Organizing and Sharing Python Code

Sebastian Proksch and Harald Gall

University of Zurich, Department of Informatics

Python Modules

- You often want to group related functions and classes
- You can do this by defining a module - this is just a Python source file
- Use the `import` statement to load a module

```
# saved as foo_bar.py
def foo():
    print('I am foo')

def bar():
    print('I am bar')

class FooBar(object):
    def m(self):
        print('I am foo_bar')
```

```
import foo_bar

foo_bar.foo() # prints 'I am foo'
foo_bar.bar() # prints 'I am bar'

o = foo_bar.FooBar()
o.m () # prints 'I am foo_bar'
```

You can import multiple modules

```
import os, time, random

# or like this


import os
import time
import random
```

You can import subfolders

```
# file: x/y/z.py
def foo():
  print “foo”
```

You can organize your code in subfolders.

```
# file: base.py
import x.y.z
z.foo()
```

... and access the defined symbols.

```
# file: a/b/c.py
import x.y.z
z.foo()
```

Import paths are relative to the “module search path”, which excludes the directory in which Python is being executed

You can also change the name

```
import random as rnd

print(rnd.randint(1, 10))
```

Do not confuse the "." notation with objects and their methods

```
import a.b.c.d.e.f as x

x.doSomething()
```

This is especially useful for deeply nested subfolders.

Import selected symbols from a module

```
# saved as x/y/foo_bar.py
def foo():
    print('I am foo')

def bar():
    print('I am bar')

class FooBar(object):
    def m(self):
        print('I am foo_bar')
```

```
from x.y.foo_bar import foo

foo() # prints 'I am foo'
bar() # Error!!!
```

This is equally useful for deeply nested subfolders, and might even be easier to read.

This is the recommended “Pythonic” way to import.

Do not import all symbols from a module!

```
from foo_bar import *

foo() # prints 'I am foo'
bar() # prints 'I am bar'
```

These “wildcard imports” seem to be useful, but good style recommends to explicitly name all the imports, instead.

- It is clear what is actually being used (and what not).

- The local scope is not automatically polluted with everything that is being declared in the other file.

Modularization in Practice

What happens at import

- Python first creates a new namespace for accessing all the objects defined in the imported source file (a namespace is a mapping from names to objects, you can think of it as a Python dictionary).
- Then it executes the code contained in the file.
- And it creates a name within the calling program that refers to the namespace and can be used to access the objects in the module.

```
import foo_bar
foo_bar.foo()
```

The Python Module Contains A Script

- At import, Python will execute all code contained in a module file.

```
# foo_bar.py
def foo():
    print('I am foo')

def bar():
    print('I am bar')

class FooBar(object):
    def foo_bar(self):
        print('I am FooBar')

print('I am the module') # !!!
```

```
# another_file.py
import foo_bar

# executing this file will print
# 'I am the module'
```

Execution of Python source files

- A Python source file can be executed in two ways:
    - from the command line (this includes PyCharm)
    - whenever you import it as a module
- In addition to definitions of classes or functions, it may contain a script that is then executed
- We prevent this by using the module name `__name__`
    - From the console: `__name__ == ‘__main__’`
    - From the Python console: `'__builtin__’` or sometimes `‘builtins’`

