- High impact in Data Science and Artificial Intelligence
- Open Source which is predominant in AI-research
- Large number of standard libraries available (Pandas, NumPy, PyTorch, …)
- Large community
- Object-orientated
- GUI programming possible
- Runs on any platform (Linux, Windows, Mac, …)
- Several IDEs available

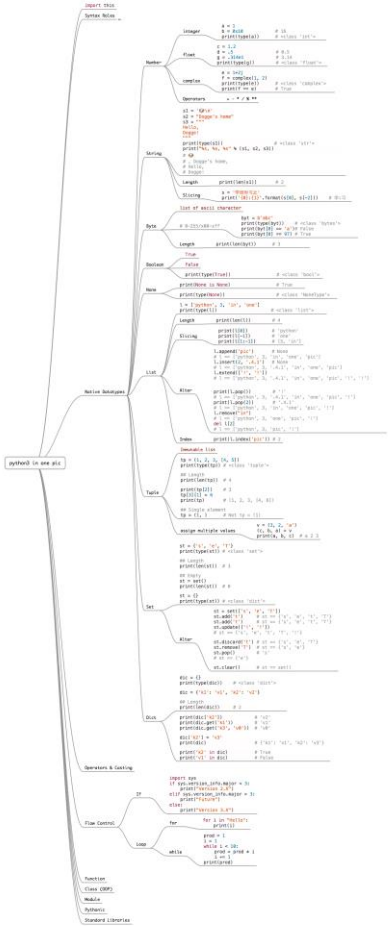
1. Programming – Python3

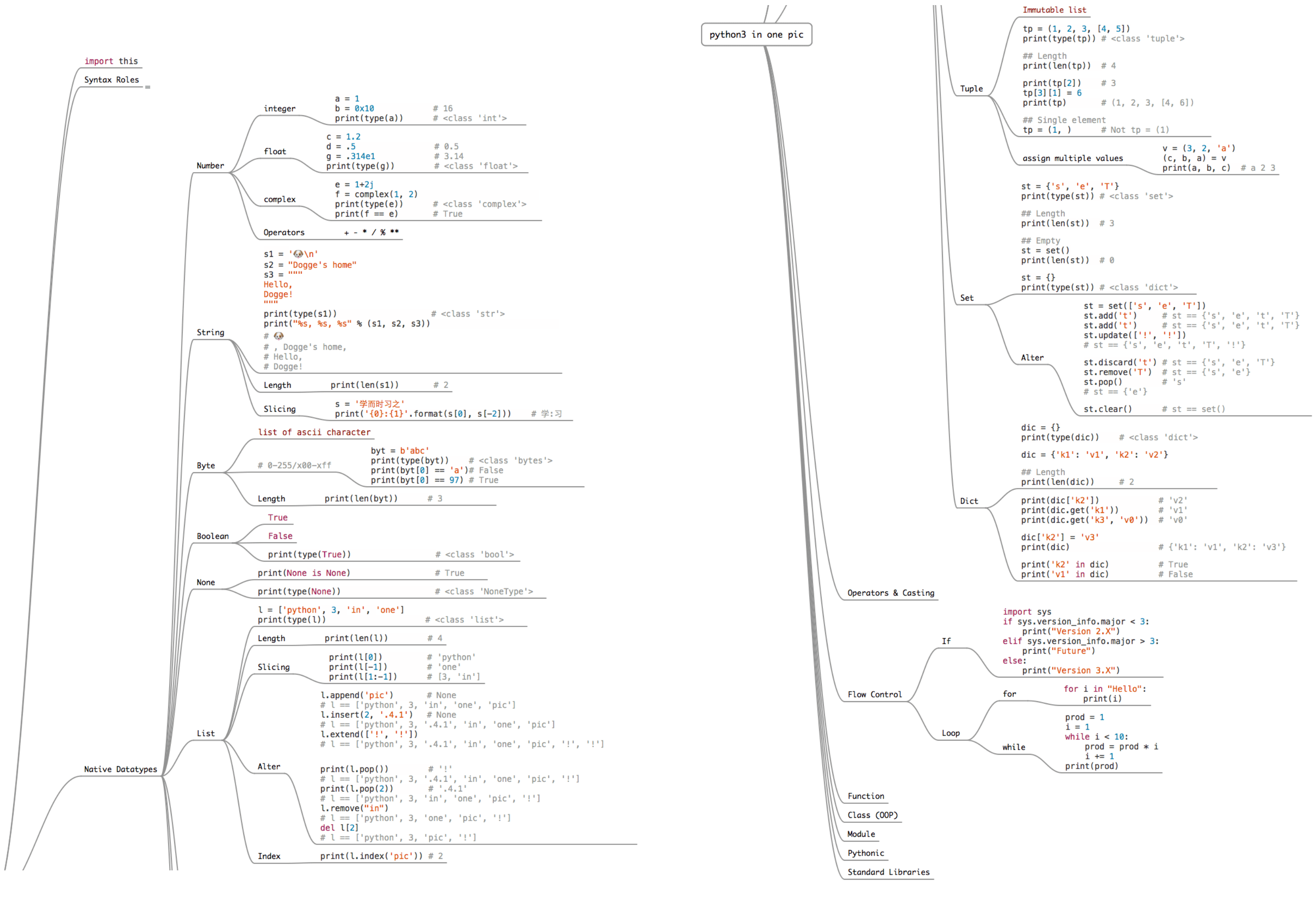
- Python is an interpreted language
- Python has no variable declaration
- Python is - relatively - slow in terms of computational speed
- Compared to other interpreted languages, there is no JIT-optimizer
- Optimized versions available (PyPy):
  - •PyPy translates Python code into machine code while the program is running
  - •PyPy tracks which parts of the code are used often (called hot loops) and compiles them into fast machine code
  - •It also optimizes memory usage and performs smart object allocation to reduce overhead
  - •PyPy has a more modern garbage collector, which often leads to better performance in memory-intensive programs
- However, Python is very well suited for rapid development!



1. Programming – Python3

- Python in one graph





python3 in one pic

import this

Syntax Roles

Native Datatypes

Number

integer

```
a = 1
b = 0x10               # 16
print(type(a))         # <class 'int'>
```

float

```
c = 1.2
d = .5                  # 0.5
g = .314e1              # 3.14
print(type(g))          # <class 'float'>
```

complex

```
e = 1+2j
f = complex(1, 2)
print(type(e))         # <class 'complex'>
print(f == e)          # True
```

Operators

```
+ - * / % **
```

String

```
s1 = '🐶\n'
s2 = "Dogge's home"
s3 = """
Hello,
Dogge!
"""
print(type(s1))                    # <class 'str'>
print("%s, %s, %s" % (s1, s2, s3))
# 🐶
# , Dogge's home,
# Hello,
# Dogge!
```

Length

```
print(len(s1))         # 2
```

Slicing

```
s = '学而时习之'
print('{0}:{1}'.format(s[0], s[-2]))    # 学:习
```

Byte

list of ascii character

```
# 0-255/x00-xff
```

```
byt = b'abc'
print(type(byt))    # <class 'bytes'>
print(byt[0] == 'a')# False
print(byt[0] == 97) # True
```

Length

```
print(len(byt))       # 3
```

Boolean

```
True
False
print(type(True))                  # <class 'bool'>
```

None

```
print(None is None)                # True
print(type(None))                  # <class 'NoneType'>
```

List

```
l = ['python', 3, 'in', 'one']
print(type(l))                     # <class 'list'>
```

Length

```
print(len(l))         # 4
```

Slicing

```
print(l[0])          # 'python'
print(l[-1])         # 'one'
print(l[1:-1])       # [3, 'in']
```

Alter

```
l.append('pic')        # None
# l == ['python', 3, 'in', 'one', 'pic']
l.insert(2, '.4.1')    # None
# l == ['python', 3, '.4.1', 'in', 'one', 'pic']
l.extend(['!', '!'])
# l == ['python', 3, '.4.1', 'in', 'one', 'pic', '!', '!']

print(l.pop())          # '!'
# l == ['python', 3, '.4.1', 'in', 'one', 'pic', '!']
print(l.pop(2))         # '.4.1'
# l == ['python', 3, 'in', 'one', 'pic', '!']
l.remove("in")
# l == ['python', 3, 'one', 'pic', '!']
del l[2]
# l == ['python', 3, 'pic', '!']
```

Index

```
print(l.index('pic')) # 2
```

Tuple

Immutable list

```
tp = (1, 2, 3, [4, 5])
print(type(tp)) # <class 'tuple'>

## Length
print(len(tp))  # 4

print(tp[2])     # 3
tp[3][1] = 6
print(tp)        # (1, 2, 3, [4, 6])

## Single element
tp = (1, )       # Not tp = (1)
```

assign multiple values

```
v = (3, 2, 'a')
(c, b, a) = v
print(a, b, c)  # a 2 3
```

Set

```
st = {'s', 'e', 'T'}
print(type(st)) # <class 'set'>

## Length
print(len(st))  # 3

## Empty
st = set()
print(len(st))  # 0

st = {}
print(type(st)) # <class 'dict'>
```

Alter

```
st = set(['s', 'e', 'T'])
st.add('t')     # st == {'s', 'e', 't', 'T'}
st.add('t')     # st == {'s', 'e', 't', 'T'}
st.update(['!', '!'])
# st == {'s', 'e', 't', 'T', '!'}

st.discard('t') # st == {'s', 'e', 'T'}
st.remove('T')  # st == {'s', 'e'}
st.pop()        # 's'
# st == {'e'}

st.clear()      # st == set()
```

Dict

```
dic = {}
print(type(dic))    # <class 'dict'>

dic = {'k1': 'v1', 'k2': 'v2'}

## Length
print(len(dic))     # 2

print(dic['k2'])            # 'v2'
print(dic.get('k1'))        # 'v1'
print(dic.get('k3', 'v0'))  # 'v0'

dic['k2'] = 'v3'
print(dic)                  # {'k1': 'v1', 'k2': 'v3'}

print('k2' in dic)          # True
print('v1' in dic)          # False
```

Operators & Casting

Flow Control

If

```
import sys
if sys.version_info.major < 3:
    print("Version 2.X")
elif sys.version_info.major > 3:
    print("Future")
else:
    print("Version 3.X")
```

Loop

for

```
for i in "Hello":
    print(i)
```

while

```
prod = 1
i = 1
while i < 10:
    prod = prod * i
    i += 1
print(prod)
```

Function

Class (OOP)

