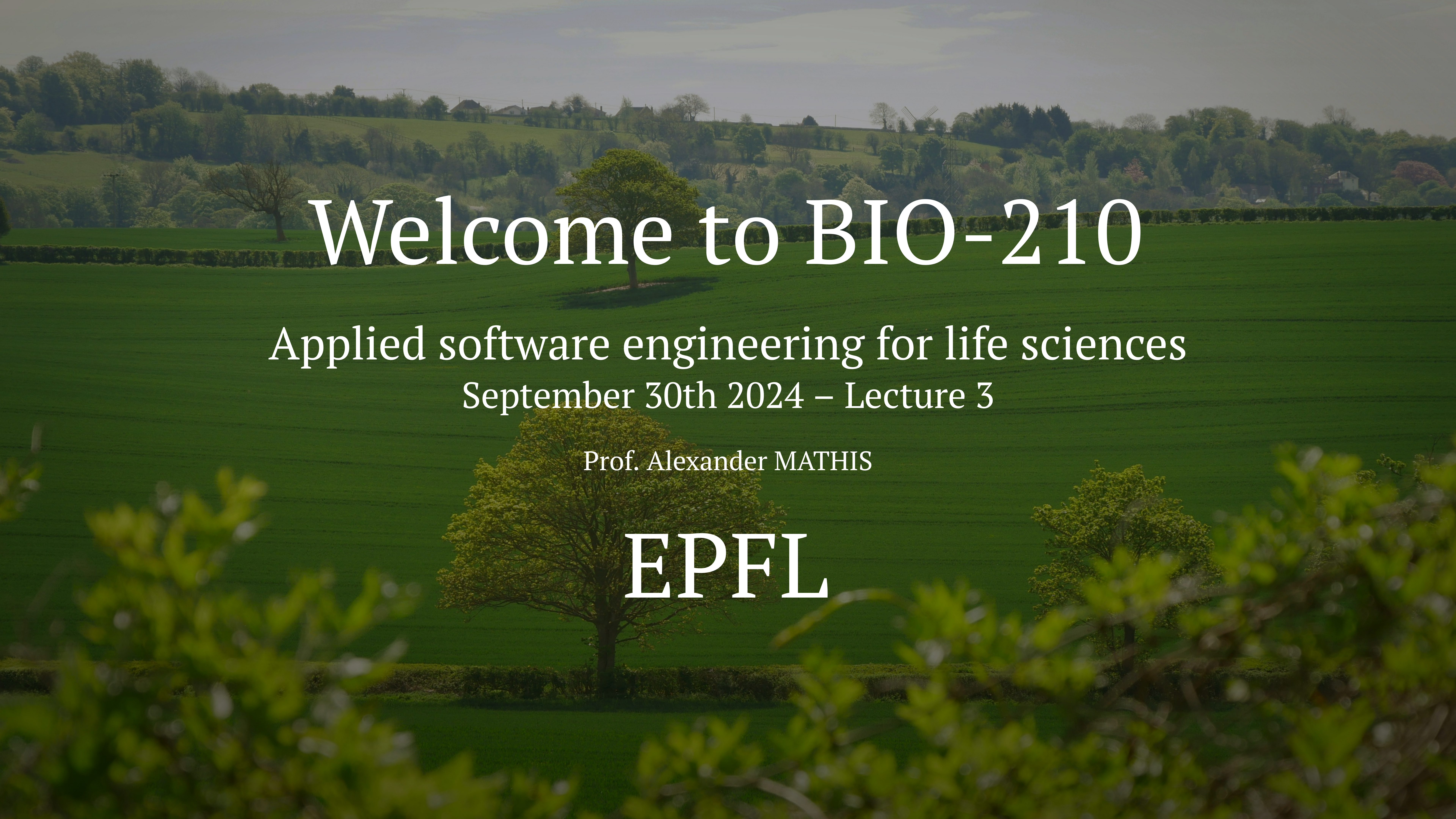


Welcome to BIO-210

Applied software engineering for life sciences

September 30th 2024 – Lecture 3

Prof. Alexander MATHIS

EPFL

Announcements

- As always solutions can be found on our GitHub
- Please install Python, git and Visual Studio Code *before* the exercise session. See guide here
- Today the first quiz, it starts at 3pm on Moodle; you have time until Friday (at midnight) to fill it out!
- Monday 15:15 - 16: my office hours at SV 2811

	Date	Topic	Software version	Software releases	Grading / Feedback
0	09/09/2024	Python introduction I			
1	16/09/2024	Public holiday			
2	23/09/2024	Python introduction II			
3	30/09/2024	Git and GitHub (+installation VS Code)			
4	07/10/2024	Project introduction	v1		
5	14/10/2024	Functionify	v2	v1	
6	21/10/2024	EPFL fall break			
7	28/10/2024	Visualization and documentation	v3	v2	code review (API)
8	04/11/2024	Unit-tests, functional tests	v4	v3	
9	11/11/2024	Code refactoring	v5	v4	graded (tests)
10	18/11/2024	Profiling and code optimization	v6	v5	code review
11	25/11/2024	Object oriented programming	v7	v6	graded (speed)
12	02/12/2024	Model analysis and project report	v8	v7	code review (OO)
13	09/12/2024	Work on project			
14	16/12/2024	Wrap up		v8	graded (project)

Functions

We have already used many functions (`` `np.sin` ``, `` `range` ``,…). But how are custom functions defined?

A simple example

```
>>> def f(x):             # Notice: keyword def, colon
        return x          # return is not required in python (then will return None)
>>> type(f)
function
>>> f(1)
1                         # Our function just returns whatever you pass
>>> f('abc')
'abc'
```

A custom module

A module is a file containing Python definitions and statements.

The file name is the module name plus the suffix `` `.py` ``.

```python
# Heaviside function module

def HeavisideFun(x):
    ''' Heaviside function with half-maximum convention '''
    if x>0:
       return 1.
    elif x<0:
       return 0.
    elif x==0:
       return .5
    else:
       return None

```

Let’s save this file as `` `mymodule.py` ``

Example use of our custom module

```
In [1]: import mymodule                     # Import our module
In [2]: mymodule.HeavisideFun(13.)
Out[2]: 1.0
In [3]: mymodule.HeavisideFun(0)
Out[3]: 0.5
In [4]: f=mymodule.HeavisideFun
In [5]: f(13.)                            # one can assign a local name.
Out [5]: 1.0
In [6]: mymodule.HeavisideFun?
Signature: mymodule.HeavisideFun(x)
Docstring: Heaviside function with half-maximum convention
File:      ~/Code/Teaching/BIO-210-materials/slides/mymodule.py
Type:      function
```

Remember, we have already seen built-in modules (e.g., `` `math` ``) and packages, which are structured modules (e.g., `` `numpy` ``).

Note, here we assume that we are in the same folder as mymodule.py. Learn more about where python looks for modules here.

Quiz: What will be printed?

```python
print(hex(254))
print(hex(0b11111110))
print(hex(0Xfe))
```

The code returns hexadecimal representations of numbers!

```python
print(hex(254))             # decimal  input decimal  254
print(hex(0b11111110))   # binary input = decimal  254
print(hex(0Xfe))            # hexadecimal  input = decimal  254
```

It prints 3 x `0xfe`, standing for the hexadecimal numbers (=base 16: represented by 0-9 and a-f).

Hash Functions

- A hash function is any function that can be used to map data of arbitrary size to fixed-size values
- hashlib gives access to many different hash algorithms in python
- SHA-1 (Secure Hash Algorithm 1) is a hash function which takes an (arbitrary byte) input and produces a 160-bit hash value – typically rendered as a hexadecimal number , and thus 40 digits long.

Example:

```
>>> import hashlib                    # importing a module
>>> m = hashlib.sha1()                # accessing a function in this module
>>> m.update(b"Some text")
>>> m.update(b" and some more...")
>>> print(m.hexdigest())
ca1933af76523fa98a5961ef6e0cf65c64866ef4
>>> print(len(m.hexdigest()))
40
>>> m.update(b"1")    # let's add a single character!
>>> print(m.hexdigest())
278c021f92e4967e230f4c39303f8dd1fd78a169 # notice the hash looks completely different!
```

Questions?

Motivational questions:

1. How do you keep track of files?
2. Who has used version control systems?

Why version control?

Version control systems (VCS) record changes to source code, or more general a file or set of files (arranged in a directory) over time so that you can recall specific versions later.

Thus, it allows you to

- keep track of projects
- collaborate with others

What do VCS allow you to do?

To (easily) answer the questions like:

- Who programmed this module/feature?
- When was this particular line of this particular piece of code edited? By whom? Why was it edited?
- Over the last 229 revisions, when/why did a particular functionality stop working?
- …

What is git?