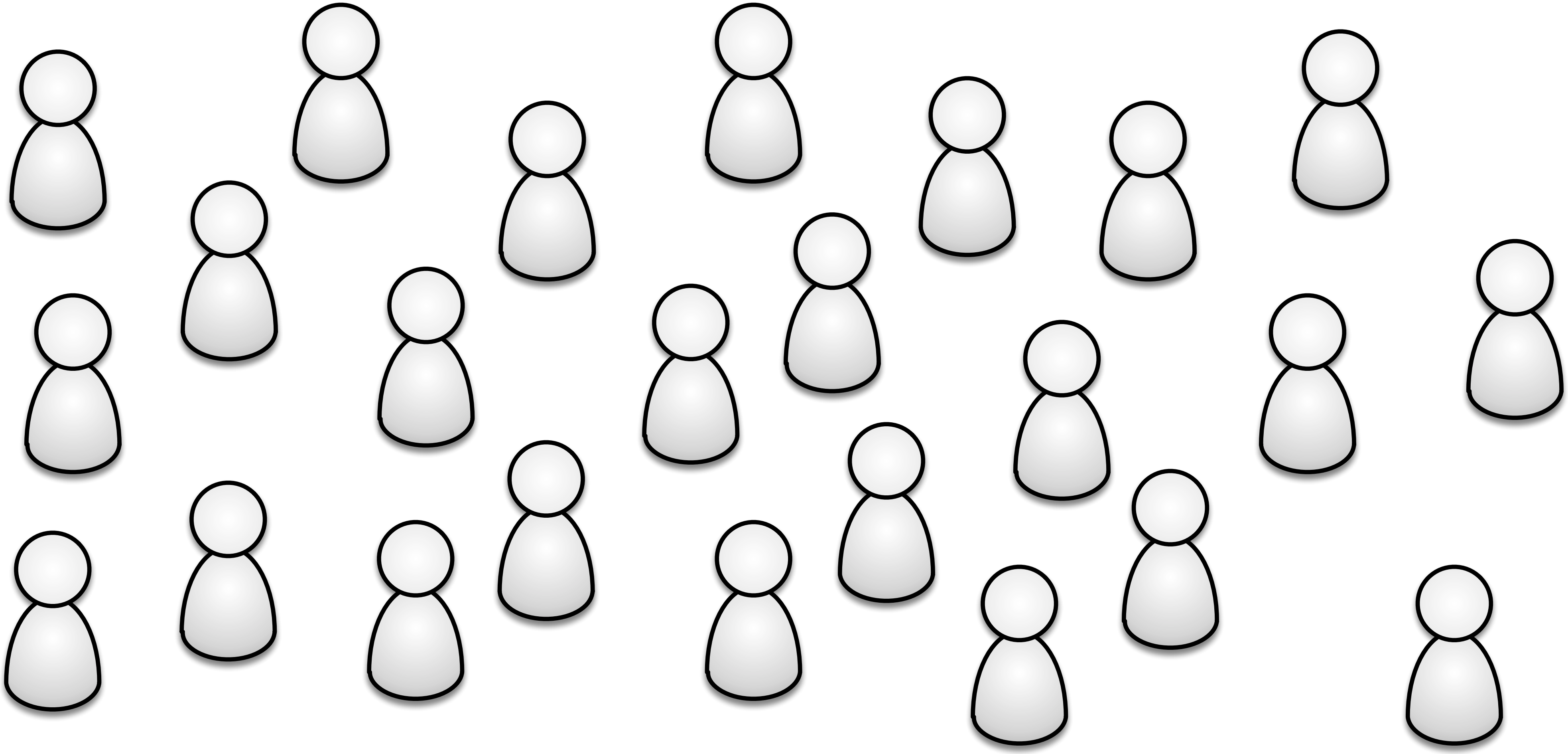
Script Execution Can Be Avoided

```
# foo_bar.py
def foo():
    print('I am foo')
…

if __name__ == '__main__':
    print('I am the foo_bar module')
# else: I am loaded as a module,
# I don't do anything
```

```
# another_file.py
import foo_bar

# importing won't print anything
```

Sharing Code

How do you share source code with others?



Email?

- Source code is not versioned
- Merging concurrent changes is hard

File Sharing?

- Easy synchronization
- Versioning of files
- Conflict detection
- No separation of development tasks
- Partial roll back is cumbersome

Collaborative Editors?

- Immediate synchronization make conflicts very unlikely
- Versioning of files
- No separation of development tasks
- Partial roll back is cumbersome

Goals

- Synchronize source code
- Detect and resolve conflicts
- Traceability (who changed what?)
- Separated development
- Ability to roll back to known working versions