Module

Pythonic

Standard Libraries

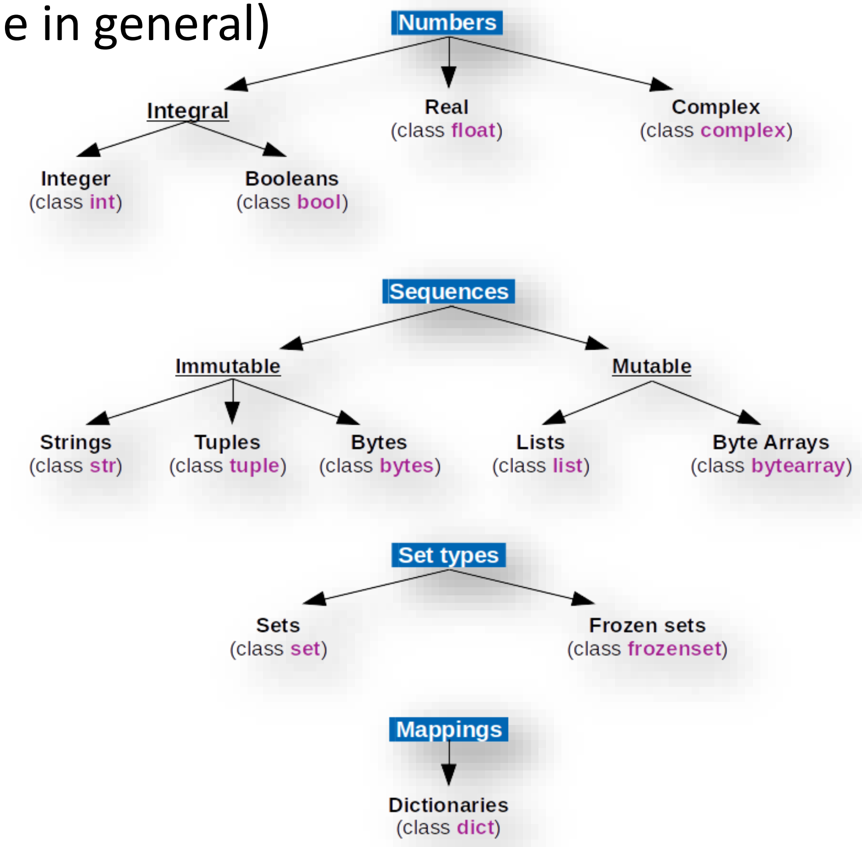
1. Programming – Python3

1_data_types.iypnb

- Use variables without declaring them
- Associativity: Left to right
- Call by “object reference” (neither call by reference, nor call by value in general)
- Do not have to deal with dereference etc. (compare to C/C++)

- Data types can be classified into:
  - Numbers
  - Sequences
  - Sets
  - Mappings

- Classes



1. Programming – Python3

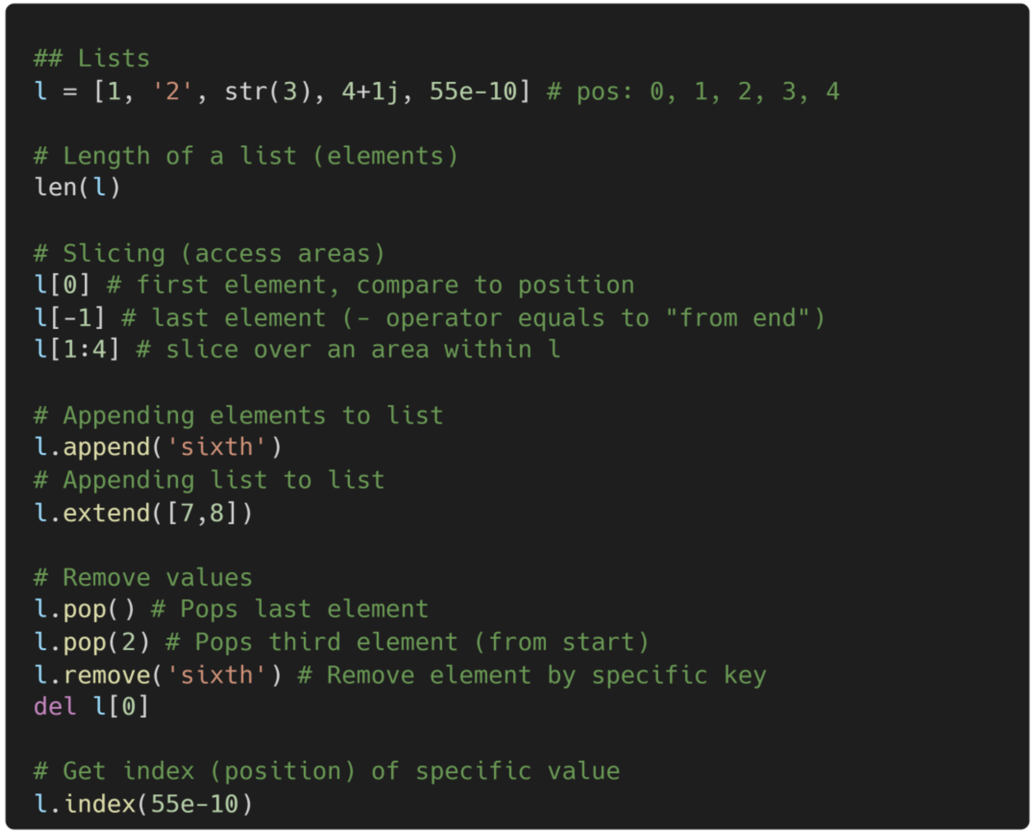
Sequences

- Stores multiple numbers within one data sequence ($\mathbf{x} \in \mathbb{R}^{1\times n}$)
  - Time series data for instance
  - time = [0.1, 0.2, 0.3, 0.4, 0.5]

- Tuple: tup = (1, 2, 3)
- List: lis = [1, 2, 3]

- Various in-build methods available
  - append, pop, remove, del, index, …

- Lists of list also possible ($\mathbf{X} \in \mathbb{R}^{n\times m}$)
  - Greyscale image
  - img = [[1, 2, 3], [1, 2, 3], [1, 2, 3]]



```python
## Lists
l = [1, '2', str(3), 4+1j, 55e-10] # pos: 0, 1, 2, 3, 4

# Length of a list (elements)
len(l)

# Slicing (access areas)
l[0] # first element, compare to position
l[-1] # last element (- operator equals to "from end")
l[1:4] # slice over an area within l

# Appending elements to list
l.append('sixth')
# Appending list to list
l.extend([7,8])

# Remove values
l.pop() # Pops last element
l.pop(2) # Pops third element (from start)
l.remove('sixth') # Remove element by specific key
del l[0]

# Get index (position) of specific value
l.index(55e-10)
```

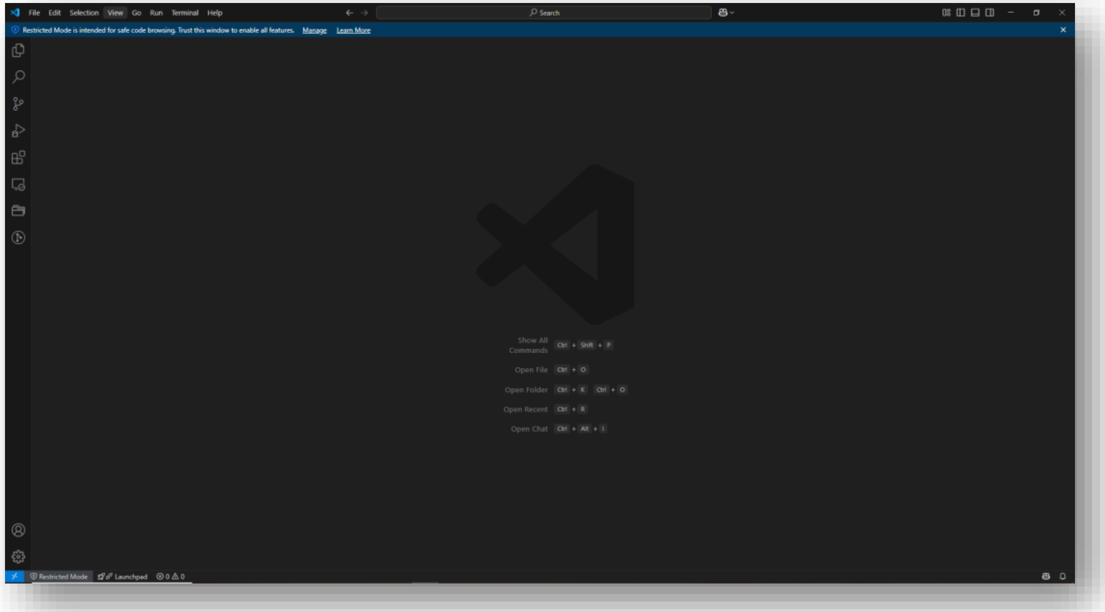
1. Programming – Python3

What to do with Python code?

- Python 3.10 sufficient for the course
- PIP is the package installer for Python
- Virtual environments (venv) are often used to handle dependencies in different projects

Where to code?

- Microsoft Visual Studio code (Click)
- PyCharm (free, pro-version for students available)
- Jupyter Notebook / Anaconda



Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More

Show All Commands	Ctrl + Shift + P
Open File	Ctrl + O
Open Folder	Ctrl + K Ctrl + O
Open Recent	Ctrl + R
Open Chat	Ctrl + Alt + I

1. Programming – Python3

PEP - Python Enhancement Proposals

- PEPs are design documents providing information and guidelines for Python's development and community.
- They propose new features, clarify design issues, and serve as reference points for Python's evolution.

PEP 8 - Style Guide for Python Code

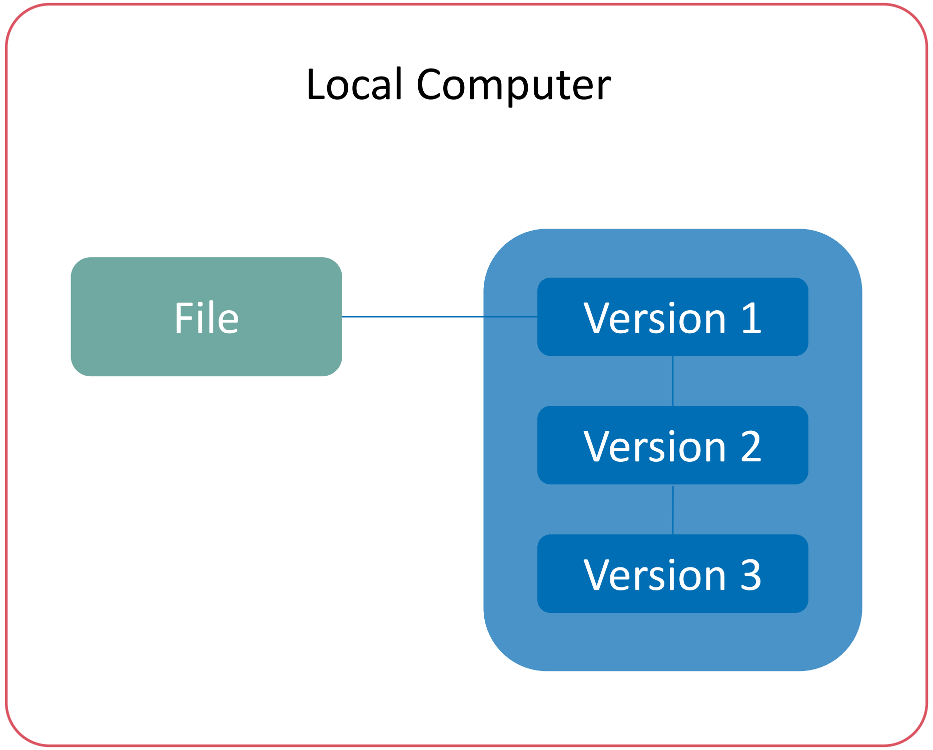
- Sets conventions for writing clear, consistent, and readable Python code
- Promotes collaboration by making code easier to share and maintain
- Covers naming conventions, indentation, spacing, comments, and more
- Following PEP 8 ensures code quality and minimizes errors in collaborative environments.