- the most commonly used version control system, >85% (others: subversion, mercurial)
- it is a distributed system and was started by Linus Torvalds

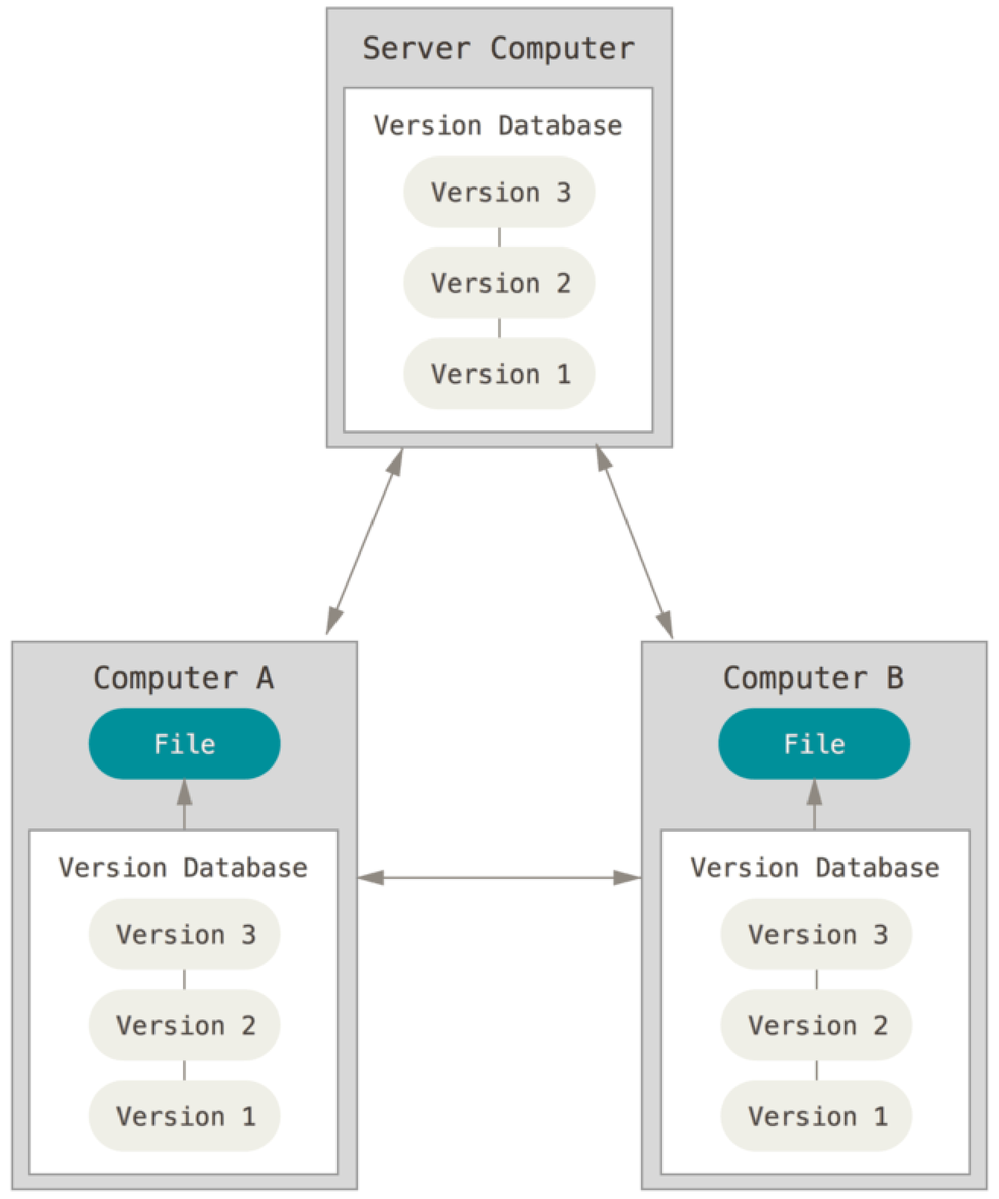


image: git-book

Git's reputation…

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Source

Output of `man git`

```
GIT(1)                                                 Git Manual

NAME
       git - the stupid content tracker

SYNOPSIS
       git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|-P|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>] [--config-env <name>=<envvar>]
           <command> [<args>]

DESCRIPTION
       Git is a fast, scalable, distributed revision control system with an unusually rich cor
       provides both high-level operations and full access to internals.

       See gittutorial(7) to get started, then see giteveryday(7) for a useful minimum set of
       Git User's Manual[1] has a more in-depth introduction.

       After you mastered the basic concepts, you can come back to this page to learn what cor
       offers. You can learn more about individual Git commands with "git help command". gitcl
       page gives you an overview of the command-line command syntax.
```

Questions?

Our strategy for diving into git

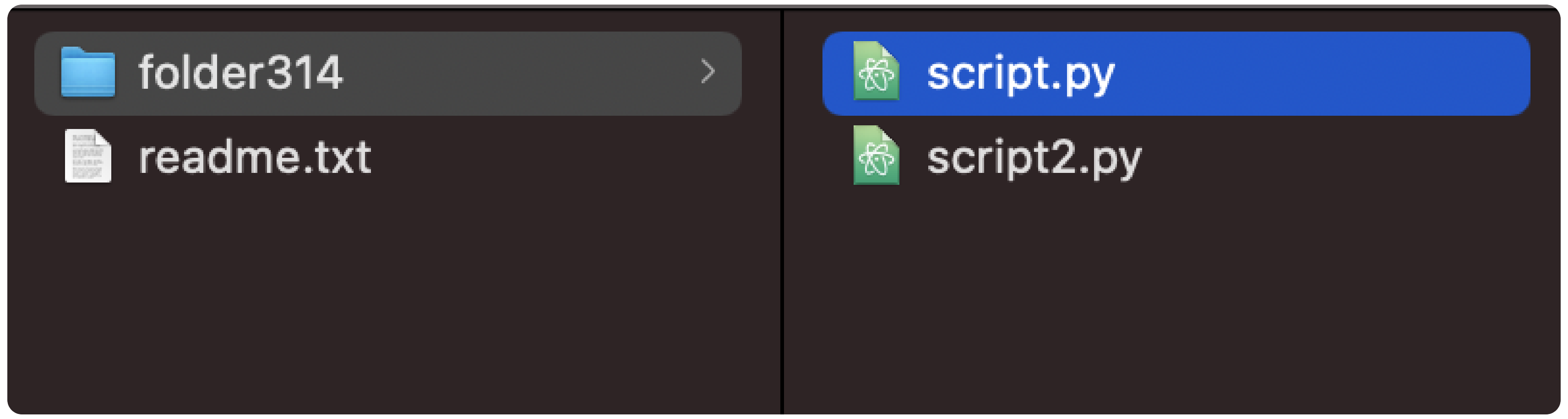
- learn about it’s data model
- then learn key commands and what they do
- over the course of the semester we will cover more aspects of git & you will use it in practice for your projects

Sources:

- my lecture is partially based on Anish Athalye’s excellent lecture on git at MIT
- Check out the comprehensive git-book

Let’s consider an example directory

Visual representation (in finder on a mac):



It contains 1 folder (directory), which contains 2 Python scripts, as well as 1 file (readme.txt)

Git’s data model: Snapshots

Folders with files and (potentially) with subfolders are stored as snapshots.

- a file is called a "binary large object" (blob)
- a directory (folder) is called "tree"

For instance, for our example folder:

```
<root> (tree)
|
+- folder314 (tree)
|  |
|  + script.py (blob, contents = "print('hello world')")
|  + script2.py (blob, contents = "print('hello BIO-210')")
|
+- readme.txt (blob, contents = "Collection of scripts")
```

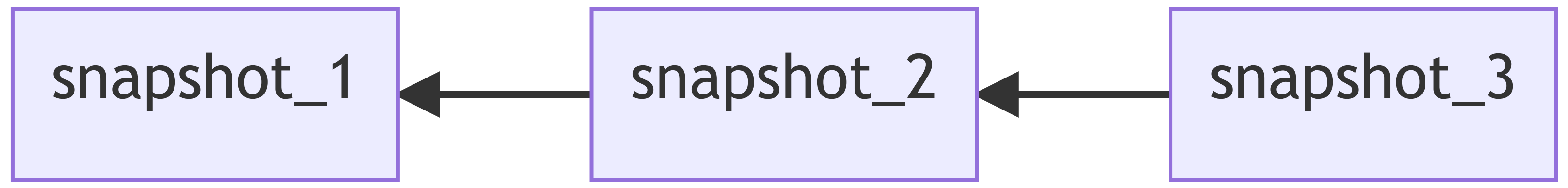
Here a tree contains the tree `folder314` with two blobs (`script.py` and `script2.py`) as well as a blob (`readme.txt`).