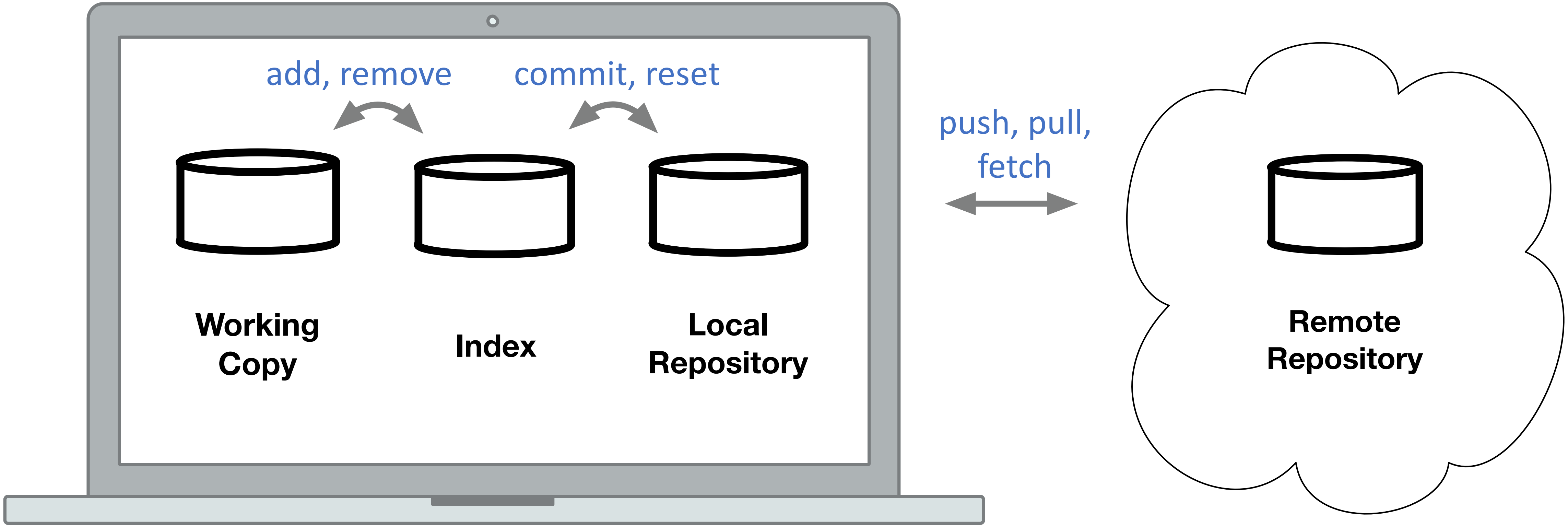
Version Control Systems

Version Control Basics

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

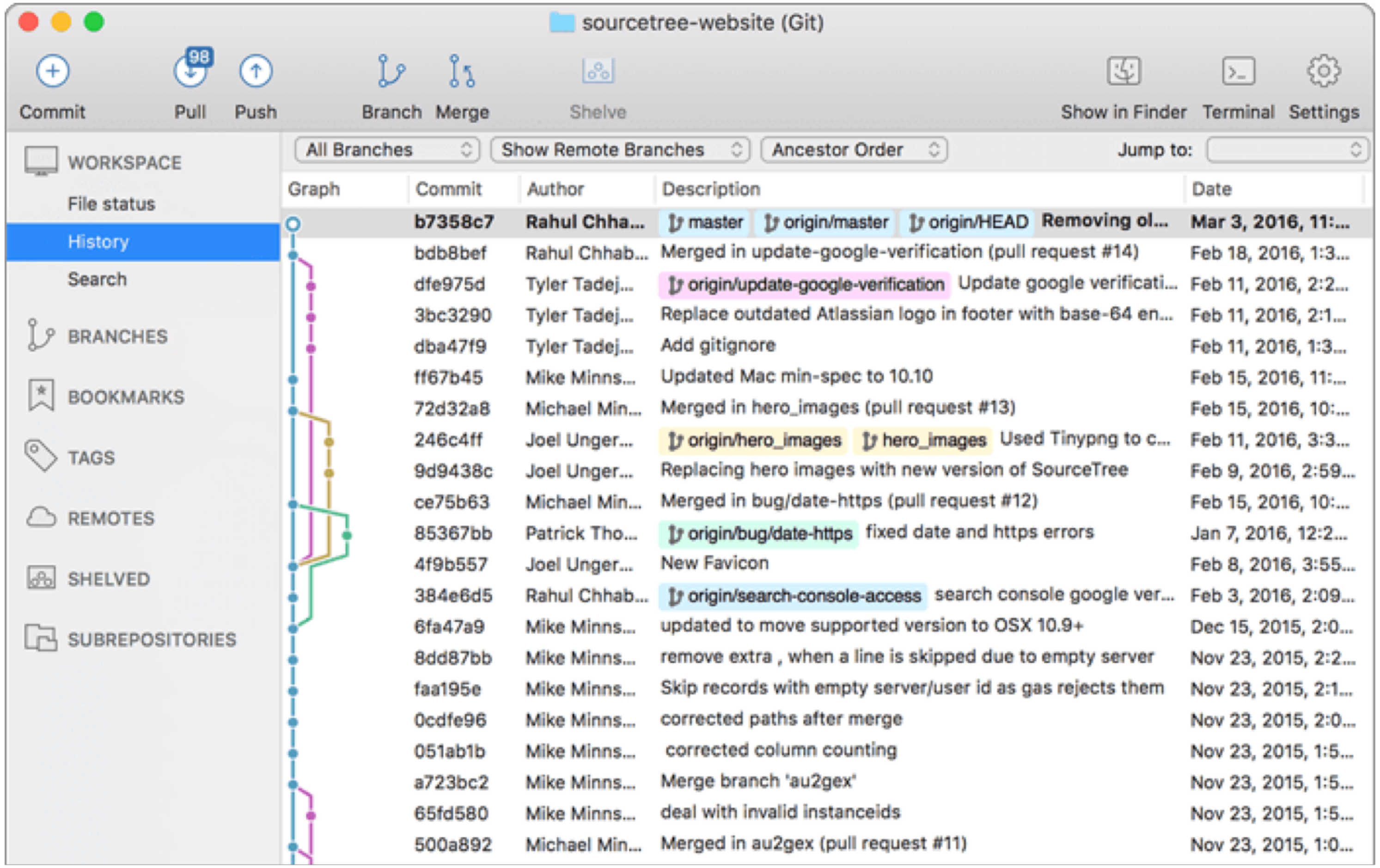
- you can see how files have been changed over time
- revert changes that cause problems