2. Software versioning

- To have control over existing software versions, a Version Control System (VCS) is required
- VCS track changes and differentiate between multiple stages/versions
- Control different branches of your software project
- Tagging of „final“ (release) versions
- GitHub, GitLab, Gitea, Bitbucket, SVN, …

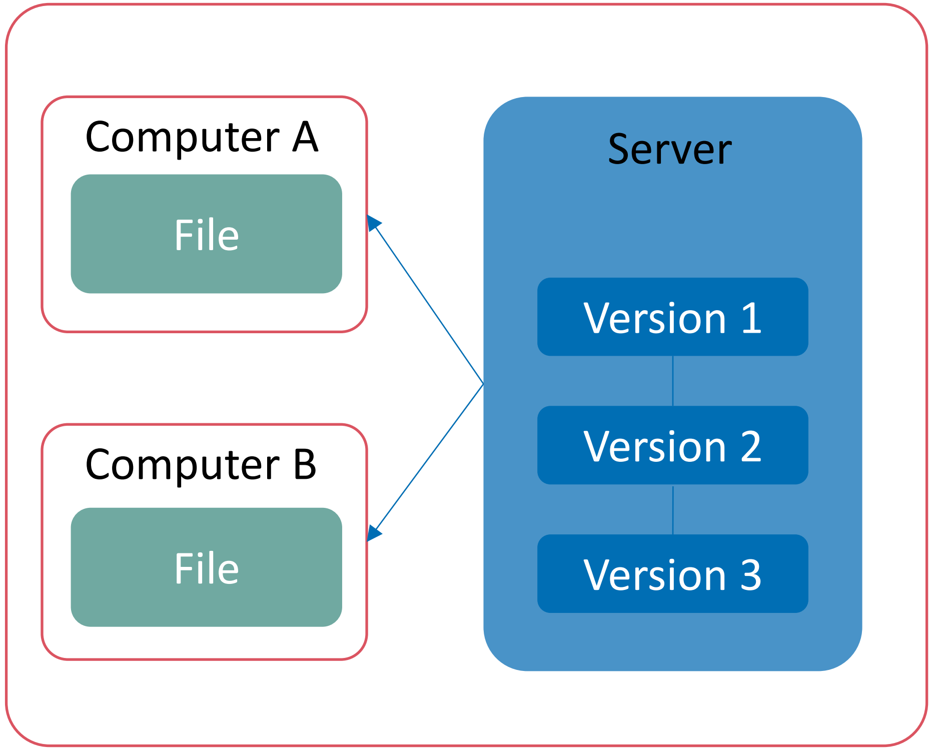
Local VCS

- RCS (Revision Control System) is an early version control tool still available on many systems
- It stores differences (patch sets) between file versions rather than full copies
- Files can be reconstructed at any point by applying the stored patches in sequence



Centralized VCS

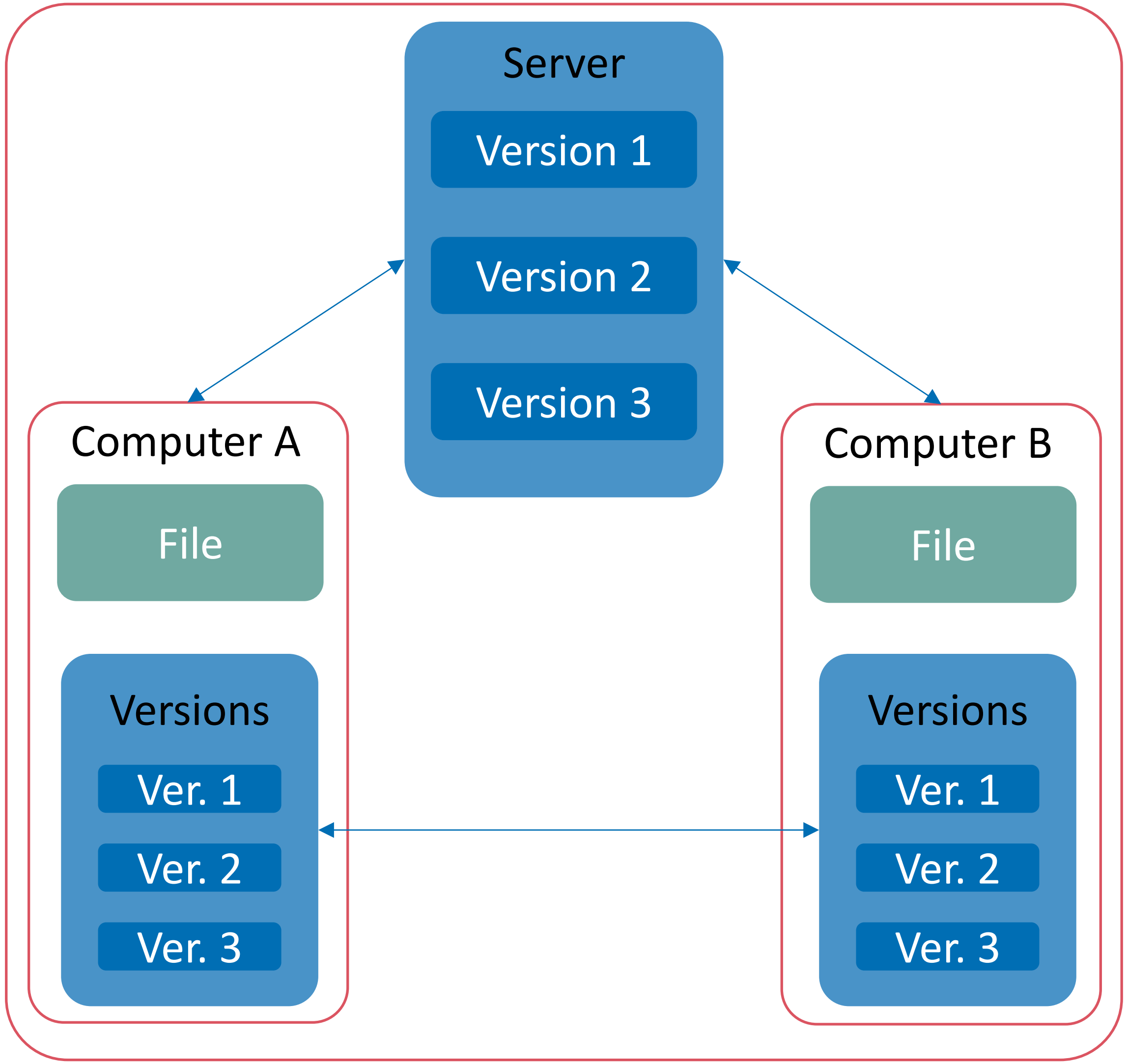
- Centralized Version Control Systems (CVCS) use a single central server to manage versioned files, with clients checking files out from that server
- Enables team collaboration, visibility into others' work, and centralized access control for administrators
- Single point of failure: if the server goes down or is corrupted, no one can collaborate or save changes
- Without proper backups, the entire project history can be lost, similar to local VCS systems



2. Software versioning

Distributed VCS

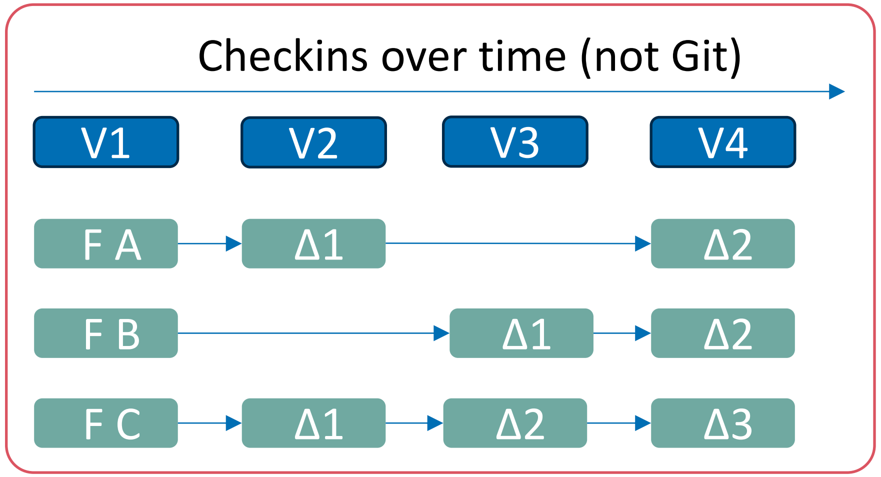
- Distributed Version Control Systems (DVCS) like Git mirror the entire repository to each client
- Every clone is a full backup, so if a server fails, it can be restored from any client
- Supports multiple remote repositories, enabling flexible and simultaneous collaboration
- Allows advanced workflows (e.g., hierarchical models) not possible with centralized systems



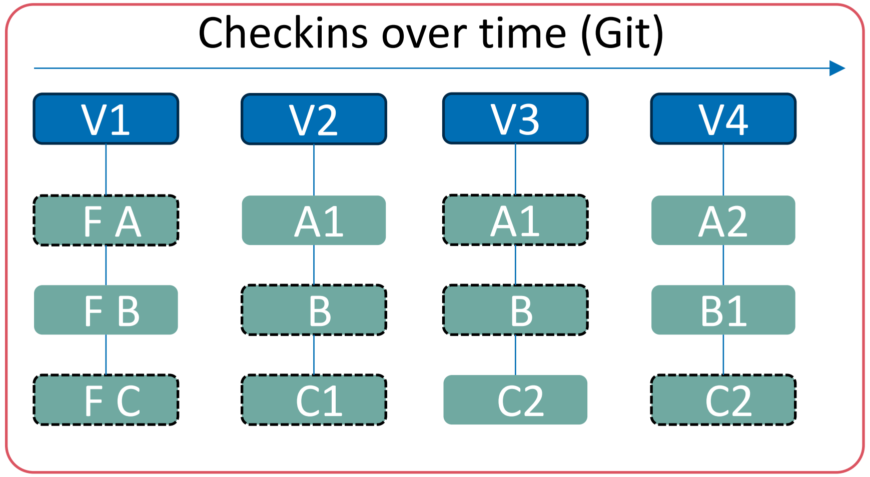
2. Software versioning

Git

- Developed in 2005 by Linus Torvalds
- Very performant, widely used
- Supports decentralized collaboration



- Most VCS tools (like Subversion, CVS, Perforce) store data as a series of file-based changes over time
- Git stores data as snapshots of the entire project at each commit - like a miniature filesystem
- If a file hasn't changed, Git simply links to the previous version instead of storing it again
- Git treats project history as a stream of snapshots, not a sequence of diffs