Blobs and trees in pseudo code

```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains directories and/or named files
type tree = map<string, tree | blob>
```

How are snapshots related?

Consider three snapshots. In the simplest case, the snapshots are serially developed (dependent on each other)

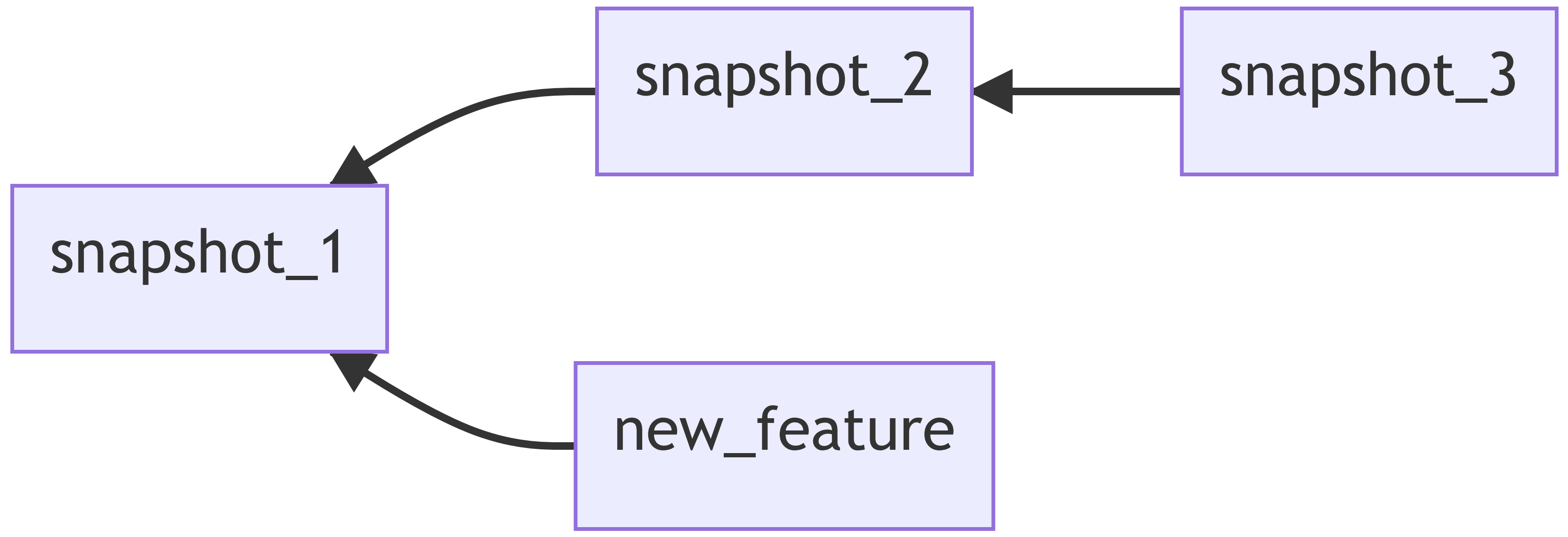


Here 1 is the parent of 2 and 2 the parent of 3.

Snapshots are called **commits** in git!

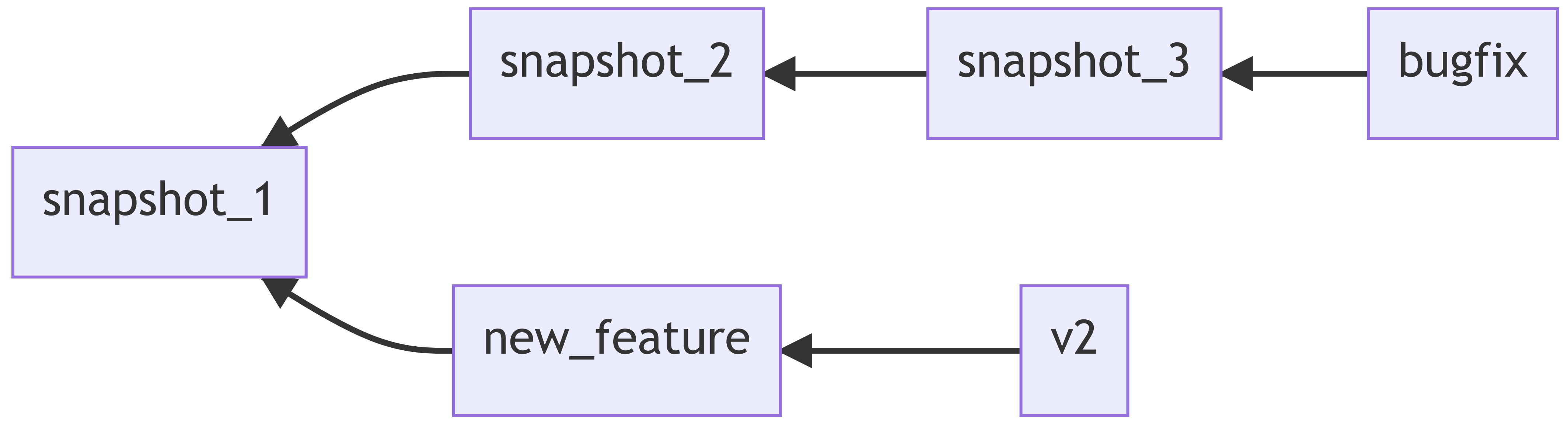
How are commits related?

One can develop features in parallel, thus snapshots can give rise to multiple other snapshots (branches)



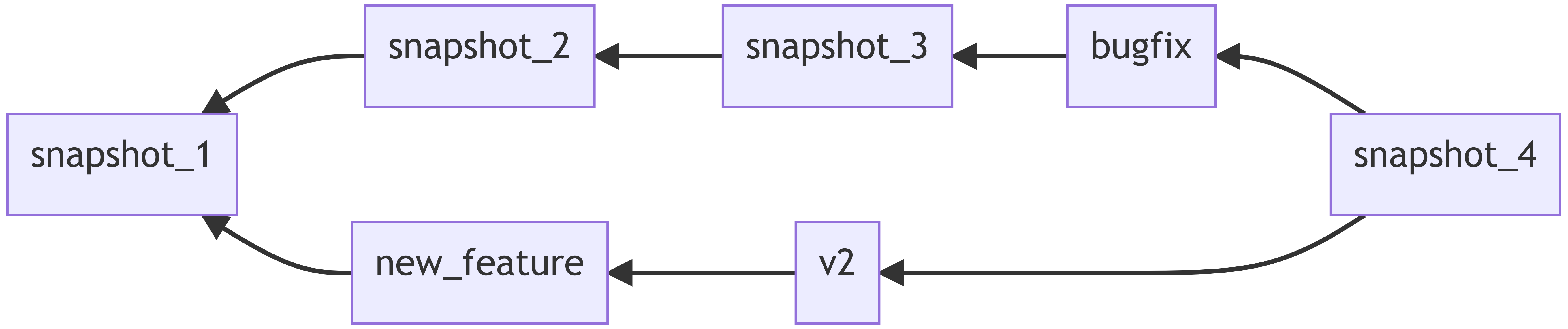
How are commits related?

Further developing the new feature and fixing a bug in parallel



How are commits related?

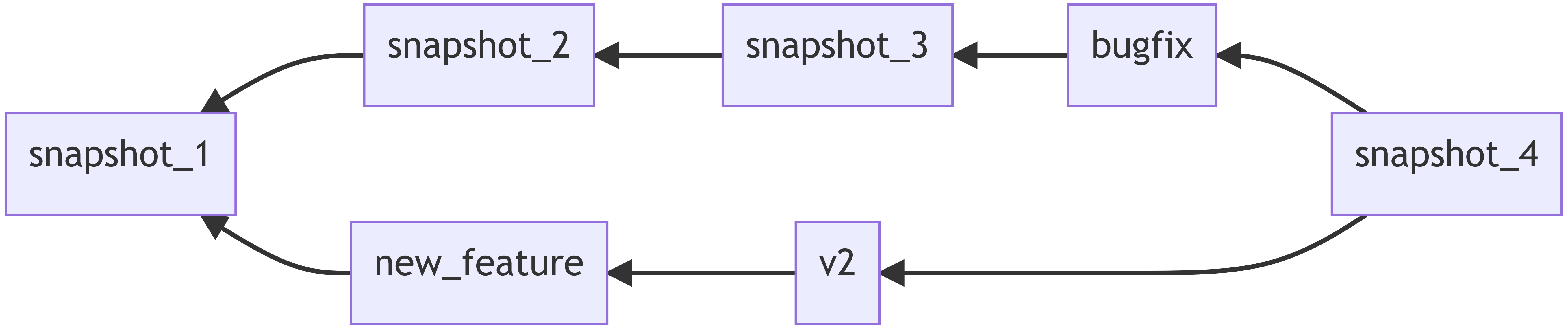
Another common scenario is to create a version that combines two (or multiple other versions). For instance, in our example one might want to create a version that contains the bugfix and the new feature. This is called `merging`



Each commit has (at least one) parent (commit). Merges can have multiple.