Overview over Git



Git Basics – How to use it

- Command Line
- GUI client



Graph	Commit	Author	Description	Date
	b7358c7	Rahul Chha...	master origin/master origin/HEAD Removing ol...	Mar 3, 2016, 11:...
	bdb8bef	Rahul Chhab...	Merged in update-google-verification (pull request #14)	Feb 18, 2016, 1:3...
	dfe975d	Tyler Tadej...	origin/update-google-verification Update google verificati...	Feb 11, 2016, 2:2...
	3bc3290	Tyler Tadej...	Replace outdated Atlassian logo in footer with base-64 en...	Feb 11, 2016, 2:1...
	dba47f9	Tyler Tadej...	Add gitignore	Feb 11, 2016, 1:3...
	ff67b45	Mike Minns...	Updated Mac min-spec to 10.10	Feb 15, 2016, 11:...
	72d32a8	Michael Min...	Merged in hero_images (pull request #13)	Feb 15, 2016, 10:...
	246c4ff	Joel Unger...	origin/hero_images hero_images Used Tinypng to c...	Feb 11, 2016, 3:3...
	9d9438c	Joel Unger...	Replacing hero images with new version of SourceTree	Feb 9, 2016, 2:59...
	ce75b63	Michael Min...	Merged in bug/date-https (pull request #12)	Feb 15, 2016, 10:...
	85367bb	Patrick Tho...	origin/bug/date-https fixed date and https errors	Jan 7, 2016, 12:2...
	4f9b557	Joel Unger...	New Favicon	Feb 8, 2016, 3:55...
	384e6d5	Rahul Chhab...	origin/search-console-access search console google ver...	Feb 3, 2016, 2:09...
	6fa47a9	Mike Minns...	updated to move supported version to OSX 10.9+	Dec 15, 2015, 2:0...
	8dd87bb	Mike Minns...	remove extra , when a line is skipped due to empty server	Nov 23, 2015, 2:2...
	faa195e	Mike Minns...	Skip records with empty server/user id as gas rejects them	Nov 23, 2015, 2:1...
	0cdfe96	Mike Minns...	corrected paths after merge	Nov 23, 2015, 2:0...
	051ab1b	Mike Minns...	corrected column counting	Nov 23, 2015, 1:5...
	a723bc2	Mike Minns...	Merge branch 'au2gex'	Nov 23, 2015, 1:5...
	65fd580	Mike Minns...	deal with invalid instanceids	Nov 23, 2015, 1:5...
	500a892	Michael Min...	Merged in au2gex (pull request #11)	Nov 23, 2015, 1:0...

https://www.sourcetreeapp.com

Git Basics – Creating a repository

- initialize a Git repository in the current directory or clone an existing one

```
$ git init  # new repository

or

$ git clone «repository url»  # existing repository
```

Git Basics – File states

- Files in the working directory can be tracked or untracked.
- Whenever you create a new file, it will be untracked first.
- You can check the status of your working dir and index.

```
$ echo ‘…’ > foo_bar.py
$ git status
…
Untracked files:
   foo_bar.py
```

Git Basics – File states

- You tell Git to stage all changes to a file in the index using the add command.

```
…
$ git add foo_bar.py
$ git status
…
Changes to be committed:
   new file:   foo_bar.py
```

Git Basics – Track all files

- You can add multiple (all) files at once.

```
$ touch file_a.py && touch file_b.py
$ git add .
$ git status
…
Changes to be committed:
   new file:   file_a.py
   new file:   file_b.py
```

Git Basics – Committing the first file

- You can commit your changes, this will save all changes that are stored in the index as a consistent change set
- Each commit is being identified by a commit ID
- Each commit has one or more parents

```
$ git commit -m "Added foo_bar script"
```

Git Basics – Commit History

- to view what has happened so far in a repository you can use the log command
- this lists the commits in reverse chronological order