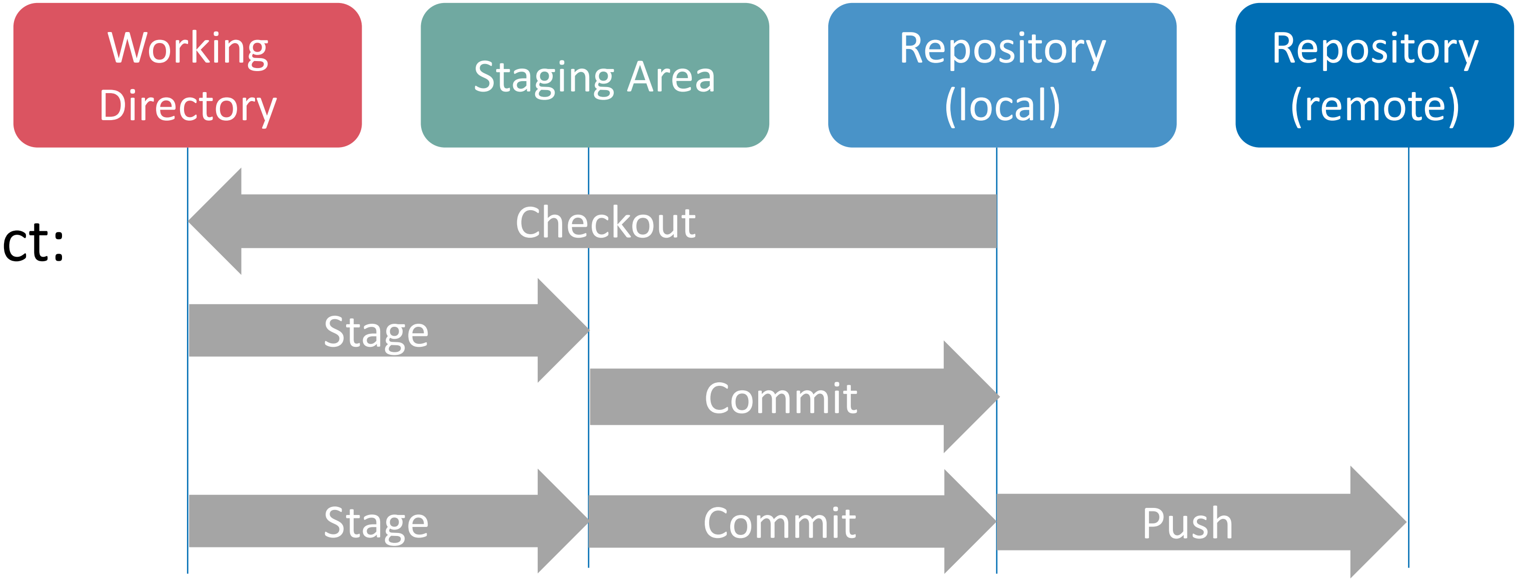


Git workflow – Three steps to maintain:

1. Modify files (working directory)
2. Stage changes (`git add`)
3. Commit snapshot (`git commit`)

Three main sections of a Git project:

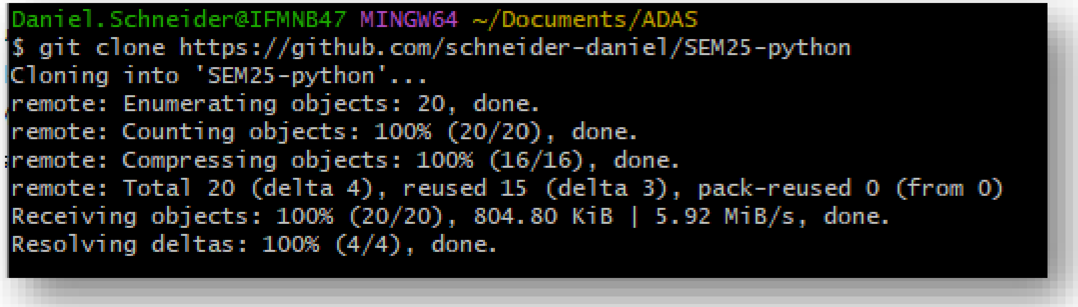
1. Git directory – stores all metadata & objects
2. Working directory – files, pulled from the Git directory
3. Staging area (index) – snapshot of what will be committed next



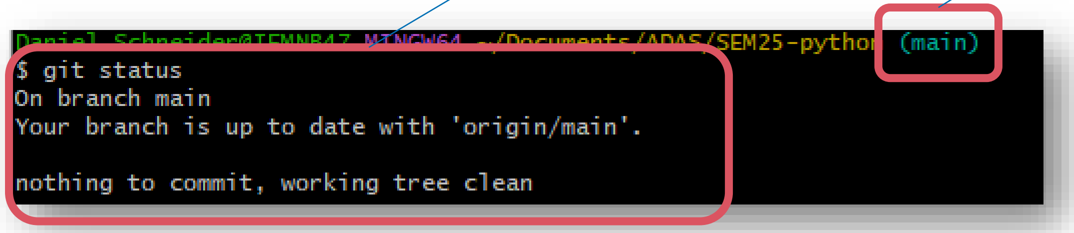
2. Software versioning

Working with Git

1. Once a repository is initialized or pulled, git status shows the status of the files



```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS
$ git clone https://github.com/schneider-daniel/SEM25-python
Cloning into 'SEM25-python'...
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (20/20), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 20 (delta 4), reused 15 (delta 3), pack-reused 0 (from 0)
Receiving objects: 100% (20/20), 804.80 KiB | 5.92 MiB/s, done.
Resolving deltas: 100% (4/4), done.
```



Status of the files

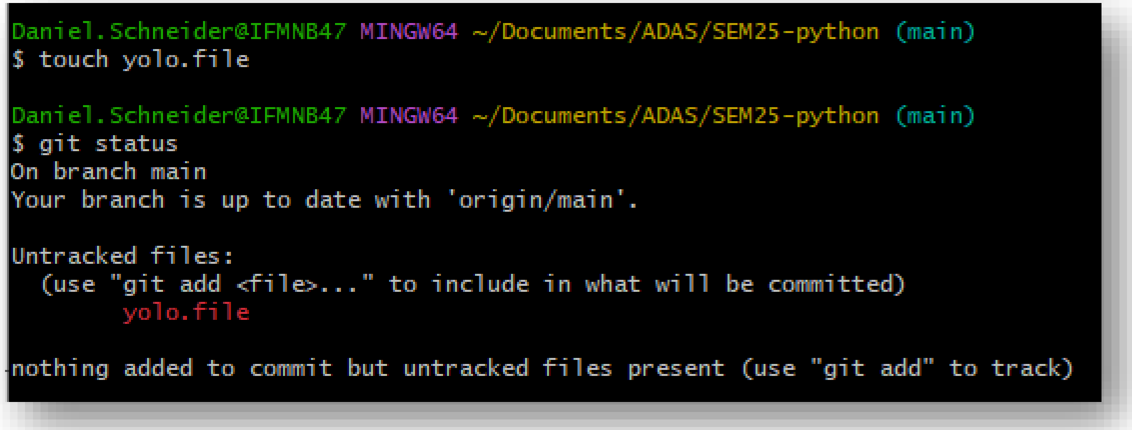
Actual branch

```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

2. Software versioning

Working with Git

1. Once a repository is initialized or pulled, git status shows the status of the files
2. Adding a new file changes the status
   1. `touch yolo.file`
   2. `git status`
3. The added file is untracked, Git sees it as a new file that wasn't in the last commit (snapshot)
4. Git won't include untracked files in commits unless you explicitly add them
5. This prevents accidentally committing things like build artifacts or temporary files.
6. To start tracking a file write
   1. `git add yolo.file`

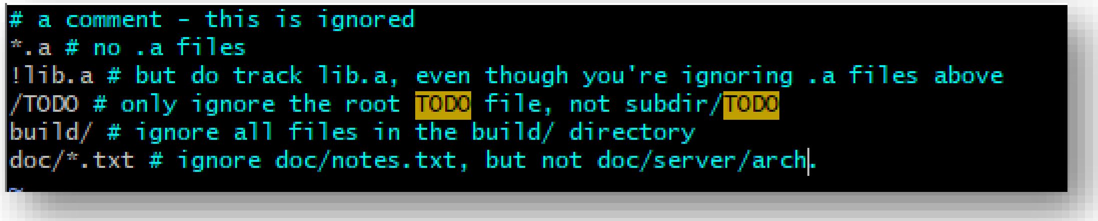


```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ touch yolo.file

Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        yolo.file

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

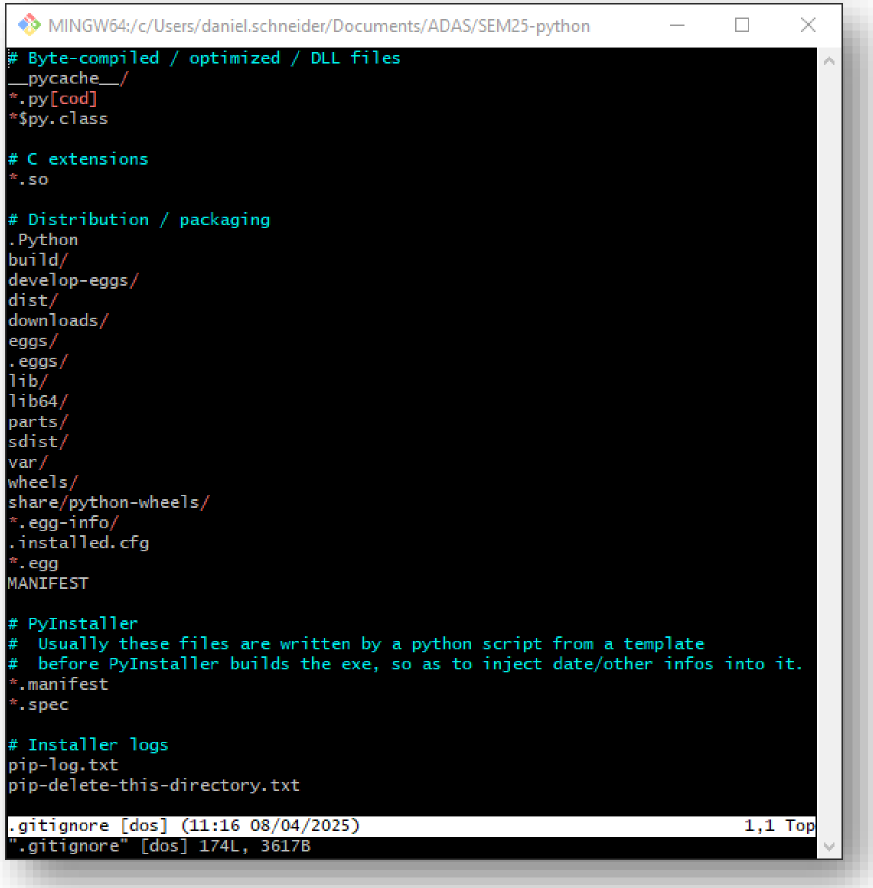
2. Software versioning

Working with Git

1. Ignoring files and directories can be done via .gitignore file
2. Either generic extensions, files, or directories can be ignored for the staging



```
# a comment - this is ignored
*.a # no .a files
!lib.a # but do track lib.a, even though you're ignoring .a files above
/TODO # only ignore the root TODO file, not subdir/TODO
build/ # ignore all files in the build/ directory
doc/*.txt # ignore doc/notes.txt, but not doc/server/arch
```