Note that merging can sometimes not be done automatically (by git), but might need input from the programmer. This case is called merge conflict (see later).

Commits from a directed acyclic graph (DAG)



To flexibly allow all those possiblities, commits are related like nodes in **directed acyclic graphs** in git!

Git’s data model in pseudo code

Git’s simple model of history (and it’s contents).

```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains named files and directories
type tree = map<string, tree | blob>

// a commit has parents, metadata, and the top-level tree
type commit = struct {
    parents: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

Objects and content-addressing

On object is a blob, tree or commit.

```
type object = blob | tree | commit
```

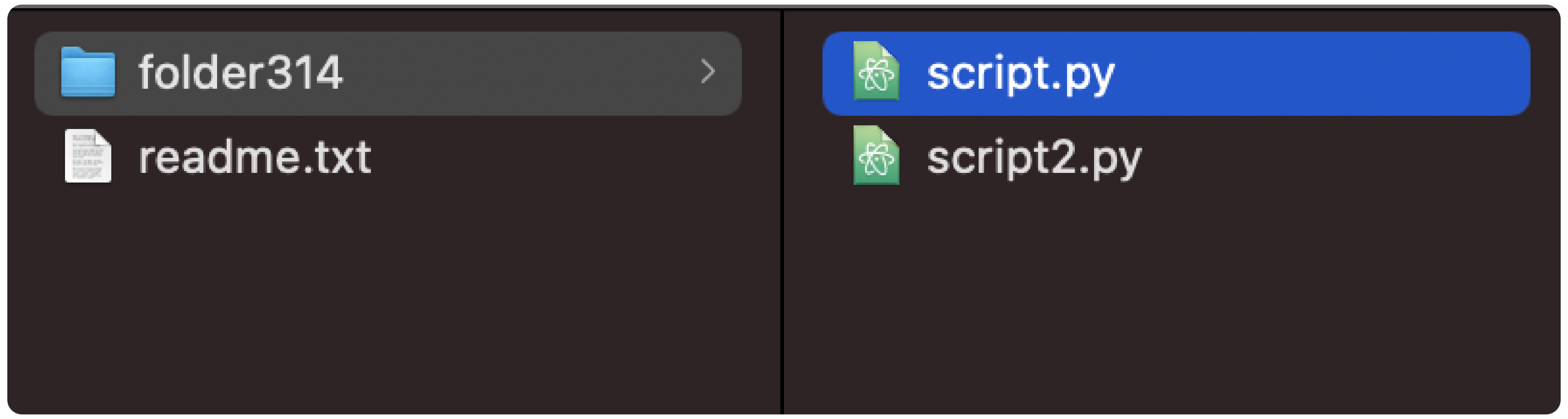
In Git’s data store, all objects are content-addressed by their SHA-1 hash.

```
objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
```

Thus, git has integrity – you can’t lose information in transit or get file corruption without git being able to detect it as the id is based on the content!

Looking up content for our example folder

Visual representation (in finder on a mac):



Looking up content (we will later see how we know the ID, for now let's assume this folder is a git repository)

```
alex@mac git_demo % git cat-file -p bcb0914d46780824038f469a15d4d613a6143d17
040000 tree 4b2451c9990d8b6cb3978bc413f7681a787a3209    folder314
100644 blob ef79fe5c6f74de0d99f90601794d96c9c40f1578    readme.txt
```

Diving deeper into the tree

```
alex@mac git_demo % git cat-file -p bcb0914d46780824038f469a15d4d613a6143d17
040000 tree 4b2451c9990d8b6cb3978bc413f7681a787a3209    folder314
100644 blob ef79fe5c6f74de0d99f90601794d96c9c40f1578    readme.txt
alex@mac git_demo % git cat-file -p 4b2451c9990d8b6cb3978bc413f7681a787a3209
100644 blob 75d9766db981cf4e8c59be50ff01e574581d43fc    script.py
100644 blob 9a7f6fc3ea99e6fbb8134a0329a3b517f5d6c827    script2.py
alex@mac git_demo % git cat-file -p 9a7f6fc3ea99e6fbb8134a0329a3b517f5d6c827
print('hello BIO-210')
alex@mac git_demo % git cat-file -p ef79fe
Collection of scripts
# So far we checked blobs and trees....
# Looking up a commit object!
alex@mac git_demo % git cat-file -p 53e1d98078b0499f4a0a5e9874633e478415ef4b
tree bcb0914d46780824038f469a15d4d613a6143d17
author AlexEMG <alexander@deeplabcut.org> 1634469842 +0200
committer AlexEMG <alexander@deeplabcut.org> 1634469842 +0200

Example folder
```

Notice: you don't need to give the full hexadecimal ID!

References

Having unique commit identifiers via SHA-1 hashes is great (data integrity and unique), but inconvenient for users (try remembering lots of 40 hexadecimal characters).

Git provides references. References are pointers to commits. Unlike objects, which are immutable, references are mutable.

```
references = map<string, string>

def update_reference(name, id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

Check out: https://git-scm.com/book/en/v2/Git-Internals-Git-References

Important references:

- *master/main* reference usually points to the latest commit in the main branch of development.
- *master* is the old name and is being phased out, but you will find it in older projects
- The special reference *HEAD* refers to where we currently are in the history
- *HEAD* also defines the state of the working directory. The working directory's state is the tree of the *HEAD*'s snapshot
- commits are relative to the *HEAD* (i.e. where you are, which provides the parents file)

Git repositories

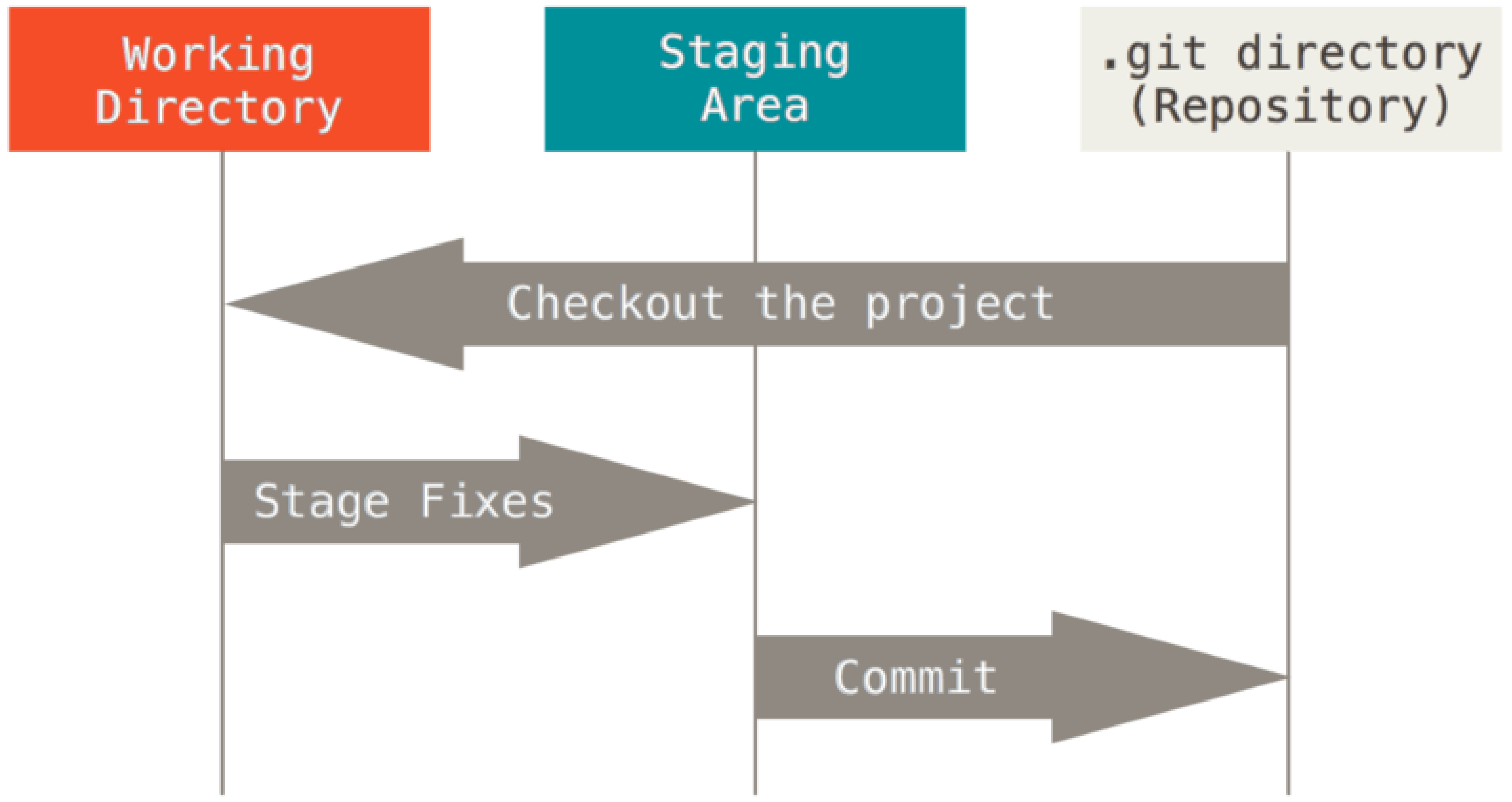
- git stores objects and references on disk
- git thinks about its data more like a stream of snapshots (commits); to be efficient, if files have not changed, git does not store the file again, just a link to the previous identical file it has already stored.
- Git commands alter the commit DAG by adding objects and adding/updating references

Staging area

In order to be part of a commit, blobs and trees need to be staged (added).