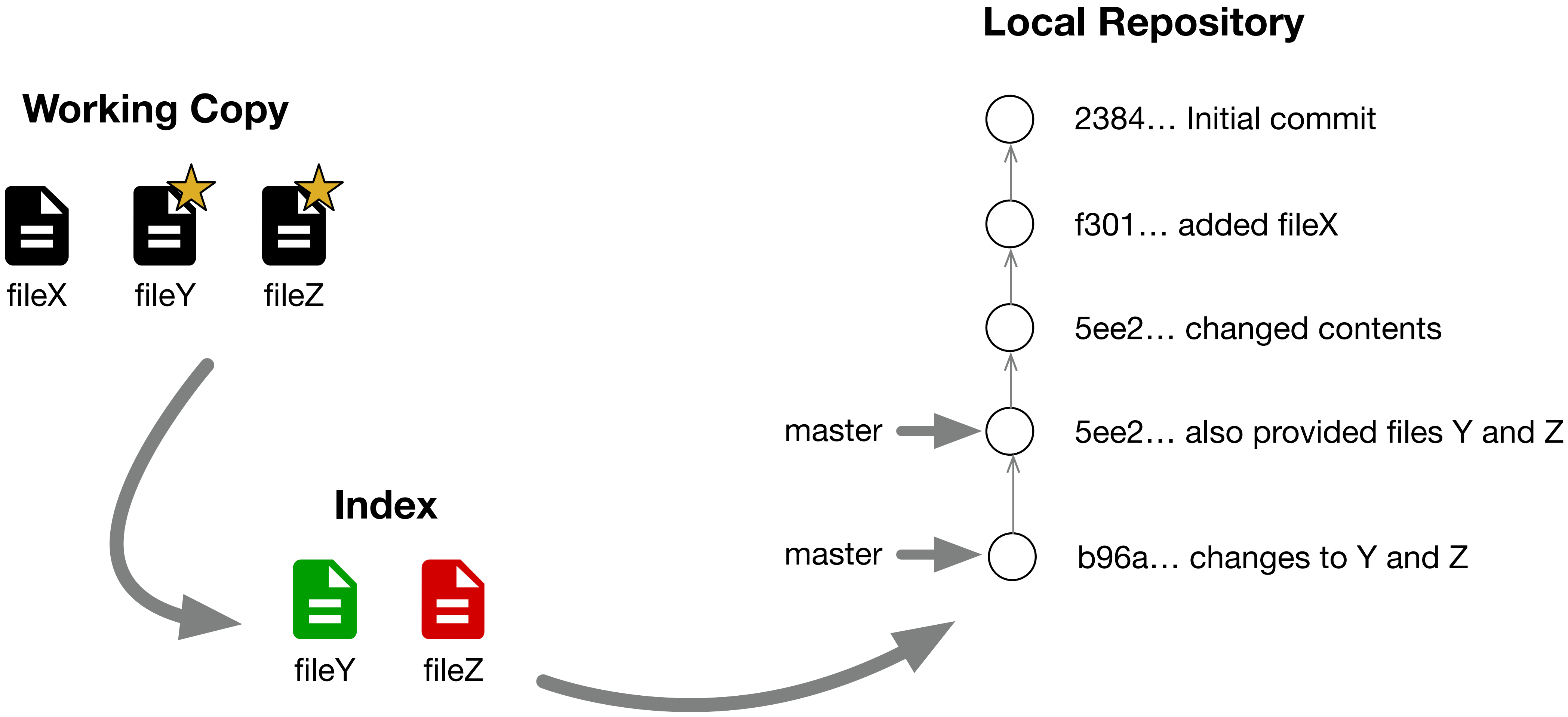
```
$ git log
commit 08f9133af8ec… (HEAD -> master)
Author: seb <proksch@ifi.uzh.ch>
Date:   Tue Dec 4 18:16:31 2017 +0100
    Added foo_bar script
```

Git Basics – Recovering from Unwanted Changes

- core advantage of using Git is that you can always roll back to previous commits
- common example: rolling back to last time you commited

```
$ git reset –hard
```

Git Visualization



Branching

Git Branching

- the Git branching model often considered its most important and useful feature
- a **branch** is a parallel line of development
- you can create a new branch starting from the master (the main line of development), make some changes and once you are satisfied with them **merge** them back to the master

Git Branching - Example

- let us check the status of the current repository
- please note the message 'On branch master'

```
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to
track)
```

Git Branching - Example

- let us create a README file with one line of text and commit it to the repository

```
$ echo 'Master README' >> 'README'
$ git add README
$ git commit -m "Added README file."
```

Git Branching - Example

- and check the current status, please note again the reference to master branch, this means we are currently on the master branch

```
$ git status
On branch master
nothing to commit, working directory clean
```

Git Branching - Example

- let's create a new branch called testing

```
$ git branch testing
```

Git Branching - Example

- we can view the branches we currently have with the **branch** command
- the * indicates the current branch you are on

```
$ git branch
* master
  testing
```

Git Branching - Example

- to switch to the testing branch we need to use the **checkout** command

```
$ git checkout testing
Switched to branch 'testing’
$ git branch
master
* testing
```

Git Branching - Example

- let's add a second file called README_2 and commit it

```
$ echo 'Second README' >> 'README_2'
$ git add README_2
$ git commit –m 'Added second README.'
```

Git Branching - Example

- if you list the contents of the current directory, you will see you have two files (ls for Mac or Linux, dir for Windows)

```
$ ls
README          README_2
```