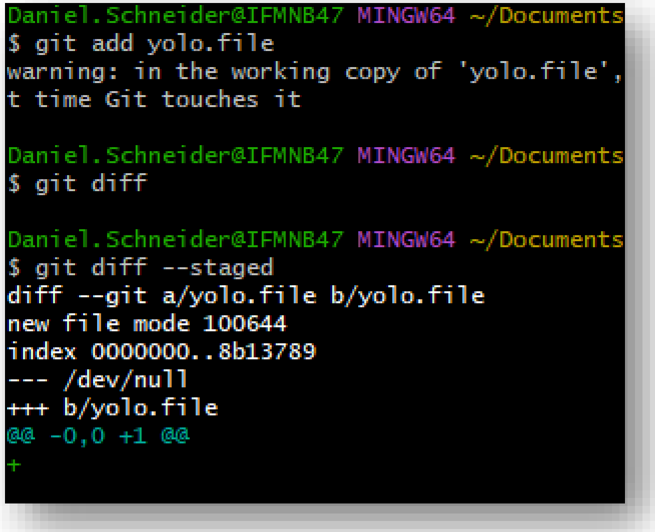

MINGW64:/c/Users/daniel.schneider/Documents/ADAS/SEM25-python

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
share/python-wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST

# PyInstaller
#  Usually these files are written by a python script from a template
#  before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt

.gitignore [dos] (11:16 08/04/2025)                                      1,1 Top
".gitignore" [dos] 174L, 3617B
```

2. Software versioning

Working with Git

1. `git status` shows which files have been modified or staged, but not what exactly changed.
2. To see detailed changes:
3. Use `git diff` to view unstaged changes (what you've modified but not yet staged).
4. Use `git diff --staged` (or `--cached`) to view staged changes (what will be committed).



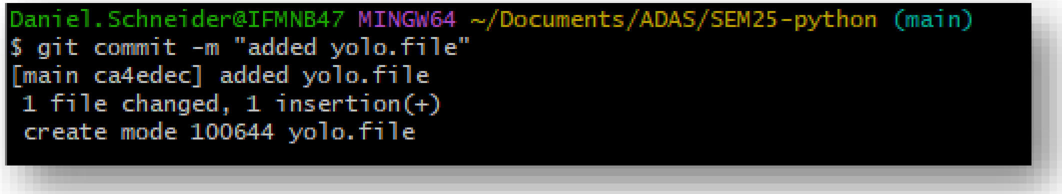
```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents
$ git add yolo.file
warning: in the working copy of 'yolo.file',
t time Git touches it

Daniel.Schneider@IFMNB47 MINGW64 ~/Documents
$ git diff

Daniel.Schneider@IFMNB47 MINGW64 ~/Documents
$ git diff --staged
diff --git a/yolo.file b/yolo.file
new file mode 100644
index 0000000..8b13789
--- /dev/null
+++ b/yolo.file
@@ -0,0 +1 @@
+
```

2. Software versioning

Working with Git

1. Git commit stages files (added via git add) in a commit.
2. Unstaged changes remain on disk and will not be committed.
3. To commit staged changes:

```
git commit –m “commit message”
```



```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git commit -m "added yolo.file"
[main ca4edec] added yolo.file
 1 file changed, 1 insertion(+)
 create mode 100644 yolo.file
```

Working with Git

- To remove a file from Git, you must remove it from your tracked files (more accurately, remove it from your staging area) and then commit
- To do so, use:

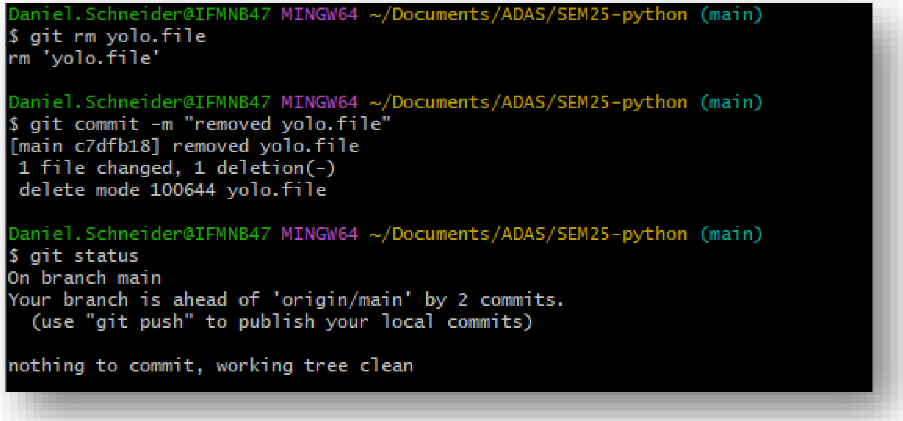
```
git rm yolo.file

git commit –m „removed yolo.file“
```

- If a file is already staged (added) and you want to remove it, you must force removal:

```
git rm -f <filename>
```

- This protects against accidentally removing uncommitted work
- To stop tracking a file but keep it on disk, use:

```
git rm --cached <filename>
```



```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git rm yolo.file
rm 'yolo.file'

Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git commit -m "removed yolo.file"
[main c7dfb18] removed yolo.file
 1 file changed, 1 deletion(-)
 delete mode 100644 yolo.file

Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

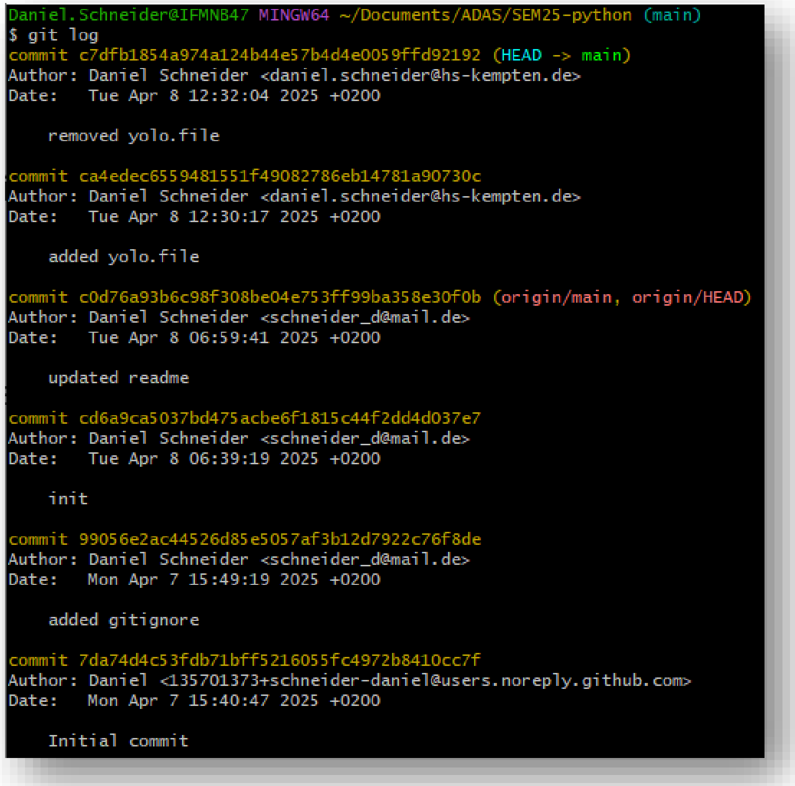
Working with Git

- Git does not track renames explicitly—it detects them by comparing file content between commits
- The `git mv` command is a convenience tool that:
  - Renames the file on disk
  - Stages the deletion of the old file and addition of the new one in one step

Working with Git

- After making commits or cloning a repo, use git log to view the project’s history `git log` shows:
    - Commit ID (SHA)
    - Author
    - Date
    - Commit message

- Useful for reviewing who changed what, and when



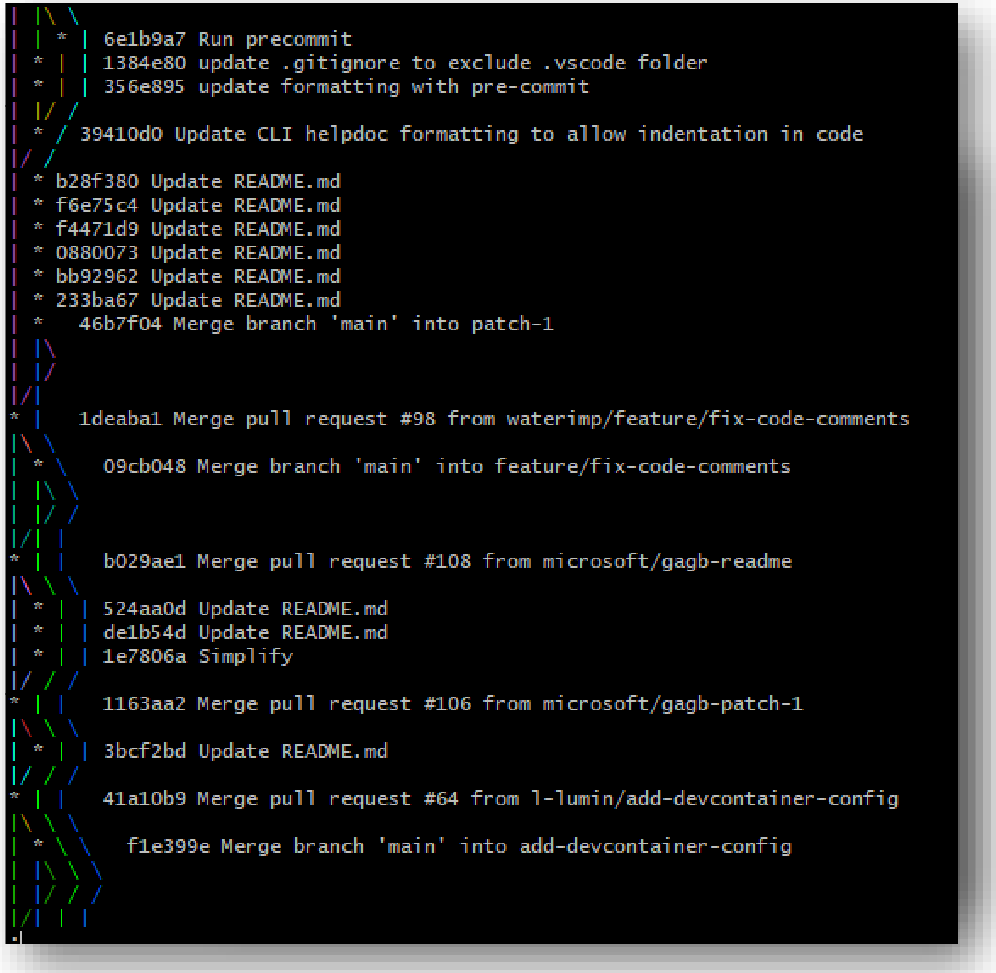
```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git log
commit c7dfb1854a974a124b44e57b4d4e0059ffd92192 (HEAD -> main)
Author: Daniel Schneider <daniel.schneider@hs-kempten.de>
Date:   Tue Apr 8 12:32:04 2025 +0200

    removed yolo.file

commit ca4edec6559481551f49082786eb14781a90730c
Author: Daniel Schneider <daniel.schneider@hs-kempten.de>
Date:   Tue Apr 8 12:30:17 2025 +0200

    added yolo.file

commit c0d76a93b6c98f308be04e753ff99ba358e30f0b (origin/main, origin/HEAD)
Author: Daniel Schneider <schneider_d@mail.de>
Date:   Tue Apr 8 06:59:41 2025 +0200

    updated readme

commit cd6a9ca5037bd475acbe6f1815c44f2dd4d037e7
Author: Daniel Schneider <schneider_d@mail.de>
Date:   Tue Apr 8 06:39:19 2025 +0200

    init

commit 99056e2ac44526d85e5057af3b12d7922c76f8de
Author: Daniel Schneider <schneider_d@mail.de>
Date:   Mon Apr 7 15:49:19 2025 +0200

    added gitignore

commit 7da74d4c53fdb71bff5216055fc4972b8410cc7f
Author: Daniel <135701373+schneider-daniel@users.noreply.github.com>
Date:   Mon Apr 7 15:40:47 2025 +0200

    Initial commit
```