This allows to control which new files should be committed in what order!

Visual illustration of staging and committing



Fundamentally, staging gives you control to decide what should be commited from your working directory.

Source: git docs

Questions?

Quiz: What are?

- man
- echo
- ls
- mkdir
- cd
- cat

Useful bash (unix shell) commands we will use later

man	look up the manual (e.g., `` `man man` ``, `` `man echo` ``)
echo	write arguments to standard output
ls	list directory contents
mkdir	make a directory
cd	change directory
cat	concatenate and print files

Basic git commands

For more details, see git’s reference manual.

- `` `git help <command>` ``: get help for a git command
- `` `git init` ``: creates a new git repo, with data stored in the .git directory
- `` `git status` ``: tells you what’s going on
- `` `git add <filename>` ``: adds files to staging area
- `` `git commit` ``: creates a new commit
- `` `git log` ``: shows a (flattened) log of history
- `` `git log --all --graph --decorate` ``: visualizes history as a DAG
- `` `git diff <filename>` ``: show changes you made relative to the staging area
- `` `git diff <revision> <filename>` ``: shows differences in a file between snapshots
- `` `git checkout <revision>` ``: switch branches or restore working tree files

Let’s use them. We will step-by-step create a *demo_repo* folder.

Initializing a git repository

Note, we run this in a bash/zsh (shell). For windows see here and in particular with git: git BASH.

```
alex@mac Code % mkdir demo-repo
alex@mac Code % cd demo-repo
alex@mac demo-repo % ls
alex@mac demo-repo % git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/alex/Code/demo-repo/.git/
alex@mac demo-repo % ls - a
.git
```

Note: Hidden folder .git is created when we initialize the repository (try it out!)

Initializing a git repository

Note, we run this in a bash/zsh (shell). For windows see here and in particular with git: git BASH.

```
alex@mac Code % mkdir demo-repo
alex@mac Code % cd demo-repo
alex@mac demo-repo % ls
alex@mac demo-repo % git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/alex/Code/demo-repo/.git/
alex@mac demo-repo % ls - a
.git
```

Note: Hidden folder .git is created when we initialize the repository (try it out!)

Initializing a git repository

Note, we run this in a bash/zsh (shell). For windows see here and in particular with git: git BASH.

```
alex@mac Code % mkdir demo-repo
alex@mac Code % cd demo-repo
alex@mac demo-repo % ls
alex@mac demo-repo % git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/alex/Code/demo-repo/.git/
alex@mac demo-repo % ls - a
.git
```

Note: Hidden folder .git is created when we initialize the repository (try it out!)

Creating content (in demo-repo)

```
alex@mac demo-repo % echo "print('Hello world!')" > script.py
alex@mac demo-repo % ls
script.py
alex@mac demo-repo % cat script.py
print('Hello world!')
alex@mac demo-repo % python3 script.py
Hello world!
```

Creating content (in demo-repo)

```
alex@mac demo-repo % echo "print('Hello world!')" > script.py
alex@mac demo-repo % ls
script.py
alex@mac demo-repo % cat script.py
print('Hello world!')
alex@mac demo-repo % python3 script.py
Hello world!
```

Creating content (in demo-repo)

```
alex@mac demo-repo % echo "print('Hello world!')" > script.py
alex@mac demo-repo % ls
script.py
alex@mac demo-repo % cat script.py
print('Hello world!')
alex@mac demo-repo % python3 script.py
Hello world!
```

Tracking and staging content with git

```
alex@mac demo-repo % git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
alex@mac demo-repo % git add script.py      # staging!
alex@mac demo-repo % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   script.py

alex@mac demo-repo %
```

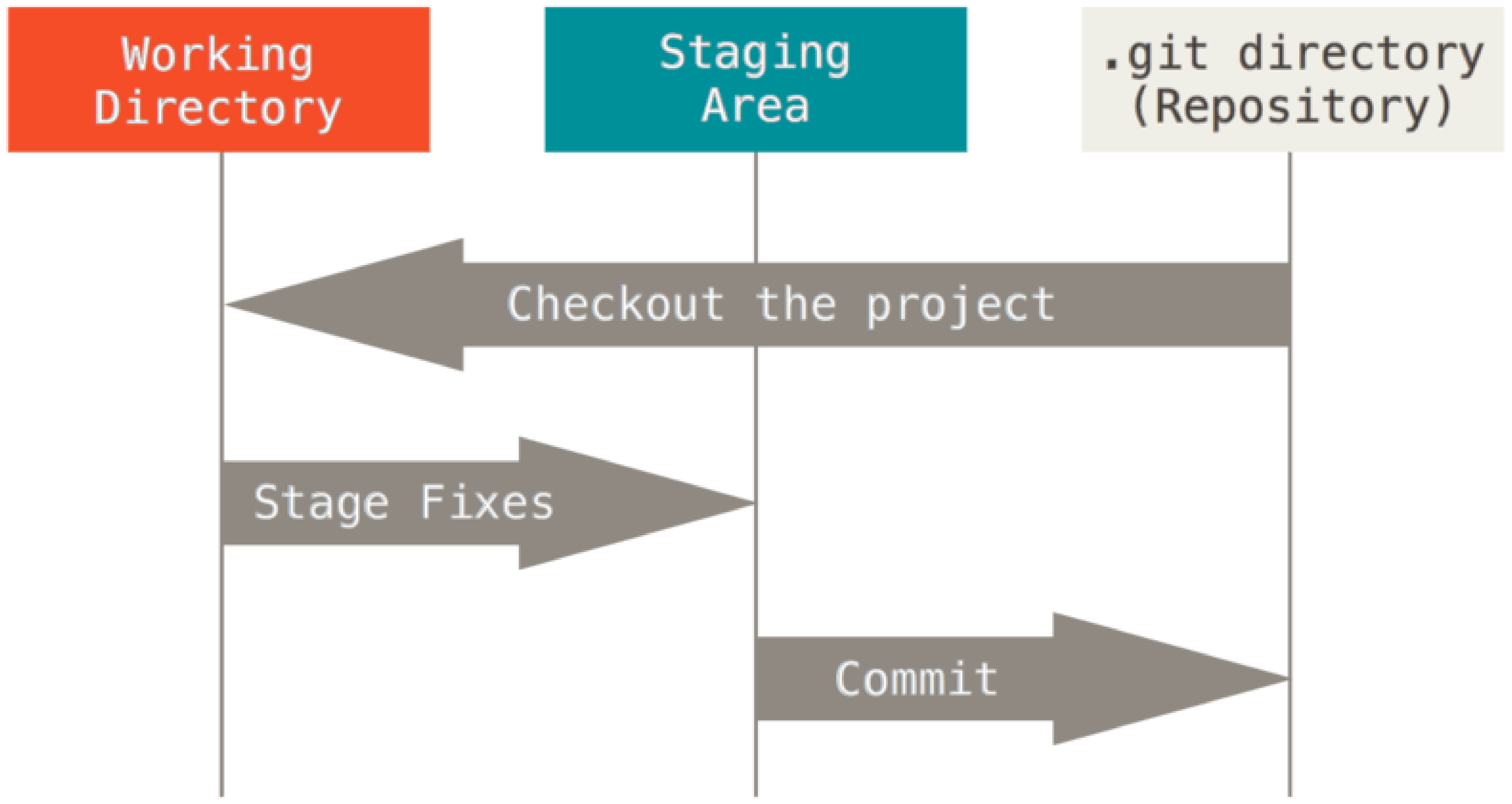
Tracking and staging content with git

```
alex@mac demo-repo % git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
alex@mac demo-repo % git add script.py      # staging!
alex@mac demo-repo % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   script.py

alex@mac demo-repo %
```

Our first commit

```
alex@mac demo-repo % git commit -m "Adding script"
[master (root-commit) 5be7f54] Adding script
 1 file changed, 1 insertion(+)
 create mode 100644 script.py
alex@mac demo-repo % git status
On branch master
nothing to commit, working tree clean
alex@mac demo-repo % git log
commit 5be7f545770e2a42fc8831102f1667aad5d1228c (HEAD -> master)
Author: AlexEMG <alexander@deeplabcut.org>
Date:   Sun Sep 29 19:18:47 2024 +0200

    Adding script
```

Visual illustration of staging and committing



Fundamentally, staging gives you control to decide what should be commited from your working directory.