Git Branching - Example

- please note, that we are still on the testing branch

```
$ git branch
  master
* testing
```

Git Branching - Example

- let's switch to the master branch and check what files we have in the current directory
- we only have one README file

```
$ git checkout master
$ ls
README
```

Git Branching - Example

- Important: whenever you switch branches, Git will reset your working directory to the state it had the last time you committed on that branch

```
$ git checkout master
$ ls
README
```

Git Branching - Example

- now you would like to add the changes you did on the testing branch to the master branch, for that you need to use the **merge** command
- we should first make sure we are on the master branch

```
$ git branch
* master
  testing
```

Git Branching - Example

- we can then **merge** the changes

```
$ git merge testing
Updating 14bd296..4fd536e
Fast-forward
 README_2 | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 README_2
```

Git Branching - Example

- if we now check the current directory, we can also see the README_2 file

```
$ ls
README          README_2
```

Git Branching - Example

- we do not need the testing branch anymore, so we can delete it
- now we only have the master branch

```
$ git branch -d testing
Deleted branch testing (was 4fd536e).
$ git branch
* master
```

Git Branching – Merge conflicts

- this can happen for example if you modified the README file both on the master and testing branch

Git Branching – Merge conflicts

- when you try to merge you will get a message saying you have a conflict

```
$ git merge testing
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

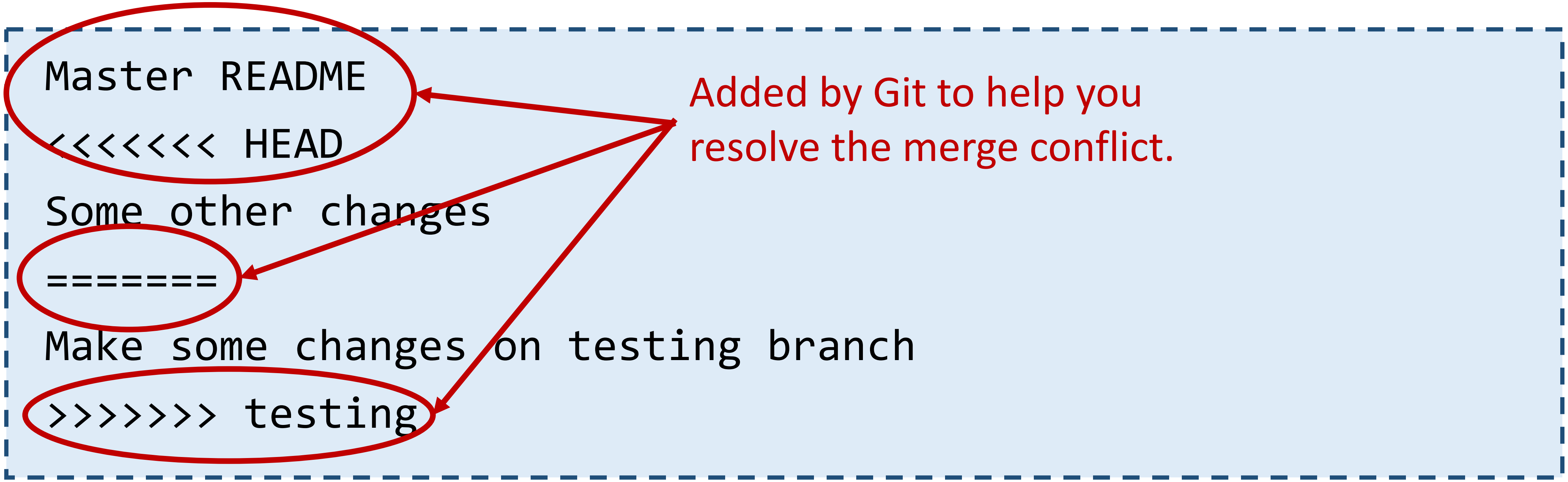
Git Branching – Merge conflicts

- this is something that you will have to handle manually, the content of the README file will look something like this:

```
Master README
<<<<<<< HEAD
Some other changes
=======
Make some changes on testing branch
>>>>>>> testing
```

Git Branching – Merge conflicts

- Git adds conflict resolution markers to the files with merge conflicts, so you can fix them