2. Software versioning

Working with Git

- After making commits or cloning a repo, use `git log` to view the history `git log` shows:
  - Commit ID (SHA)
  - Author
  - Date
  - Commit message

- Useful for reviewing who changed what, and when



```
| |\ \
| | * | 6e1b9a7 Run precommit
| * | | 1384e80 update .gitignore to exclude .vscode folder
| * | | 356e895 update formatting with pre-commit
| |/ /
| * / 39410d0 Update CLI helpdoc formatting to allow indentation in code
|/ /
| * b28f380 Update README.md
| * f6e75c4 Update README.md
| * f4471d9 Update README.md
| * 0880073 Update README.md
| * bb92962 Update README.md
| * 233ba67 Update README.md
| *   46b7f04 Merge branch 'main' into patch-1
| |\
| |/
|/|
* |   1deaba1 Merge pull request #98 from waterimp/feature/fix-code-comments
|\ \
| * \   09cb048 Merge branch 'main' into feature/fix-code-comments
| |\ \
| |/ /
|/| |
* | |   b029ae1 Merge pull request #108 from microsoft/gagb-readme
|\ \ \
| * | | 524aa0d Update README.md
| * | | de1b54d Update README.md
| * | | 1e7806a Simplify
|/ / /
* | |   1163aa2 Merge pull request #106 from microsoft/gagb-patch-1
|\ \ \
| * | | 3bcf2bd Update README.md
|/ / /
* | |   41a10b9 Merge pull request #64 from l-lumin/add-devcontainer-config
|\ \ \
| * \ \   f1e399e Merge branch 'main' into add-devcontainer-config
| |\ \ \
| |/ / /
|/| | |
```

2. Software versioning

Working with Git

- Undoing the last commit with `--amend`
- If you committed too early or made a mistake in your message or files, use:

```
git commit --amend
```

- This lets you:
  - Edit the commit message
  - Add new staged changes to the last commit
- It replaces the previous commit with a new one — the old commit is discarded

- If you accidentally staged multiple files and want to commit them separately, you can unstage specific files

```
git restore --staged <filename>
```

- This removes the file from the staging area but keeps your changes in the working directory

Working with Git

- If you've modified a file but don't want to keep the changes, you can revert it to the last committed version.

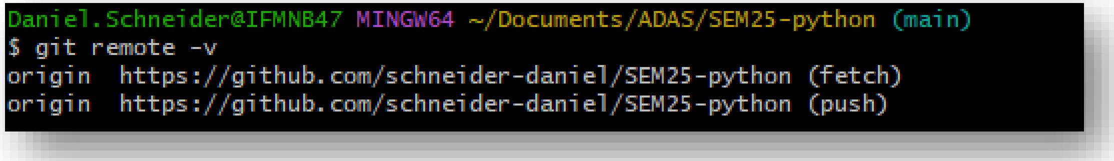
- Use this command to discard unstaged changes:

```
git checkout -- <filename>  # or
git restore <filename>  # in newer versions
```

- This resets the file to match the last commit
- Changes will be lost, so use with caution!

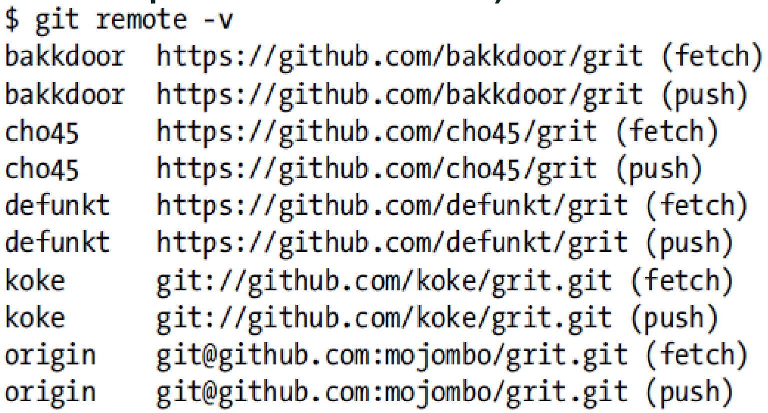
Working with Git remotely

- If you've cloned a repository from a remote branch, you can check its addresses easily using `git remote -v`



```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git remote -v
origin  https://github.com/schneider-daniel/SEM25-python (fetch)
origin  https://github.com/schneider-daniel/SEM25-python (push)
```

- If there are multiple contributors, different remotes are existing:



```
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Working with Git remotely

- Using git fetch <remote> downloads new data (commits) from the remote
- Does not update your current branch or files!
- Safe way to review changes before integrating

- Using git pull <remote> is equivalent to: git fetch + git merge
- Downloads changes from the remote and merges them into your current branch
- Useful when your branch is set to track a remote branch
- The remote is automatically named origin
- Your local master branch is set to track the remote one

Use git fetch when:

- You want to see what changed before merging.
- You're working in a review-before-merge workflow

Use git pull when

- You're ready to sync your local branch with the latest remote updates

Working with Git remotely

- Using `git push <remote-name> <branch-name>` sends your commits from the local branch to the corresponding remote branch
- Common default:
  - origin = the remote repo you cloned from
  - master (or main) = the default branch
- You must have write-access to the remote
- If someone else has pushed before you:
  - Your push will be rejected
  - You need to

    `git pull` (`git fetch` + `git merge` to update the code)

    to fetch and merge their changes before pushing again
- Use git push to share your changes with teammates or deploy to remote repositories

Working with Git remotely

- Using `git push <remote-name> <branch-name>` sends your commits from the local branch to the corresponding remote branch
- Common default:
  - origin = the remote repo you cloned from
  - master (or main) = the default branch
- You must have write-access to the remote
- If someone else has pushed before you:
  - Your push will be rejected
  - You need to

    `git pull (git fetch + git merge` to update the code)

    to fetch and merge their changes before pushing again
- Use git push to share your changes with teammates or deploy to remote repositories

Working with Git remotely

- Tags mark important points in your project’s history. Typically used to mark release versions (e.g., v1.0, v2.1.3)
- Two main types of tags:
  - Lightweight tags – like a bookmark. It’s just a name pointing to a commit
  - Annotated tags – stored as full Git objects with metadata (tagger name, date, message)
- Create a lightweight tag:

```
git tag v1.0
```

- Create an annotated tag:

```
git tag -a v1.0 -m "Version 1.0 release“
```

- List tags of the remote repository

```
git tag
```

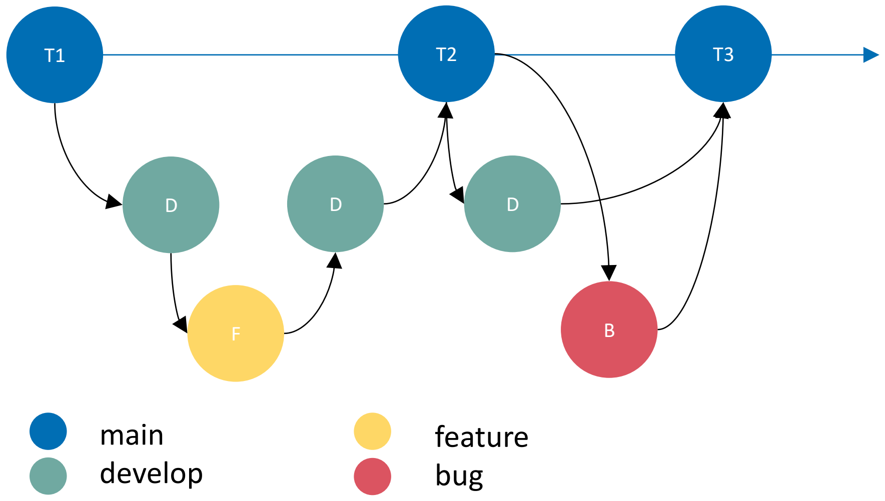
- Push tags to remote:

```
git push origin <tagname>

git push origin --tags
```

2. Software versioning

Branching in Git

- Branching = working on different features or ideas without affecting the main codebase
- In many VCSs, branching is slow and resource-heavy (creates full copies of the project)
- Git’s branching is:
    - Lightweight – branches are just pointers to commits
    - Fast – creating or switching branches is nearly instant
- Git makes frequent branching and merging easy, even multiple times per day
- This encourages flexible, experimental workflows:
    - Feature development
    - Bug fixing
    - Testing alternative ideas



2. Software versioning

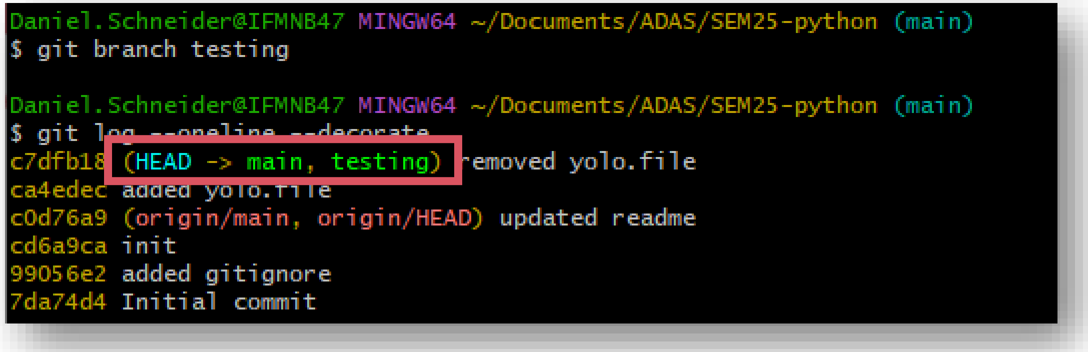
Branching in Git

- Create a new branch with

```
git branch <branch-name>
```

- Example:

```
git branch testing
```



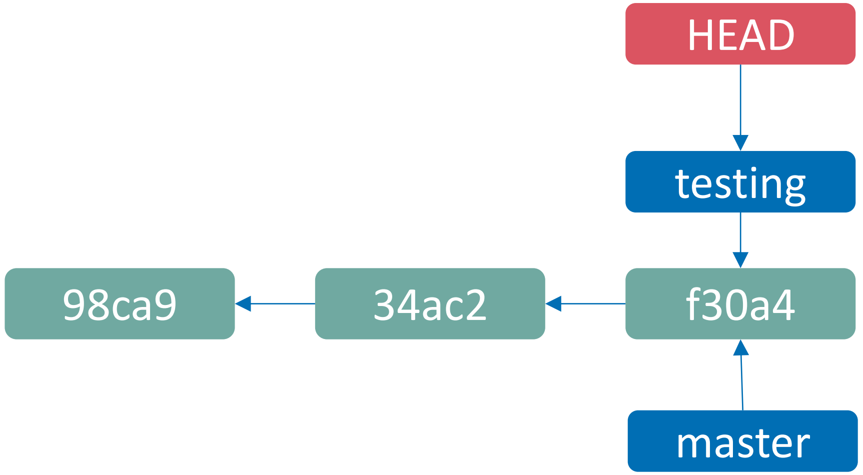
```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git branch testing

Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/SEM25-python (main)
$ git log --oneline --decorate
c7dfb18 (HEAD -> main, testing) removed yolo.file
ca4edec added yolo.file
c0d76a9 (origin/main, origin/HEAD) updated readme
cd6a9ca init
99056e2 added gitignore
7da74d4 Initial commit
```

- This creates a pointer to the current commit but does not switch to it
- Git tracks your current branch using a special pointer called HEAD
- To see where branches point:

```
git log --oneline --decorate
```

- It shows testing and master pointing to the same commit
- To switch to the new branch:

```
git checkout testing
git switch testing
```



2. Software versioning

Branching in Git

- Adding a test case in the testing branch and staging it

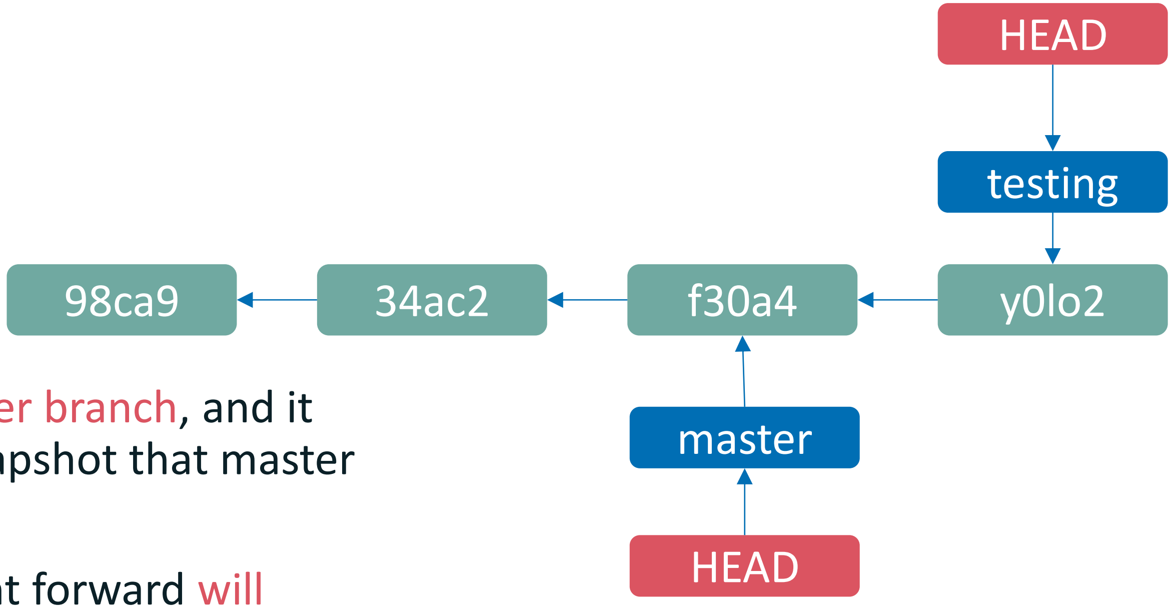
```
git add test.file
git commit –m „added test.file“
```

- The master branch points still to f30a4!

```
git switch master
```

- It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory to the snapshot that master points to
- This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you've done in your testing branch

SWITCHING BRANCHES CHANGES FILES IN YOUR WORKING DIRECTORY!



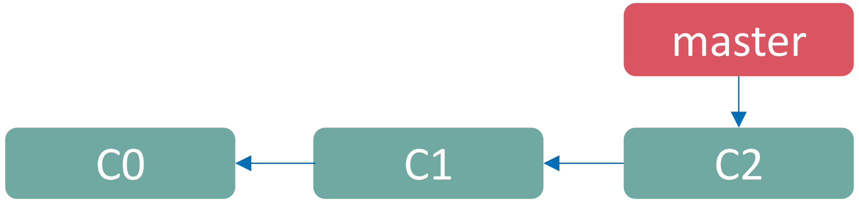
Branching in Git – A real world example

Situation:

- Working as a software developer on the field of AD
- Do work on a perception program and create a new branch for the feature-development (`feature_optical_flow (FO)`)
- During this, an issue came up and requires a hotfix immediately

Approach:

- Switch to your production branch (`master`)
- Create a hotfix-branch
- After fixing and testing merge hotfix and push to production (Fridays at 15:59)
- Switch back to your feature development

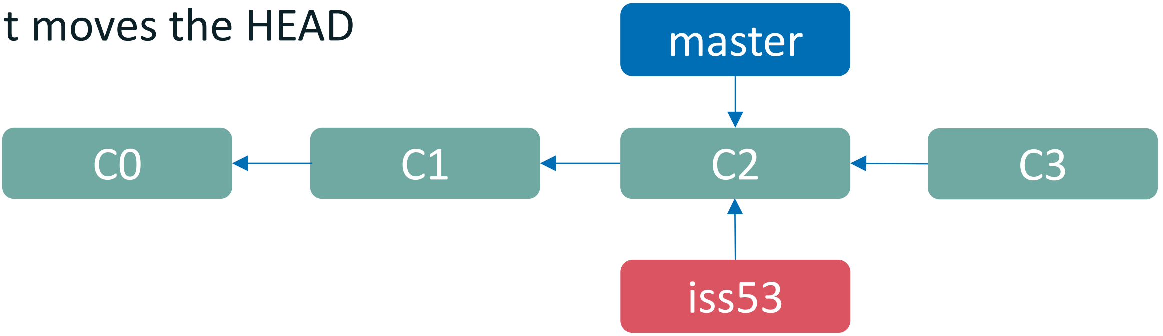


Branching in Git – A real world example

- Create a issue-branch → issue53

```
git checkout –b issue53  # or
git branch issue53
git checkout issue53
```

- Working on the bugfix and committing it moves the HEAD



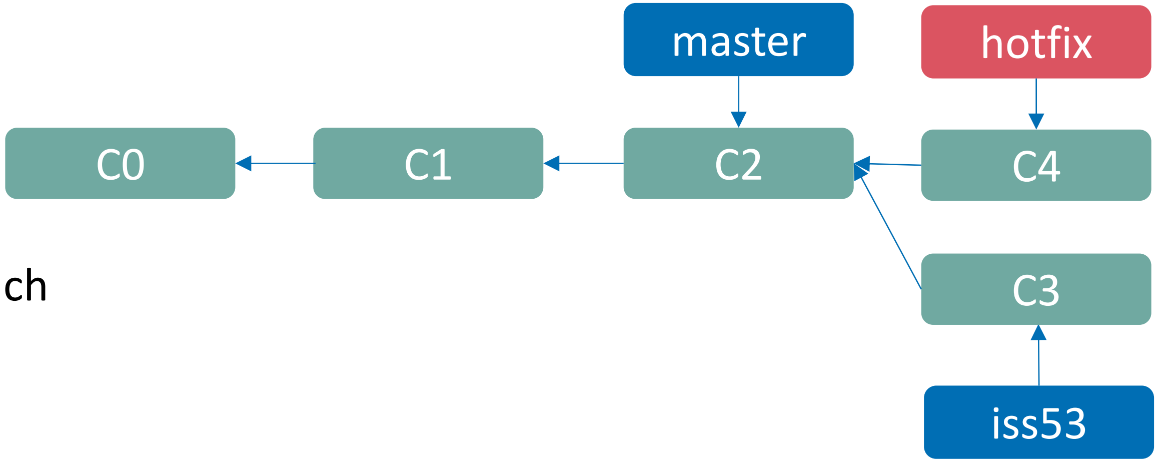
Branching in Git – A real world example

- During fixing the issue, another urged issue occurred
- There is no need to deploy the fix along with the iss53 changes you’ve made, and you don’t have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production
- All you must do is switch back to your master branch
- To avoid loss of data, stage everything in iss53

```
git add changedfile
git checkout master
```

- From the master branch, you branch out a hotfix branch

```
git checkout –b hotfix
```



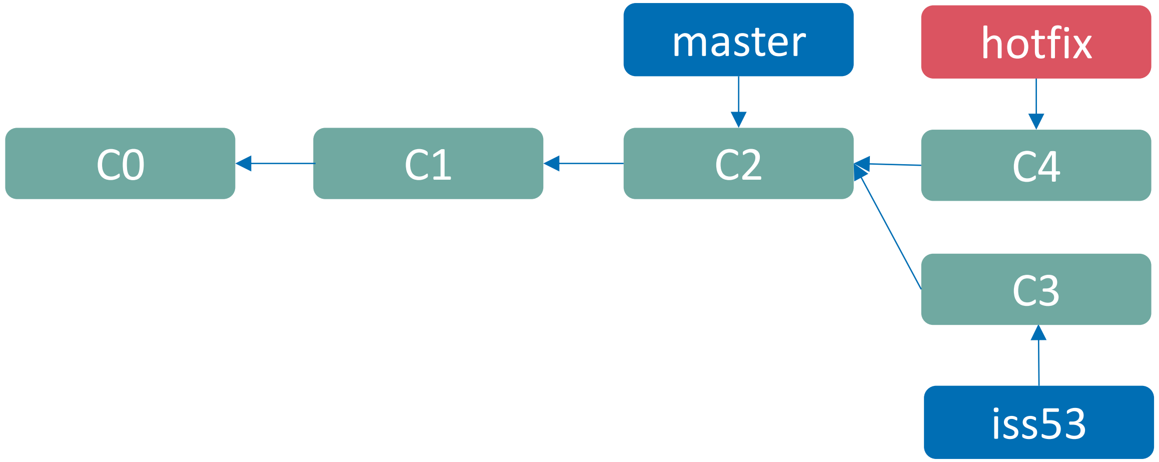
Branching in Git – A real world example

- Once you have finished development and testing on `hotfix`, it is merged back to production (`master`)
- First, stage and commit the changes

```
git commit -a -m 'fixed steering issue’
```

- Switch back to master and merge the hotfix into it