Source: git docs

Quiz: I changed *script.py*, what did I change?

```
alex@mac demo-repo % python3 script.py
Hello BIO-210 class
```

Changing content and noticing it!

Notice that `git diff` notices the changes automatically!

```
alex@mac demo-repo % python3 script.py
Hello BIO-210 class
alex@mac demo-repo % git diff script.py
diff --git a/script.py b/script.py
index 60f08aa..bddabf9 100644
--- a/script.py
+++ b/script.py
@@ -1 +1 @@
-print('Hello world!')
+print('Hello BIO-210 class')
alex@mac demo-repo % git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
	modified:   script.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Staging the changes and committing

```
alex@mac demo-repo % git add .
alex@mac demo-repo % git commit -m "new greeting"
[master 6baa498] new greeting
 1 file changed, 1 insertion(+), 1 deletion(-)
alex@mac demo-repo % git log
commit 6baa498d512d23ce81a30ac8cbc1547ce2069eec (HEAD -> master)
Author: AlexEMG <alexander@deeplabcut.org>
Date:   Sun Sep 29 19:21:58 2024 +0200

    new greeting

commit 5be7f545770e2a42fc8831102f1667aad5d1228c
Author: AlexEMG <alexander@deeplabcut.org>
Date:   Sun Sep 29 19:18:47 2024 +0200

    Adding script

```

Questions?

Quiz: What does this program do?

```python
# Simple greeting program

def greeting(context=None):
        prefix='Hello'
        if context == 'teaching':
                suffix="BIO-210!"
        else:
                suffix = "World!"

        print(prefix,suffix)

if __name__ == "__main__":
    import sys
    greeting(sys.argv[1] if len(sys.argv)>1 else None)
```

Module and script use!

When a python module is imported, `__name__` is set to the module's name!

```
# Simple greeting program

def greeting(context=None):          # Custom function that
        prefix='Hello'
        if context == 'teaching':
                suffix="BIO-210!"
        else:
                suffix = "World!"

        print(prefix,suffix)

# Execute when the module is not initialized from an import statement.
if __name__ == "__main__":
    import sys                # sys library: https://docs.python.org/3/library/sys.html
    # sys.argv -- returns list of command line arguments passed to a Python script.
    # sys.argv[0] is the script name
    greeting(sys.argv[1] if len(sys.argv)>1 else None)
```

Used as a module

Let’s assume we call this program `script.py` and are in this folder!

```
>>> import script     # Nothing is printed!
>>> script.greeting()
'Hello World!'
>>> script.greeting('teaching')
'Hello BIO-210!''
>>> print(script.__name__)
'script'
```

Used as a script

Let's update `script.py` in our repository accordingly!

```
alex@mac demo-repo % python script.py
Hello World!
alex@mac demo-repo % python script.py teaching
Hello BIO-210!
alex@mac demo-repo % git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
	modified:   script.py

no changes added to commit (use "git add" and/or "git commit -a")
alex@mac demo-repo % git add .
alex@mac demo-repo % git commit -m "Greeting program expanded"
[master f7d2e70] Greeting program expanded
 1 file changed, 14 insertions(+), 1 deletion(-)
alex@mac demo-repo % git log --all --graph --decorate --oneline
* f7d2e70 (HEAD -> master) Greeting program expanded
* 6baa498 new greeting
* 5be7f54 Adding script
```

Questions?

git init / log / status / add / commit?

Git branching and merging

- `git branch`: shows branches
- `git branch <name>`: creates a branch
- `git checkout -b <name>`: creates a branch and switches to it. This is the same as `git branch <name>`; `git checkout <name>`
- `git merge <revision>`: merges the commit `<revision>` into current branch

Creating a branch and adding a new context

```
alex@mac demo-repo % git branch feature/new_context
alex@mac demo-repo % git checkout feature/new_context
Switched to branch 'feature/new_context'
alex@mac demo-repo % vim script.py       # see below!
alex@mac demo-repo % python3 script.py research
Hello Lucas
```

```python
# Simple greeting program
def greeting(context=None):
        prefix='Hello'
        if context == 'teaching':
                suffix="BIO-210!"
        elif context == 'research':
                suffix="Lucas"
        else:
                suffix = "World!"

        print(prefix,suffix)

if __name__ == "__main__":
    import sys
    greeting(sys.argv[1] if len(sys.argv)>1 else None)
```

Let’s commit to the new branch

```
alex@mac demo-repo % git add .
alex@mac demo-repo % git commit -m "Greeting Lucas"
[feature/new_context a7ed742] Greeting Lucas
 1 file changed, 2 insertions(+)
alex@mac demo-repo % git log --all --graph --decorate --oneline
* a7ed742 (HEAD -> feature/new_context) Greeting Lucas
* f7d2e70 (master) Greeting program expanded
* 6baa498 new greeting
* 5be7f54 Adding script
```

Quiz: What is happening here?

We are still on branch: 'feature/new_context'.

```
alex@mac demo-repo % python3 script.py research
Hello Lucas
alex@mac demo-repo % git checkout master
Switched to branch 'master'
alex@mac demo-repo % python3 script.py research
Hello World!
```

Checkout also changes the working directory files!

```
alex@mac demo-repo % python3 script.py research
Hello Lucas
alex@mac demo-repo % git checkout master
Switched to branch 'master'
# NOTE: RESEARCH FUNCTIONALITY MISSING on master
alex@mac demo-repo % python3 script.py research
Hello World!
```

Create a new feature on master (in parallel)

(we are still on master and updated script.py)

```
alex@mac demo-repo % git diff script.py
diff --git a/script.py b/script.py
index 6fc7272..ce5355c 100644
--- a/script.py
+++ b/script.py
@@ -4,6 +4,8 @@ def greeting(context=None):
         prefix='Hello'
         if context == 'teaching':
                 suffix="BIO-210!"
+        elif context == 'research':
+                suffix="Alberto"
         else:
                 suffix = "World!"
alex@mac demo-repo % git add .
alex@mac demo-repo % git commit -m "Greeting Alberto"
[master 799c9ac] Greeting Alberto
1 file changed, 2 insertions(+)
```

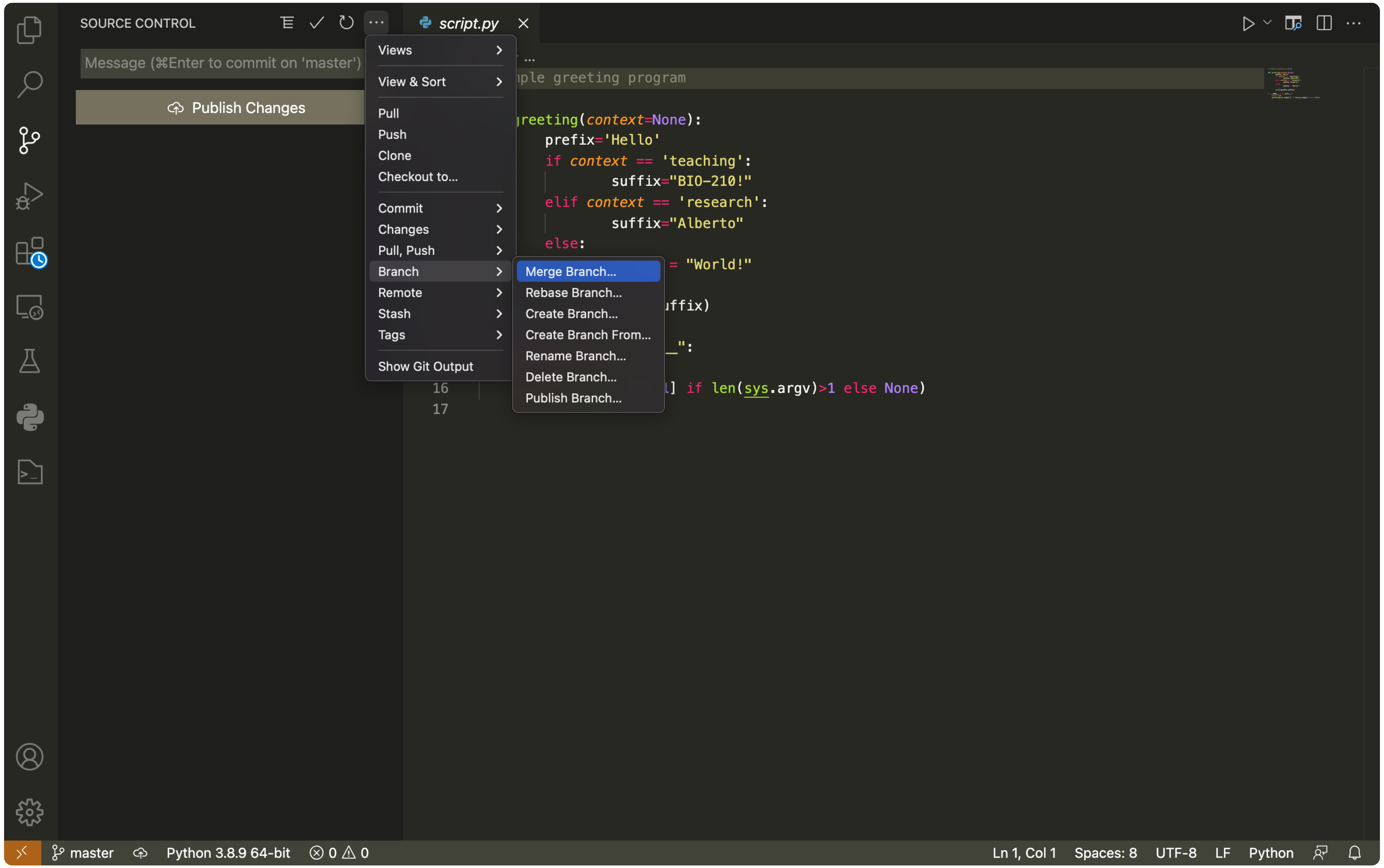
Our DAG is becoming more interesting

