Tobias Weis

# ML Praktikum 17/18

## Version control, PEP8, documentation

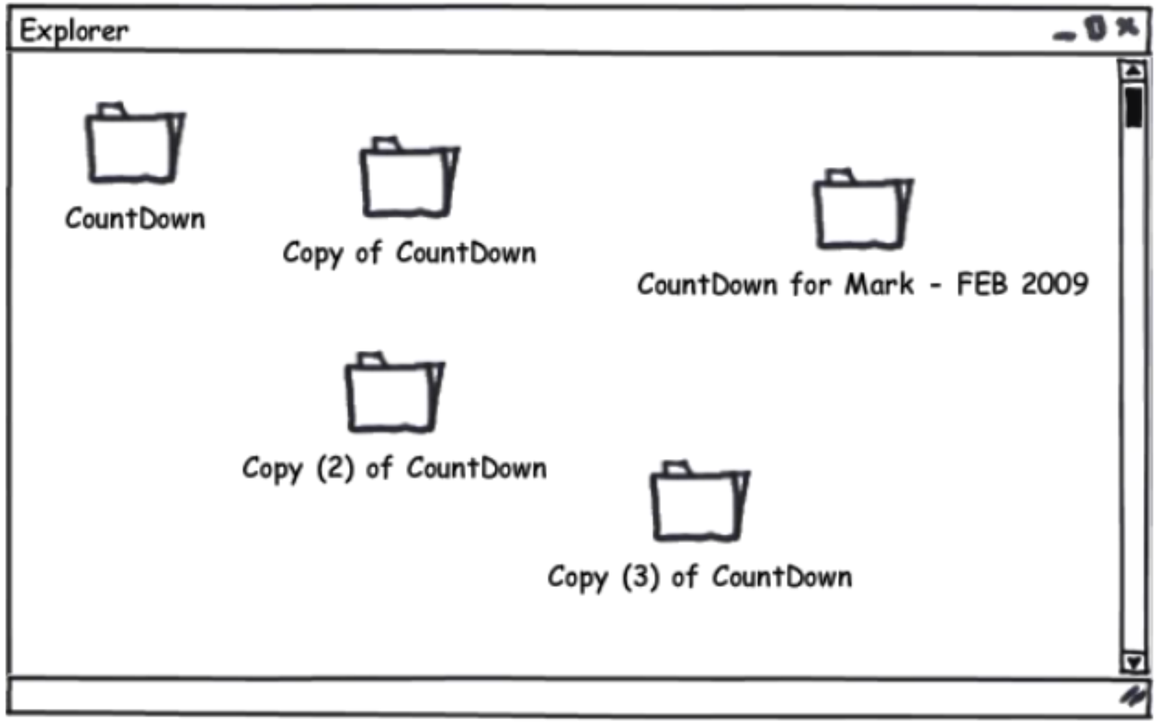
# Agenda

- **Versioning systems**
- PEP8 and docstrings
- Code documentation

# Versioning - What is it and why should I use it?

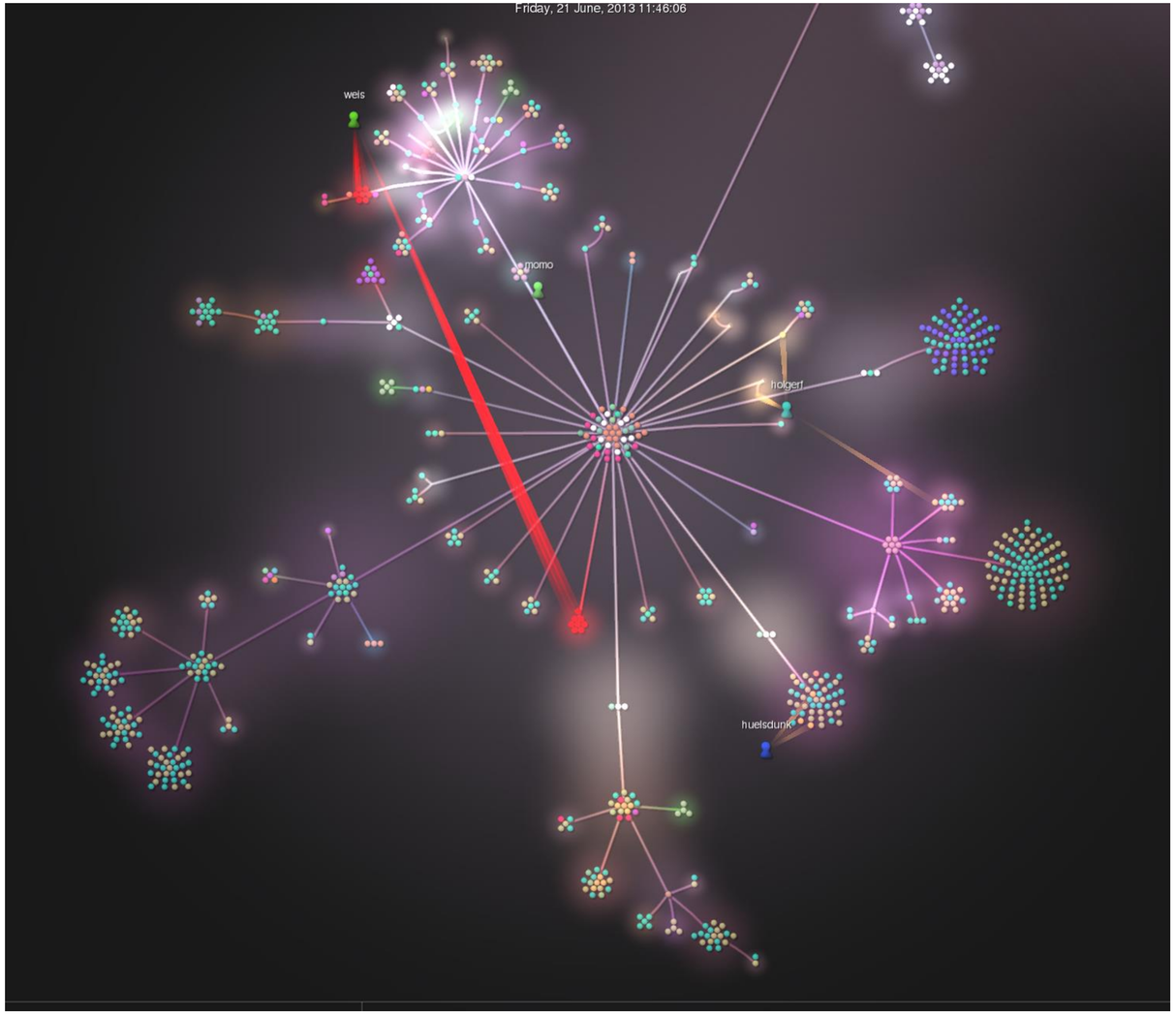
- Keeping track of changes to your code / documents
- Documenting those changes
- Some people do it like this:

- Reasons to use versioning:
  - Troubleshooting
  - Logging
  - Simplify addition of features
  - Share code with a team




Source: http://de.slideshare.net/eykanal/git-introductory-talk

# Versioning - What is it and why should I use it?




gource combined.txt -s 0.01 -f

# Versioning – Which system?

Works by having a copy of your code on a server as a master-copy, changes can be sent or received.

There exists a huge ecosystem of different version control systems, most common are:

SVN (subersion), CVS,

Non-distributed
(all commits to server)

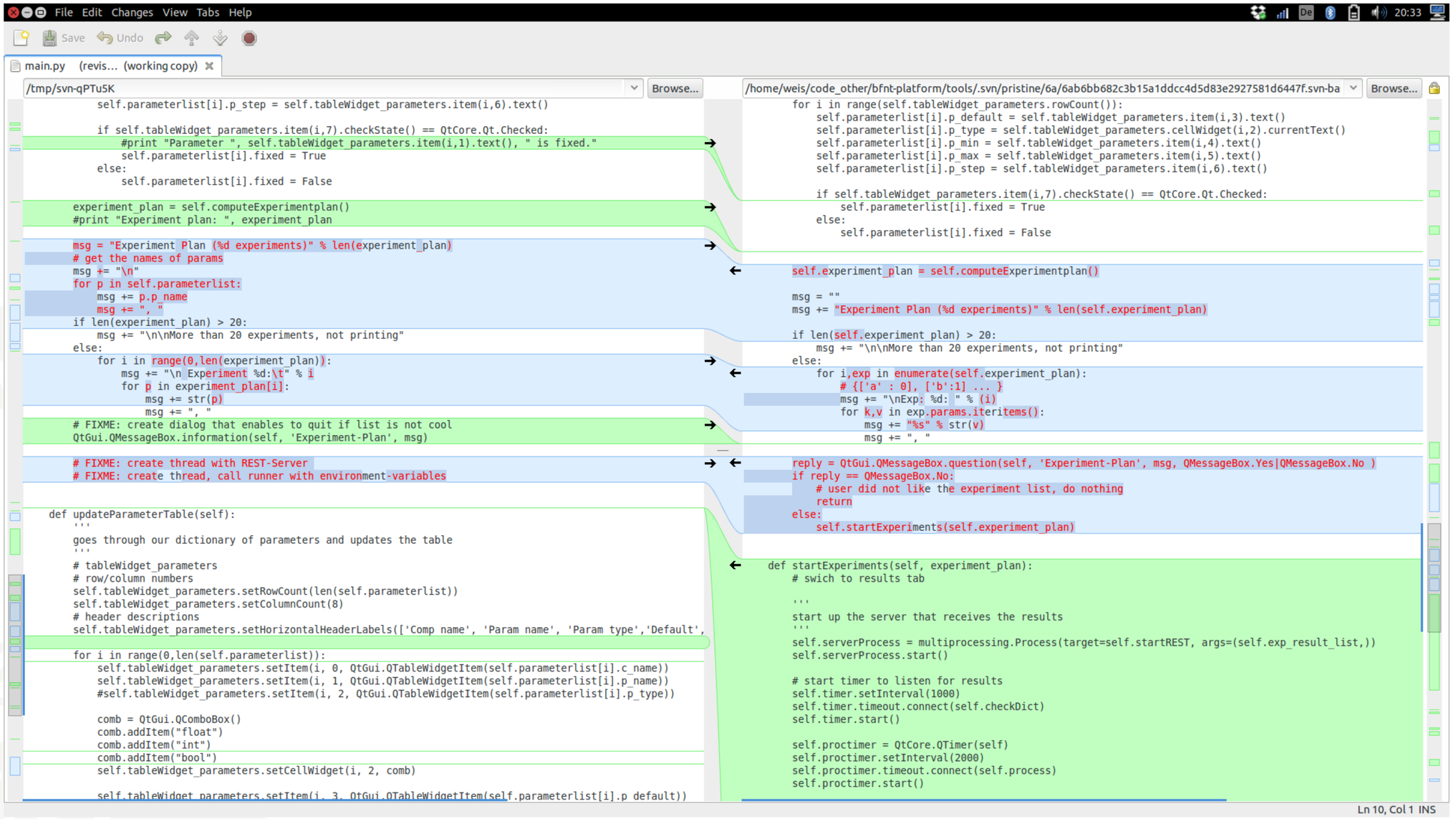
GIT, Mercurial

Distributed
Server has master-repo,
but local repo on your machine

This enables:

- Keeping track of changes to fiels (and folders)
- Teamwork on code (with multiple authors)
- A record of who did what, and when
- The „why“ is provided by commit messages

# Versioning - Diffs

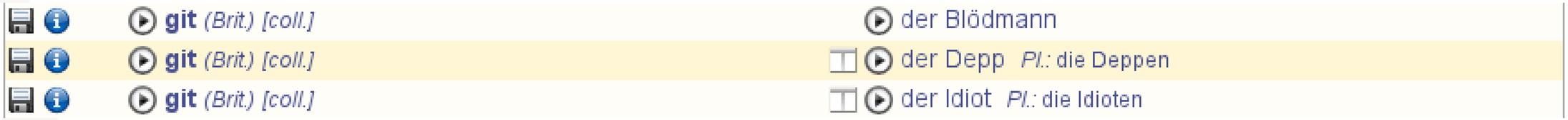


# Versioning - GIT

GIT

- Open source version control system designed for speed and efficiency
- Best insurance against
  - Accidental mistakes (like deleting files)
  - Remember what has been changed (and why)
  - Hard-drive-failures, fire, earthquakes

- Created by Linus Torvalds (for managing Linux kernel)
  - „I‘m an egoistical bastard, and I name all my projects after myself. First ‚Linux‘, now ‚Git‘“
    Linus Torvalds



# Versioning - GIT

Getting a GIT-Server:

- Hosted solutions include github.com (widespread, public repos free), bitbucket.com
- Self-hosted gitserver: gitlab

Getting the GIT client:

- Windows: Download from http://git-scm.com/downloads
- Mac: brew install git
- Linux: sudo apt-get install git

# Versioning - GIT

Why is distributed awesome?

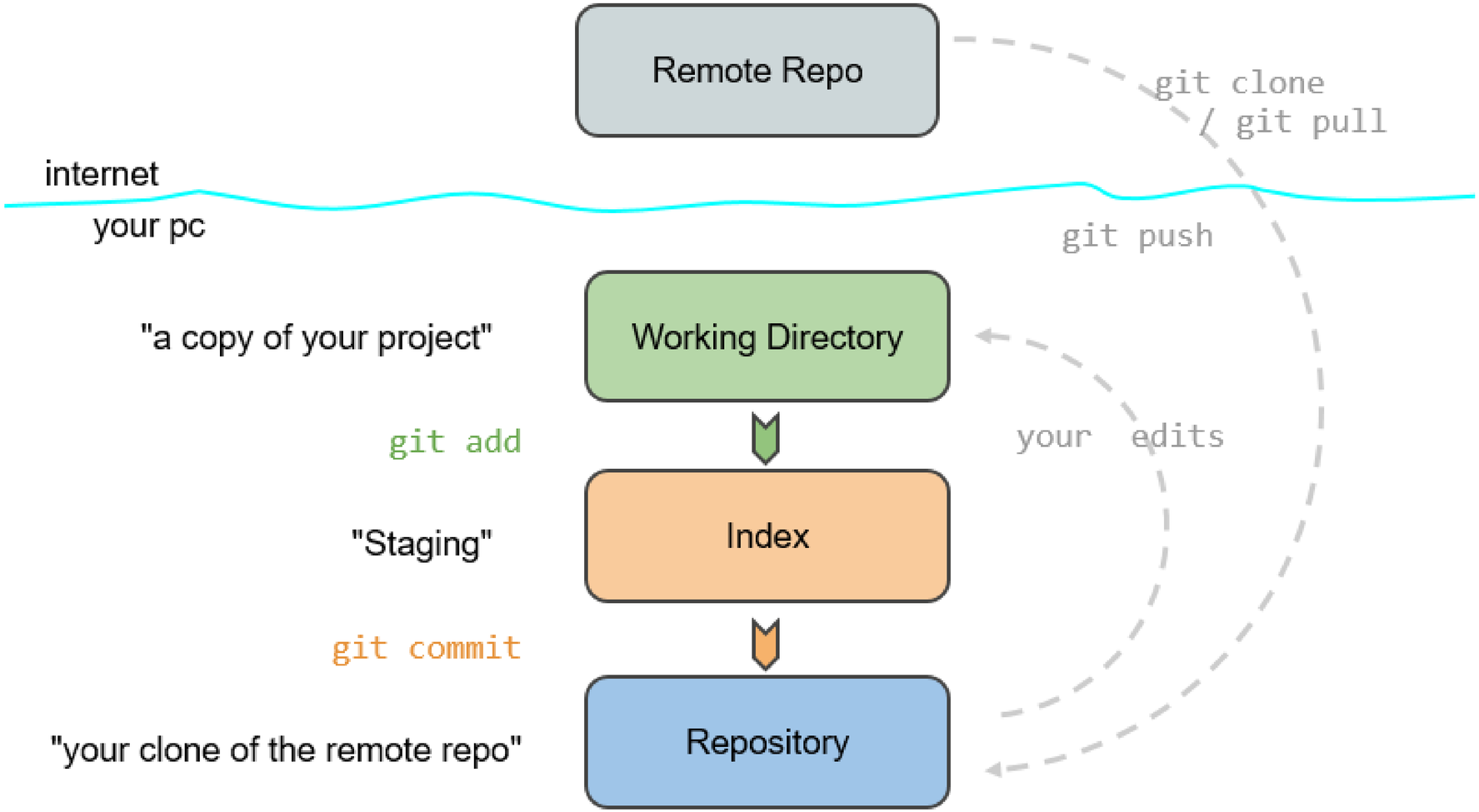
- Use GIT on your local machine without being connected to the internet
- Revisions (commits) you make to your local repo are available to you only
- The next time you are online (or you want to), you can push your changes to the server for sharing/backup
- The local nature makes it effortless to create branches to isolate your work
- A lot of minor changes (local commits) can be pushed as a single commit to remote branch