```
git checkout master
git merge hotfix
```



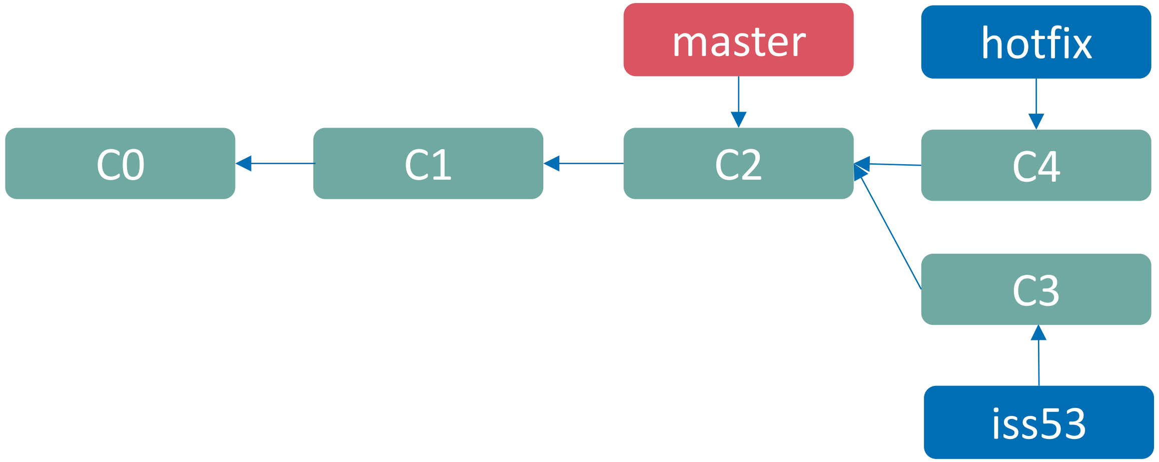
Branching in Git – A real world example

- Once you have finished development and testing on hotfix, it is merged back to production (master)
- First, stage and commit the changes

```
git commit -a -m 'fixed steering issue’
```

- Switch back to master and merge the hotfix into it

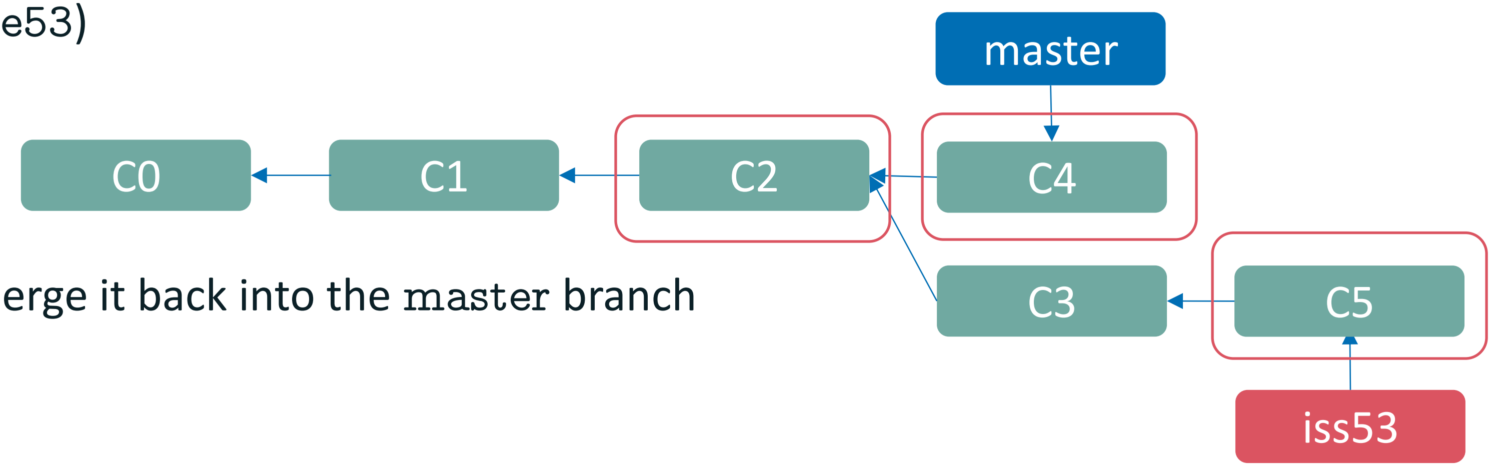
```
git checkout master
git merge hotfix
>> Fast-forward
```



Branching in Git – A real world example

- Working back again on the original issue (issue53)

```
git checkout issue53
git add changes
git commit –m “fixed issue53”
```



- Once the issue is there also solved, you can merge it back into the master branch

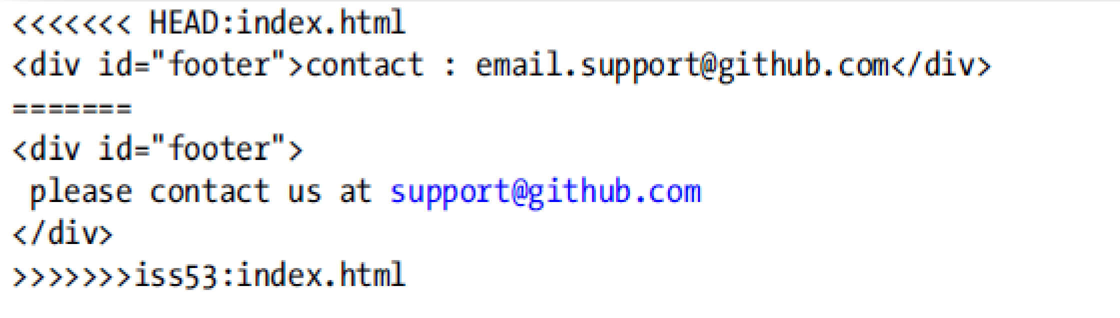
```
git checkout master

git merge issue53

>> Merge made by the `recursive`strategy
```

- This looks a bit different than the hotfix merge you did earlier, since the development history has diverged from some older point. Because the commit on the branch you’re on isn’t a direct ancestor of the branch you’re merging in, Git has to do some work
- In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two

2. Software versioning

Branching in Git – A real world example

- Merge conflicts occur regularly
- It is highlighted in the file directly



```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>>iss53:index.html
```

- This means the version in HEAD (your master branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the ========), while the version in your iss53
- branch looks like everything in the bottom part. To resolve the conflict, you must either choose one side or the other or merge the contents yourself
- You can use a tool for it, called difftool

```
git difftool
```

Git on platforms such as GitLab or GitHub

- To merge a branch, typically a so-called pull request (PR) is created

- Workflow:
  - Developer creates a new branch and makes changes
  - Developer pushes the branch to the remote repository
  - Opens a Pull Request to request merging their branch into the target branch

- Team members can:
  - Review code
  - Discuss changes
  - Request modifications
  - Approve and merge

Break

Pracital demo | Getting ready

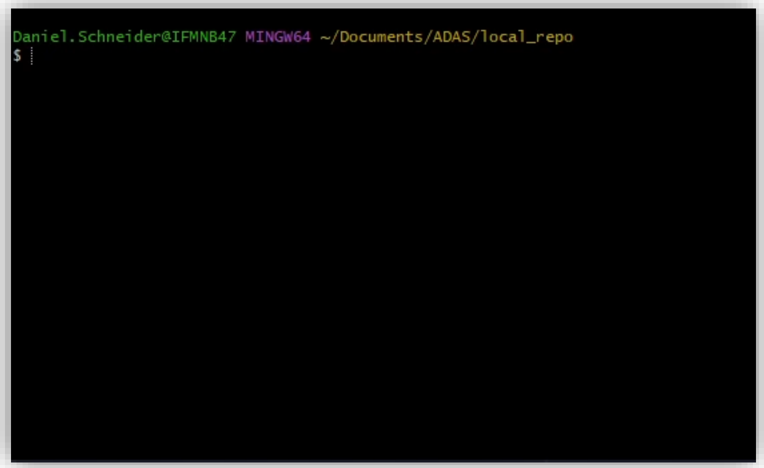
- Install Visual Studio code on your system
  - https://code.visualstudio.com/download
  - Alternative google colab

- Install Git on your system
  - https://git-scm.com/downloads

- Using VScode extensions to install
  - Python 3.1X
  - Gitlens
  - Projectmanager

Pracital demo | Git & Python basics

- Create a local VCS
  - git init my-demo-repo
- Create content
  - echo "Hello, world!" > hello.txt
- Add file to version control
  - git add hello.txt
- Commit the changes
  - git commit -m "Initial commit“
- Change content of the file
  - echo "This is a change." >> hello.txt
- Diff the changes
  - git diff



```
Daniel.Schneider@IFMNB47 MINGW64 ~/Documents/ADAS/local_repo
$
```

Pracital demo | Git & Python basics

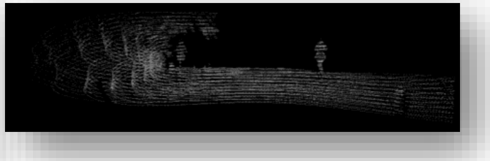
- Clone repository from: https://github.com/schneider-daniel/SEM25-python

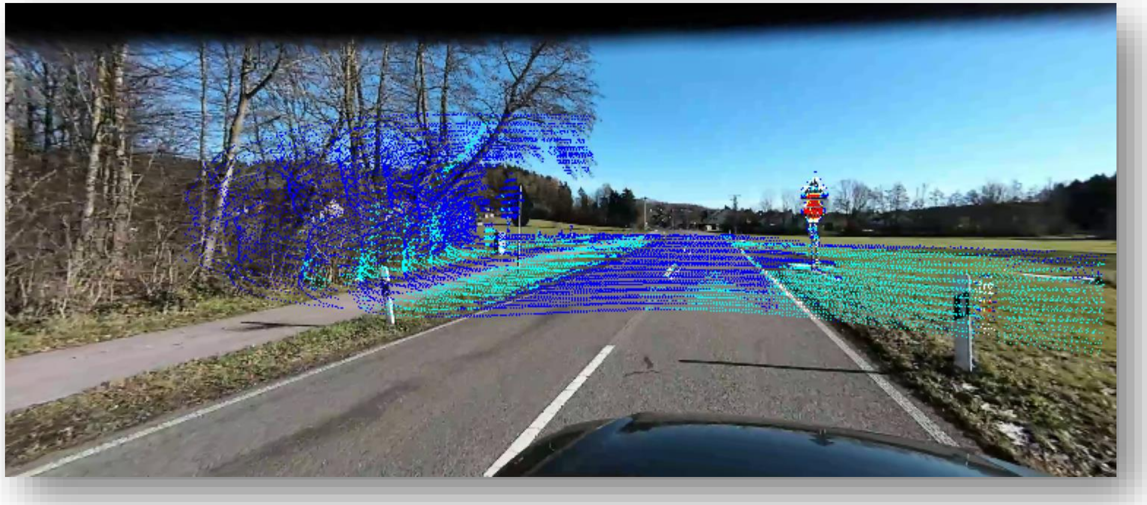
- Set up VScode and create a virtual environment (we will do this together)

- Run:
  - 0_notebook_basics.ipynb
  - 1_data_types.ipynb
  - 2_libraries.ipynb

Pracital demo | Real world example

- Find 2-3 groups
  - One group keeps an eye on importing the provided image
  - Second group implements a function to import a point cloud
- Implement the following functions:
  - read_image()
  - read_pcd()
- Use software versioning and branches
- Create pull requests
- Code skeleton is given





IFM

Institut für Fahrerassistenz
und vernetzte Mobilität

Hochschule
Kempten
University of Applied Sciences

www.hs-kempten.de/ifm