```
alex@mac demo-repo % git log --all --graph --decorate --oneline
* 799c9ac (HEAD -> master) Greeting Alberto
| * a7ed742 (feature/new_context) Greeting Lucas
|/
* f7d2e70 Greeting program expanded
* 6baa498 new greeting
* 5be7f54 Adding script
alex@mac demo-repo % python3 script.py research
Hello Alberto
alex@mac demo-repo % git checkout feature/new_context
Switched to branch 'feature/new_context'
alex@mac demo-repo % python3 script.py research
Hello Lucas
alex@mac demo-repo % python3 script.py
Hello World!
```

Let’s merge …

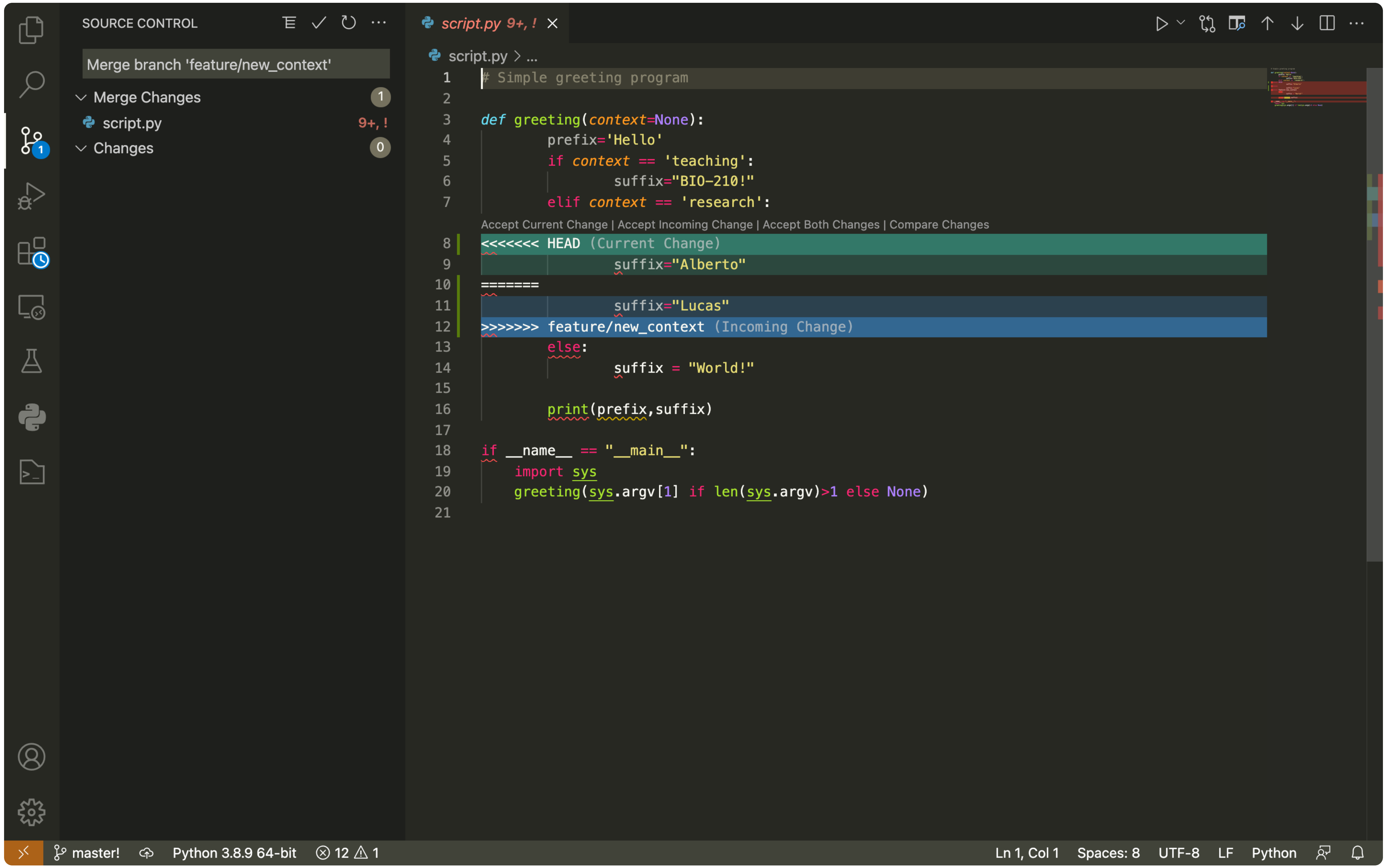
```
alex@mac demo-repo % git merge feature/new_context
Auto-merging script.py
CONFLICT (content): Merge conflict in script.py
Automatic merge failed; fix conflicts and then commit the result.
alex@mac demo-repo % vim script.py # Fixed manually (but see next slide!)
alex@mac demo-repo % git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)
alex@mac demo-repo % git commit -m "Conflicts fixed and greeting Alberto"
[master b0337d2] Conflicts fixed and greeting Alberto
alex@mac demo-repo % git log --all --graph --decorate --oneline
*   b0337d2 (HEAD -> master) Conflicts fixed and greeting Alberto
|\
| * a7ed742 (feature/new_context) Greeting Lucas
* | 799c9ac Greeting Alberto
|/
* f7d2e70 Greeting program expanded
* 6baa498 new greeting
* 5be7f54 Adding script
```

Visual support for git in Visual Studio Code



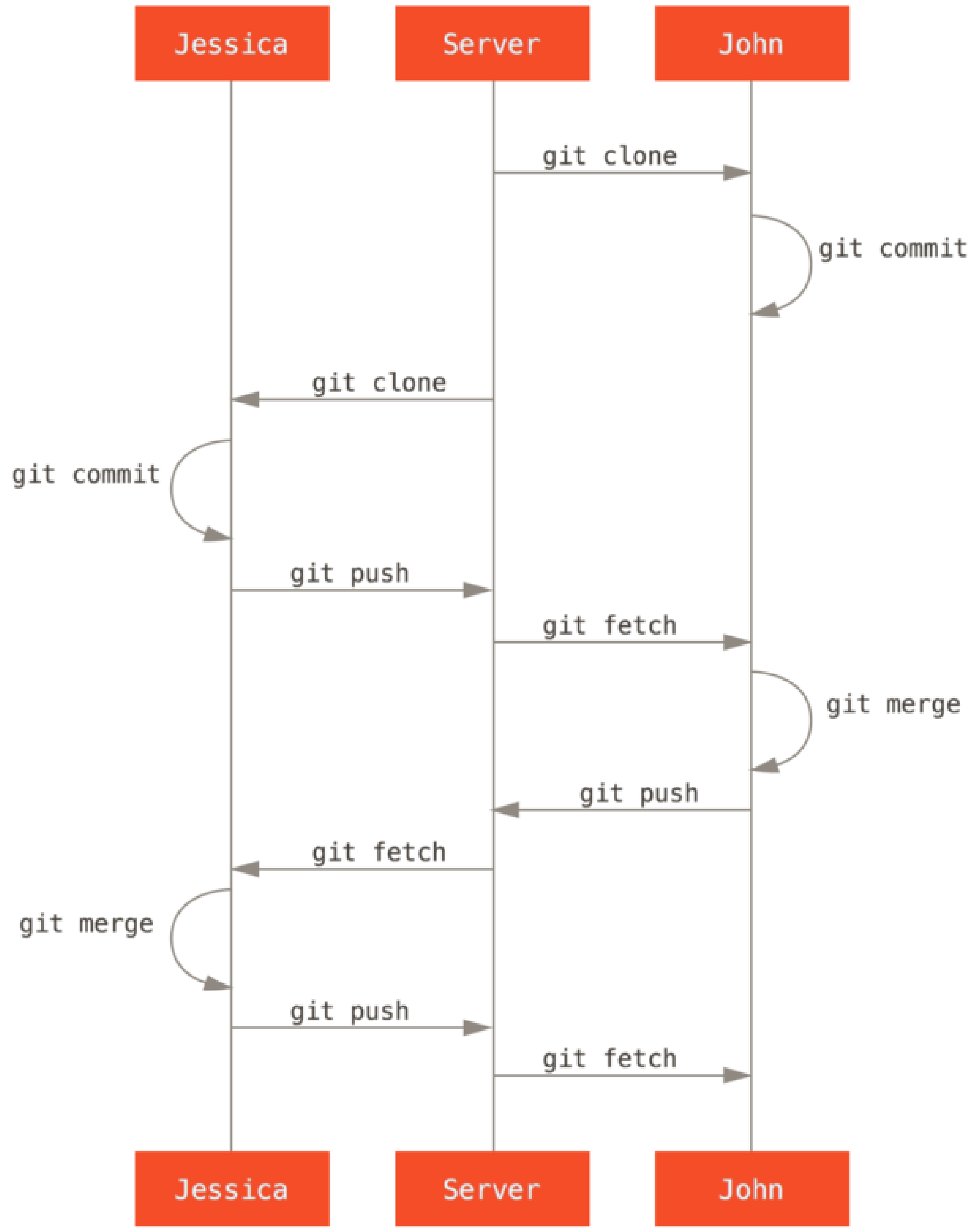
```
ple greeting program