# Versioning - GIT

Workflow

- Obtain a repo (only needed once)
  - git init, git clone
- If repo already exists, make sure to first pull any changes by others
  - git pull
- Make changes to some files, then
  - git status, git commit –m“Always write clear commit messages“
- Add new files:
  - git add mynewfile.py
  - git commit mynewfile.py –m“added mynewfile.py“
- Push to the server
  - git push

# Versioning - GIT



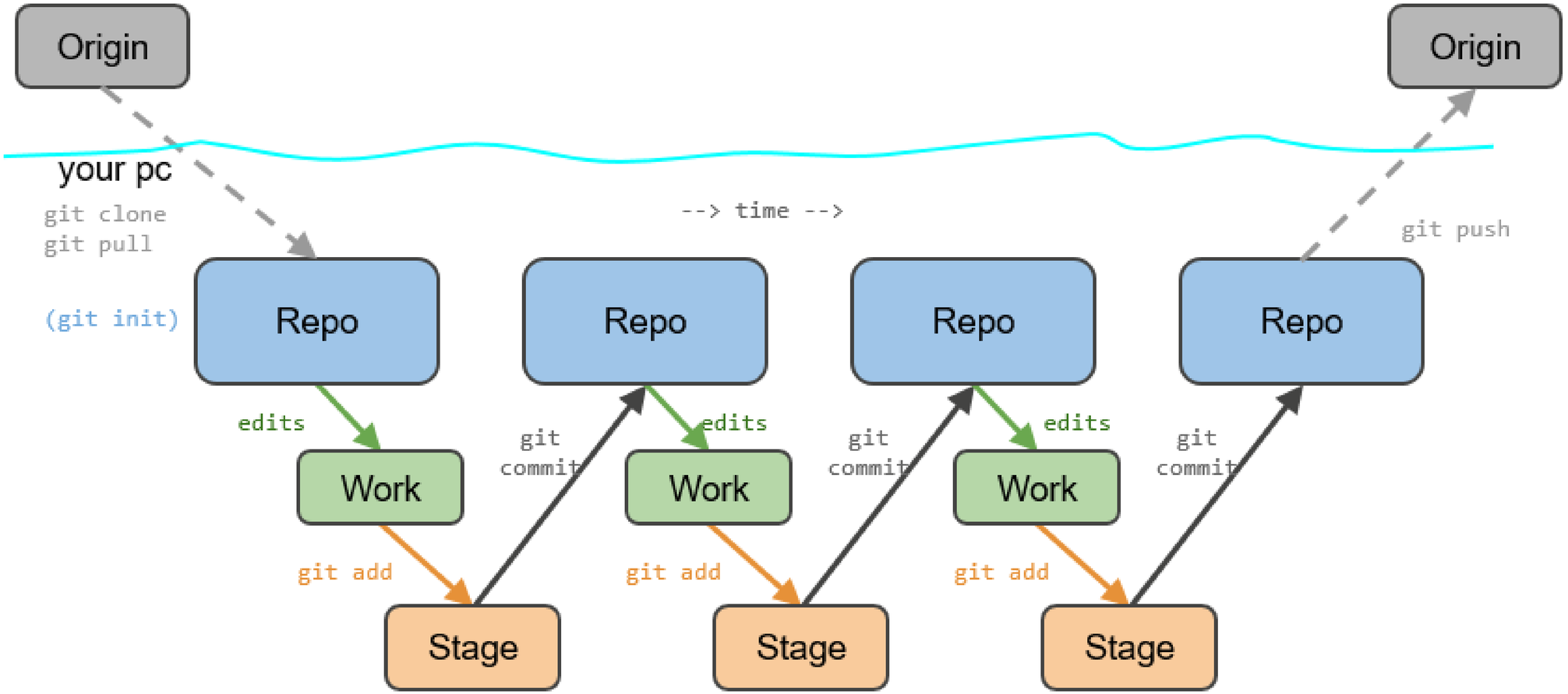

http://www.slideshare.net/jasonjnoble/git101 slide 3

# Key Concepts

| Work is done here | Things "about to be stored" are put here | History, branches are stored here | *Same as Repository, but remote* |
|---|---|---|---|
| | `$ git add` | `$ git commit` | `$ git push`<br>`$ git pull` |
|  |  |  |  |
|  |  |  |  |