```
Master README
<<<<<<< HEAD
Some other changes
========
Make some changes on testing branch
>>>>>>> testing
```

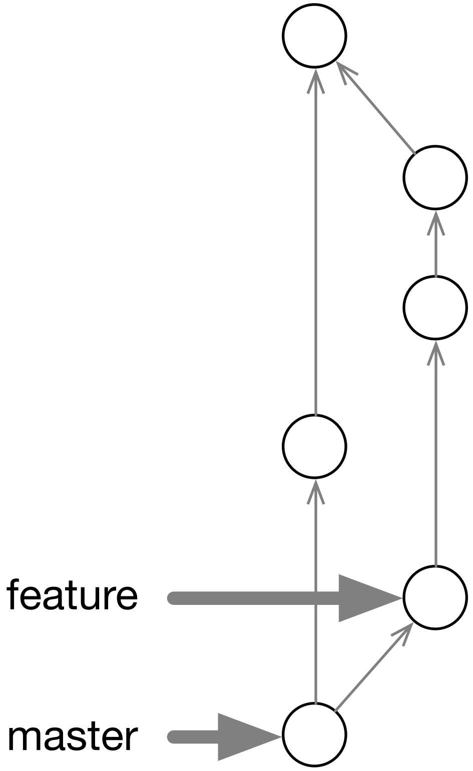
Added by Git to help you resolve the merge conflict.

Git Branching – Merge conflicts

- after handling the merge conflict, you should commit the modified file and the merge is done

```
$ git add README
$ git commit –m "Handled merge conflicts."
```

Git Branch Visualization



feature

master

2384… Initial commit

f301… added fileX

5ee2… also provided files Y and Z

5ee2… changed contents

b96a… changes to Y and Z

b96a… merge ‘feature’ onto ‘master’

Remote Repositories

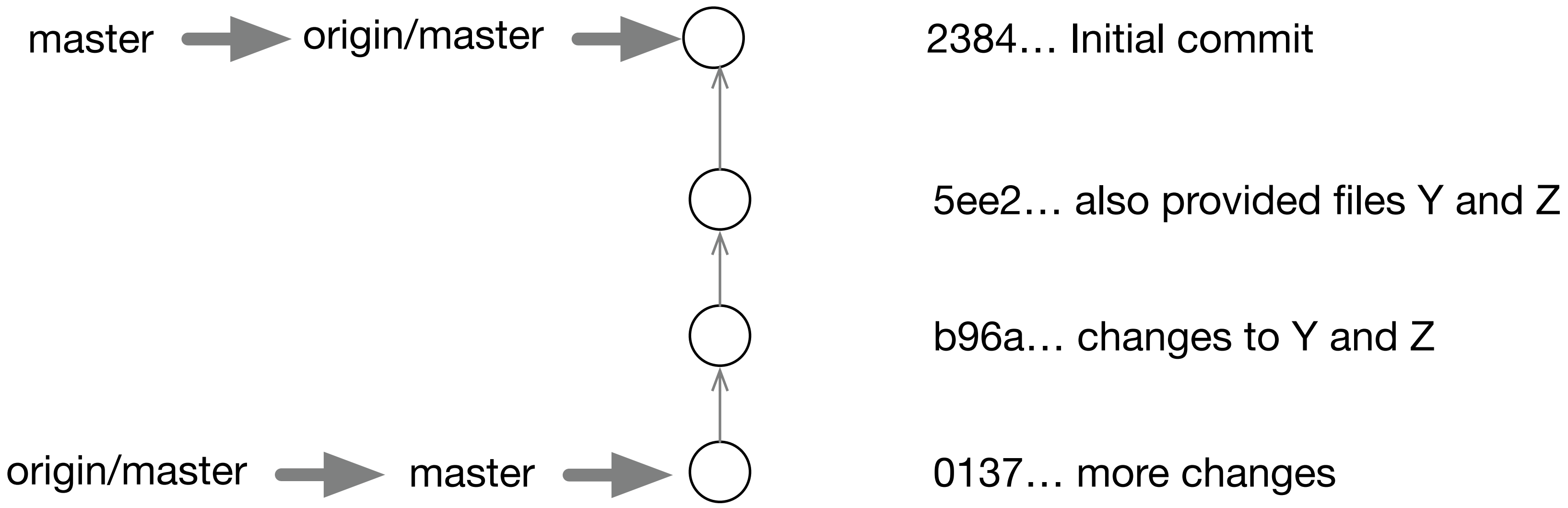
Clone a Remote Repository Locally

- clone your repository to your local computer
- The remote `master` branch is typically called `origin/master` locally.