reeting(context=None):
    prefix='Hello'
    if context == 'teaching':
            suffix="BIO-210!"
    elif context == 'research':
            suffix="Alberto"
    else:
                  = "World!"

                  uffix)

                  _":

                  ] if len(sys.argv)>1 else None)
```

VS Code visually highlights the conflicts and you can solve them (by clicking on the correct one/editing):



```python
# Simple greeting program

def greeting(context=None):
        prefix='Hello'
        if context == 'teaching':
                suffix="BIO-210!"
        elif context == 'research':
<<<<<<< HEAD (Current Change)
                suffix="Alberto"
=======
                suffix="Lucas"
>>>>>>> feature/new_context (Incoming Change)
        else:
                suffix = "World!"

        print(prefix,suffix)

if __name__ == "__main__":
    import sys
    greeting(sys.argv[1] if len(sys.argv)>1 else None)
```

A multiple-developer git workflow

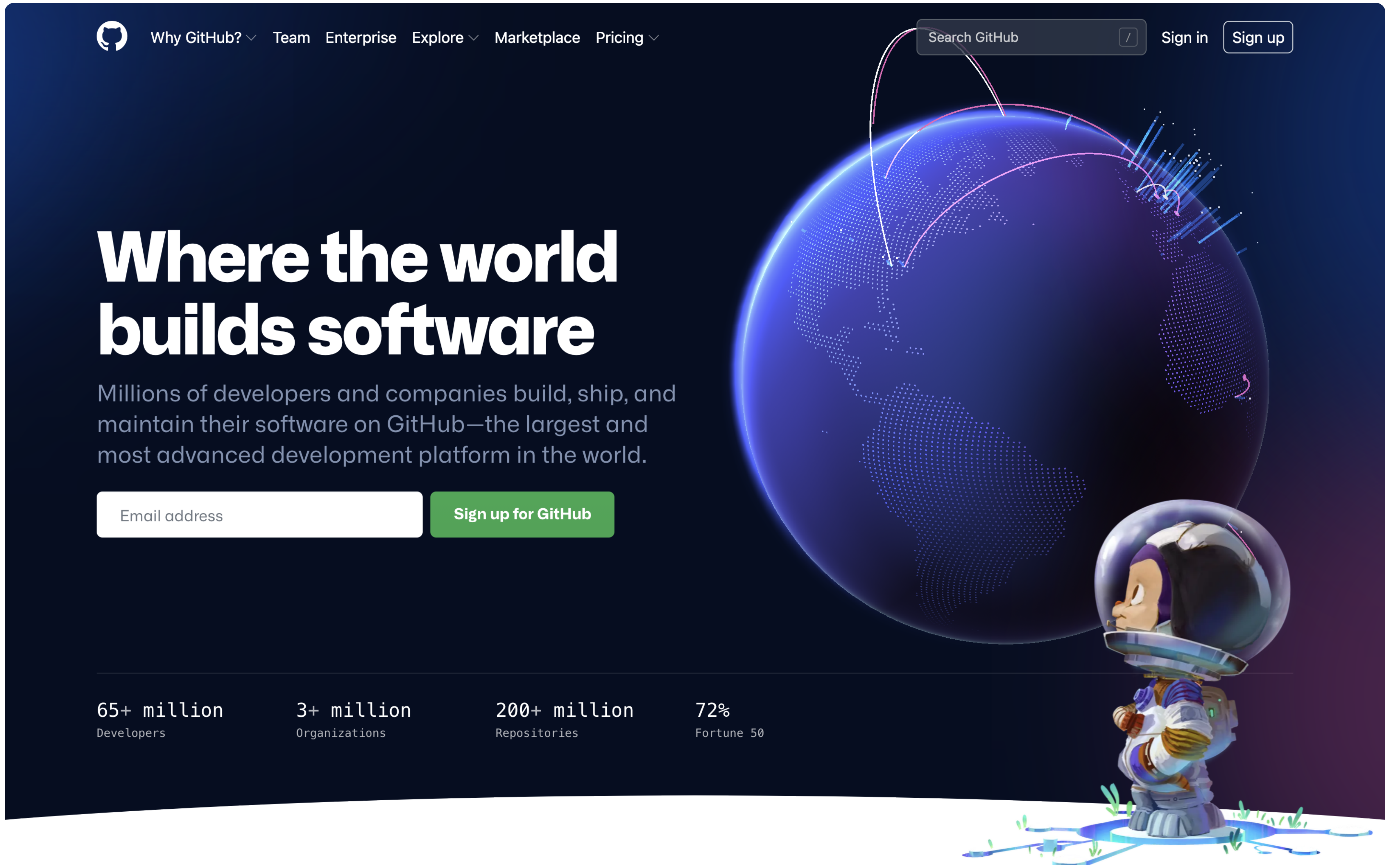


You will practice this (a bit) in the exercises and then we cover it next week. Image source: git docs

Git beyond the command line?

- there are many graphical interfaces for git
- git is also integrated in many integrated (code) development environment (IDEs, e.g., Visual Studio Code (our focus), atom, PyCharm, etc.)

GitHub provides web-based software development and version control using git



Where the world builds software

Millions of developers and companies build, ship, and maintain their software on GitHub—the largest and most advanced development platform in the world.

65+ million Developers

3+ million Organizations

200+ million Repositories

72% Fortune 50

Advanced topics

- vast resources in the git-book
- Oh Shit, Git!?! (some nice solutions to git-hell)
- removing sensitive data from git
- avoiding accidental commits

Today’s summary

We learned about:

- introduction to data model of git
- intro to git (init/staging/branching/merging)
- in exercises: you will practice those, use GitHub and familiarize yourself with git clone/pull/push

Questions?

After lunch:

- Monday 13 - 15: exercises (5 groups)
  - CO4 ← A-B + Y + Z (TA: Albert Dominguez Mantes)
  - CO5 ← C-F+V (TAs: Hale-Seda Radoykova, Shaokai Ye)
  - CO260 ← G-L+W (TA: Oliver Ulrich)
  - CO6 ← M-P (TA: Haoze Qi)
  - CO023 ← Q - U (TA: Andy Bonnetto)
- Monday 15:15 - 16: my office hours at SV 2811
- Do not forget to do the quiz.