Source: http://de.slideshare.net/eykanal/git-introductory-talk

# Versioning - GIT

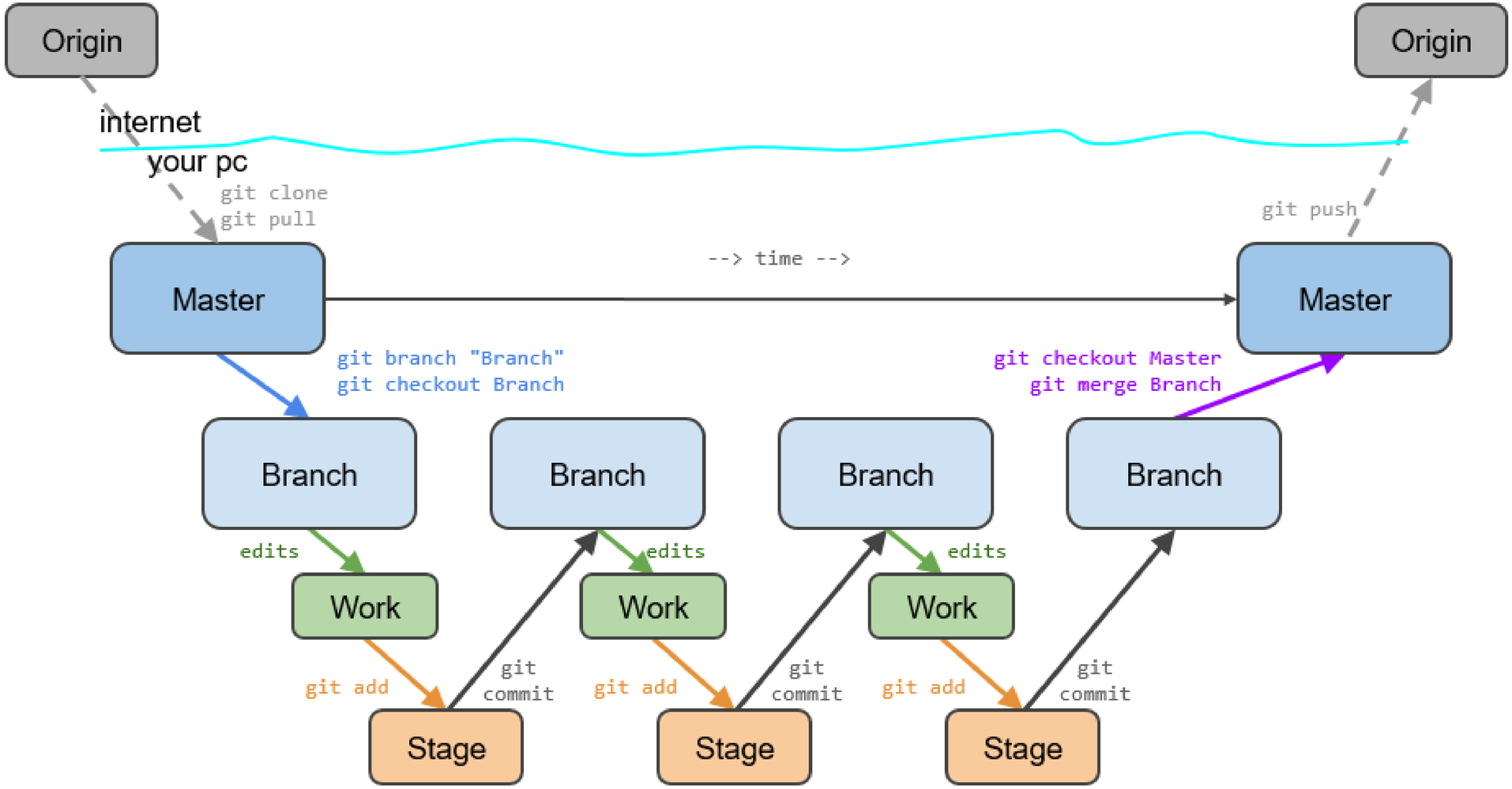


# Versioning – GIT

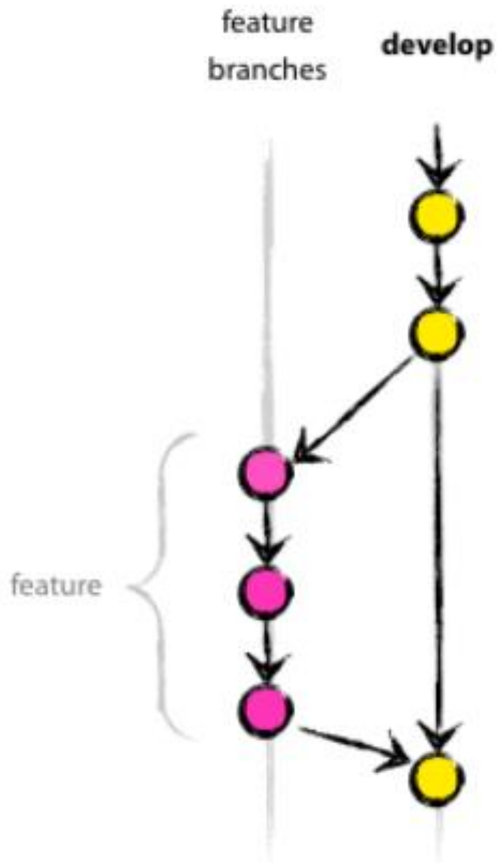
Branches

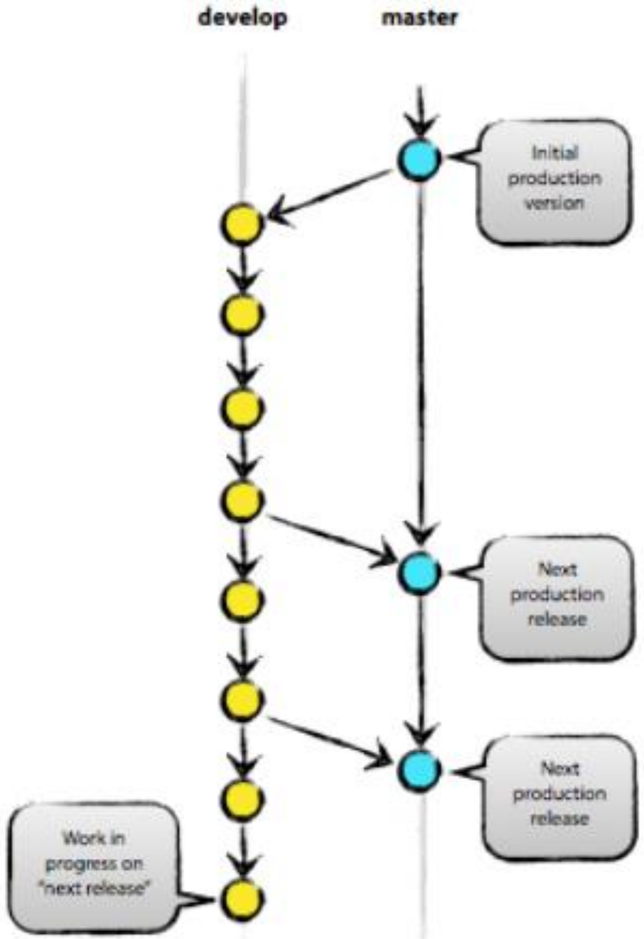
- Allows to try out ideas/experiment isolated from the master-branch of code
- Enables to work on long-running topics without interferring/breaking the master-branch
- Share work-in-progress with others
- Incorporate finished work into master-branch

# Versioning - GIT



# Versioning - GIT

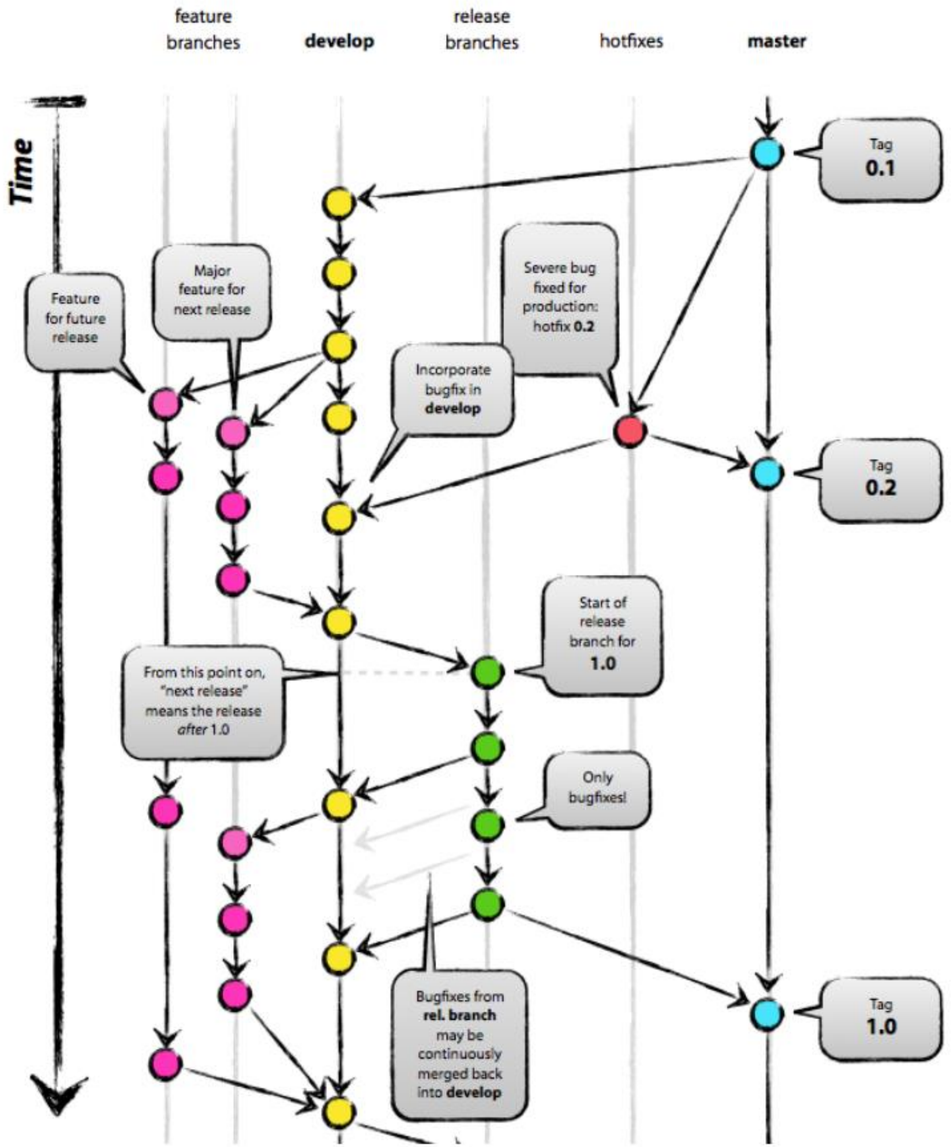




Source: http://de.slideshare.net/eykanal/git-introductory-talk

# Branches – Go Nuts




Source: http://de.slideshare.net/eykanal/git-introductory-talk

# Versioning - GIT

Download/Create a local repo
$ git clone
$ git init

Checking what's changed
$ git status
$ git log

Storing edits
$ git add
$ git commit

Updating your local repo
$ git pull (git fetch)
$ git branch
$ git merge

Storing changes offsite/offbox
$ git commit
$ git push

Storing changes offsite/offbox
$ git tag
$ git push

# Versioning - GIT

Additional tips/conventions:

- Do not put large binary files in plain git (use git-lfs, or a fileserver)
- Always create a README in the toplevel directory describing your work
- Try to only keep runnable code in the master-branch

Github.com:

- Public repos are free
- Easy-to-use web-interface
- Has markdown enabled for *.md files (README)
- Has a wiki for every repo

# Agenda

- Versioning systems
- **PEP8 and docstrings**
- Code documentation

# PEP8

We will (probably) use a lot of python code for experiments.