```
$ git clone https://github.com/yourusername/info1_test.git
Cloning into 'info1_test'...
Checking connectivity... done.
```

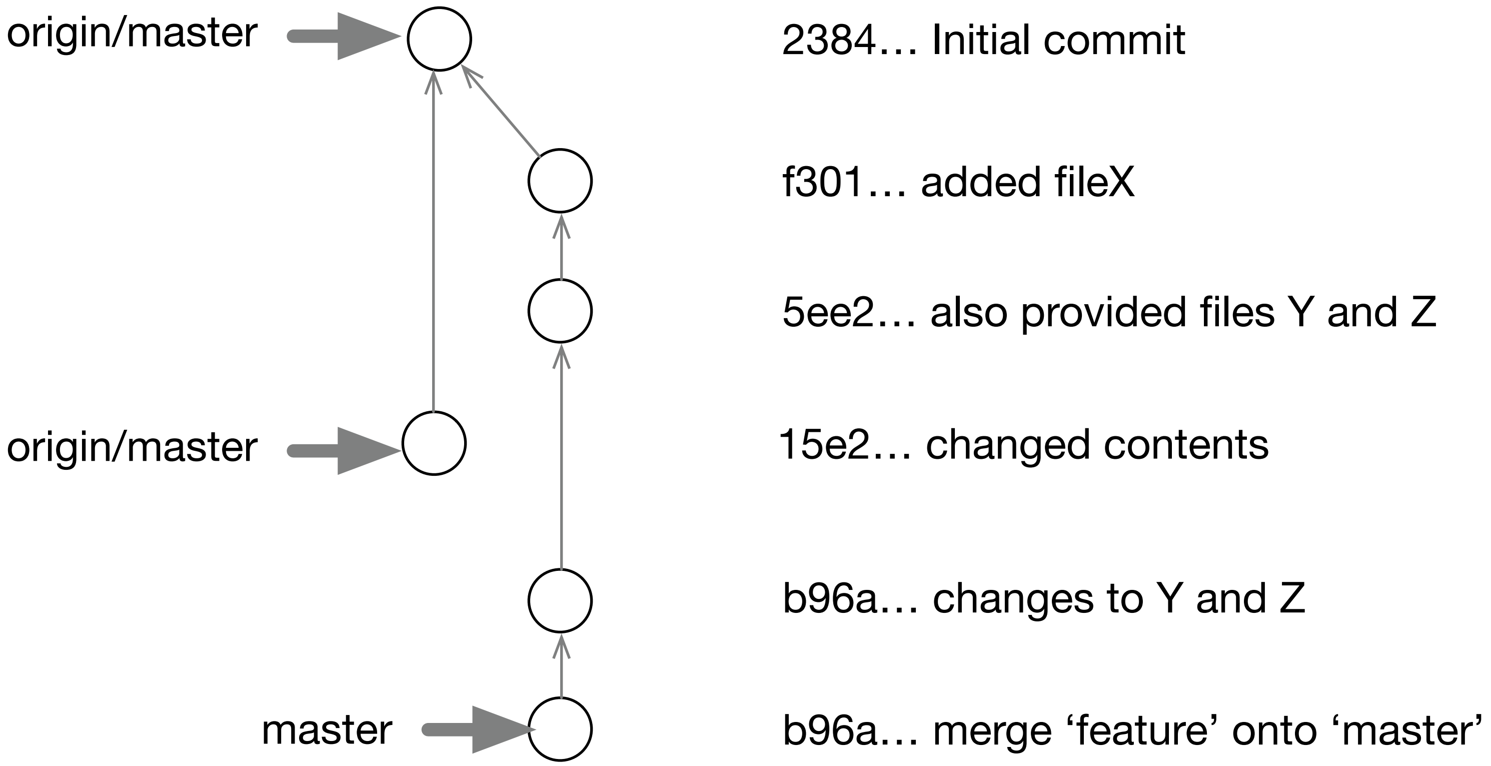
Push Your Changes

- Use `git push` to send your changes back to the remote repository
- Direct path must exists from the remote pointer to the local pointer



Synchronize Local and Remote Repositories

- Use `git fetch` to update your local repository with remote changes



2384… Initial commit

f301… added fileX

5ee2… also provided files Y and Z

15e2… changed contents

b96a… changes to Y and Z

b96a… merge ‘feature’ onto ‘master’

Git Pull

- You can use `git pull` as a convenience function that combines `git fetch` and `git merge`.
- You will automatically get many merge commits!

- Advanced strategies *rewrite* history to keep it linear (`git rebase`) or drop history to *squash* feature commits (`git reset --soft`)
- These are out of scope for this course, but looking them up is highly encouraged!

Take Home Message

You should be able to answer the following:

- How do I define a Python module? How do I import it?
- How can I define a script that is not executed for each import?
- What is Git? What is the difference between the working copy, the index, and the local and remote repositories?
- How do I add/remove changes to the index?
- How do I check status of working directory and index?
- How do I commit the changes staged in my index?
- How do I push my changes to the remote repository?
- How do I synchronize my local repository with the remote repository?
- How (and why) do I use branches?

Exercise

1) Modularization and 2) Git

Exercise 1: Modularization

- Create a class in one file and use it in another file
- Try out subfolders
- Reference a class that references another class
- Try `import`
- Try `import X as Y`
- Try `from X import Y`

Exercise 2: Git

- Create a local repository
- Add and commit changes
- Add a remote and push changes
- Create branches