Guidelines on formatting code, comments and docstrings for easy readability:

**https://www.python.org/dev/peps/pep-0008**

# PEP8

Indentation

- 4 spaces per indentation level
- Continuation lines:

```python
# Aligned with opening delimiter
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# More indentation included to distinguish this from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

YES

```python
# Arguments on first line forbidden when not using vertical alignment
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

NO

# PEP8

- Limit all lines to a max of 79 characters
- Separate top-level function and class definitions with two blank lines
- Method definitions insed a class are separated by a single blank line
- Use blank lines in functions (sparingly) to indicate logical sections

# PEP8

Imports

- Should usually be on separate lines

```
Yes: import os
     import sys

No:  import sys, os
```

- But this is OK:

```
from subprocess import Popen, PIPE
```

- Ordering:
  - Standard libraries
  - Related 3rd Party imports
  - Local specific imports

# PEP8

Whitespaces

- NOT immediately inside parentheses, brackets or braces
- NOT immediately before a comma, semicolon or colon
- NOT immediately before open parenthesis that starts the argument list
- NOT immediately before open bracket that starts indexing or slicing
- NOT more than one space around an operator

Yes:

```
x = 1
y = 2
long_variable = 3
```

No:

```
x             = 1
y             = 2
long_variable = 3
```

# PEP8

- Always surround these binary operators with a single space on either side:
  - =, +=, -=, ==, <, >, !=, <>, <=, >=, in, not in, is, is not, and, or not
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priorities (however, never use more than one whitespace, and use the same amount on both sides of the binary operator!)

Yes:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

# PEP8

- Don‘t use spaces around the = sign when used to indicate a keyword argument or a default parameter value

Yes:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

No:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

# PEP8

Multiple statements per line are generally discouraged!

Yes:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

# PEP8

Comments

- Worst: comments that contradict the code (keep them up-to-date!)
- Should be complete sentences
- Comments should always be written in English
- Block comments generally apply to code that follows them, indented to same level as the code. Each line starts with a # and a single space
- Inline comments are on the same line as the code
- Use inline comments sparingly, and only to add additional explanations

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

# PEP8

Documentation Strings (Docstrings)

- PEP257
- Write them for all public
  - Modules
  - Functions
  - Classes
  - Methods
- Not necessary for non-public methods (but put a comment there (after the def))
- Docstrings start and end with “““

# PEP8

Naming Styles / Conventions

- lowercase
- lowercase_with_underscores
- UPPERCASE
- UPPERCASE_WITH_UNDERSCORES
- CamelCase
- mixedCase
- Capitalized_Words_With_Underscores (ugly!)

- Special forms
  - _single_leading_underscore: weak „internal use“ indicator
  - single_trailing_underscore_ : avoid conflicts with python keywords
  - __double_leading_underscore: internal class attribute, invokes name mangling
  - __double_leading_and_trailing__ : magic objects or attributes

# PEP8

Naming Styles / Conventions

- Modules and packages: short, all-lowercase names, may use underscores
- Class names: CamelCase
- Exception names: see Class names
- Function names: lowercase, may use underscores
- Constants: Usually defined on module level, UPPERCASE_WITH_UNDERSCORES

# PEP8

The python style checker, pep8, can be installed using pip, it analyses codefiles with respect to the PEP8 styleguide:

*Example usage and output*

```
$ pep8 --first optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

# Agenda

- Versioning systems
- PEP8 and docstrings
- **Code documentation**

# Code documentation

As already touched above in PEP8, documenting code is important

- Inside of code: docstrings
- Outside of code: specific documents, READMEs, reports, handbooks, etc.
- Docstrings + handbook-style + sphinx = Awesome!

# Code documentation

Inside code, for this class: „sphinxy“ docstrings:

```
def public_fn_with_sphinxy_docstring(name, state=None):
    """This function does something.

    :param name: The name to use.
    :type name: str.
    :param state: Current state to be in.
    :type state: bool.
    :returns:  int -- the return code.
    :raises: AttributeError, KeyError

    """
    return 0
```

# Doc generation

Python-code + docstrings + handbook-style written text + sphinx
(learn more at https://pythonhosted.org/an_example_pypi_project/sphinx.html )

- In source folder; sphinx-quickstart
  - Set root folder to ./docs
- In docs-folder: sphinx-apidoc –o source/ ../
- Open index.rst, insert source/modules.rst to toc
- In conf.py, add the source-path to your system path
  import os, sys
  sys.path.insert(0, os.path.abspath(„..“)
- In conf.py, add globaltoc.html to html_sidebars
- Make html -> Google-chrome ./_build/html/index.html
- Make latexpdf -> evince ./_build/pdf/projectname.pdf

# Sphinx example

main.py

```
#!/usr/bin/python
'''
author: Tobias Weis
'''

from A import *
from B import *

def main():
    """
    This function does something and uses other classes and their functions
    """
    pass

if __name__ == "__main__":
    main()
```

# Sphinx example

A.py

```
#!/usr/bin/python
'''
author: Tobias Weis
'''


class A():
    """This class holds some mathematical functions"""
    def __init__(self):
        pass

    def do_something(self,x,y):
        """This function computes the sum of its arguments

        :param x: first number
        :type x: float
        :param y: second number
        :type y: float
        :returns: float -- the sum of both numbers
        """
        return x+y
```

# Sphinx example

index.rst

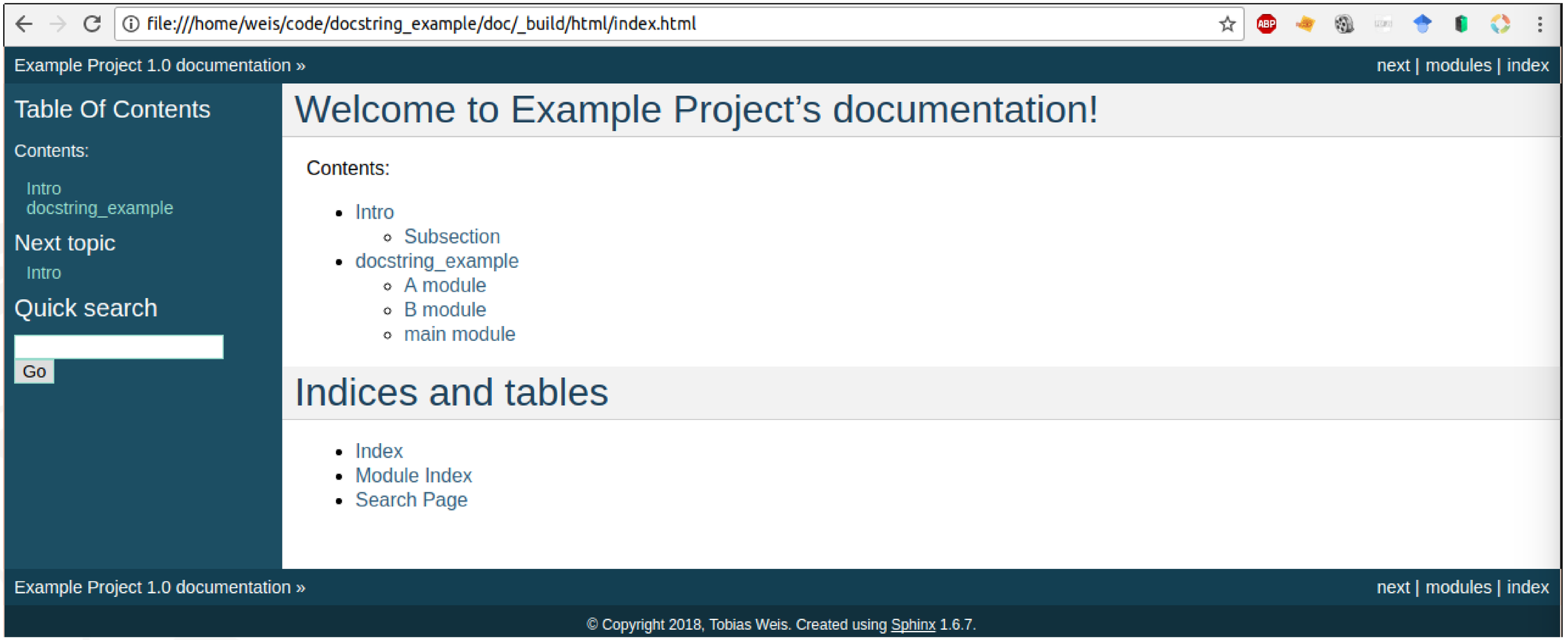
```
.. Example Project documentation master file, created by
   sphinx-quickstart on Thu Feb  8 16:17:01 2018.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to Example Project's documentation!
===========================================

.. toctree::
   :maxdepth: 2
   :caption: Contents:

   description.rst
   source/modules.rst



Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

# Sphinx example

description.rst

```
Intro
=====

This is written text (in rst), that is independent of the source code.

You could use these pages for
* a description of your project
* presenting results
* writing tutorials on how to use your code

Subsection
----------

This is a minor subsection
~
```

# Sphinx example



# Sphinx example

## Latex pdf report:

**Example Project Documentation**

***Release 1.0***

**Tobias Weis**

**Feb 08, 2018**

**CONTENTS:**





CHAPTER

**ONE**

**INTRO**

This is written text (in rst), that is independent of the source code.

You could use these pages for * a description of your project * presenting results * writing tutorials on how to use your code

### 1.1 Subsection

This is a minor subsection



CHAPTER

**TWO**

**DOCSTRING_EXAMPLE**

### 2.1 A module

author: Tobias Weis

**class** A.**A**

This class holds some mathematical functions

**do_something**(*x*, *y*)

This function computes the sum of its arguments

**Parameters**

- **x** (*float*) – first number
- **y** (*float*) – second number

**Returns** float – the sum of both numbers

### 2.2 B module

author: Tobias Weis

**class** B.**A**

This class holds some string functions

**do_something**(*x*, *y*)

This function computes the concatenation of its arguments

**Parameters**

- **x** (*str*) – first number
- **y** (*str*) – second number

**Returns** str – the concatenation of both numbers

### 2.3 main module

author: Tobias Weis

main.**main**()

This function does something and uses other classes and their functions