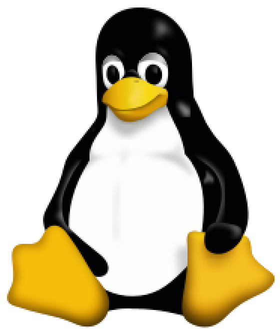
1. Introduction

Version control repositories like CVS, Subversion or Git can be a real gold mine for software developers. They contain every change to the source code including the date (the "when"), the responsible developer (the "who"), as well as little message that describes the intention (the "what") of a change.

(https://commons.wikimedia.org/wiki/File:Tux.svg)



In this notebook, we will analyze the evolution of a very famous open-source project – the Linux kernel. The Linux kernel is the heart of some Linux distributions like Debian, Ubuntu or CentOS.

We get some first insights into the work of the development efforts by

- identifying the TOP 10 contributors and
- visualizing the commits over the years.

Linus Torvalds, the (spoiler alert!) main contributor to the Linux kernel (and also the creator of Git), created a mirror of the Linux repository on GitHub (https://github.com/torvalds/linux/). It contains the complete history of kernel development for the last 13 years.

For our analysis, we will use a Git log file with the following content:

In [686]:

```
# Printing the content of git_log_excerpt.csv
# ... YOUR CODE FOR TASK 1 ...
import pandas as pd
file = open('datasets/git_log_excerpt.csv','r')
read_data = file.read()
print(read_data)
```

```
1502382966#Linus Torvalds
1501368308#Max Gurtovoy
1501625560#James Smart
1501625559#James Smart
1500568442#Martin Wilck
1502273719#Xin Long
1502278684#Nikolay Borisov
1502238384#Girish Moodalbail
1502228709#Florian Fainelli
1502223836#Jon Paul Maloy
```

In [687]:

```
%%nose

def test_listing_of_file_contents():

    # FIXME1: if student executes cell more than once, variable _i2 is then not define
d. Solution?

    #PATH = "datasets/git_log_excerpt.csv"
    # hard coded cell number: maybe a little bit fragile
    #cell_input_from_sample_code = _i2
    #assert PATH in cell_input_from_sample_code, \
    #"The file %s should be read in." % PATH

    # FIXME2: can't access the sample code cell's output here because of the use of 'pr
int'

    # test currently deactivated: too hard to create a table test case
    assert True
```

Out[687]:

```
1/1 tests passed
```

2. Reading in the dataset

The dataset was created by using the command `git log --encoding=latin-1 --pretty="%at#%aN"`. The `latin-1` encoded text output was saved in a header-less csv file. In this file, each row is a commit entry with the following information:

- `timestamp`: the time of the commit as a UNIX timestamp in seconds since 1970-01-01 00:00:00 (Git log placeholder "`%at`")
- `author`: the name of the author that performed the commit (Git log placeholder "`%aN`")

The columns are separated by the number sign `#`. The complete dataset is in the `datasets/` directory. It is a `gz`-compressed csv file named `git_log.gz`.

In [688]:

```
# Loading in the pandas module
# ... YOUR CODE FOR TASK 2 ...

# Reading in the log file
git_log = pd.read_csv( 'datasets/git_log.gz', sep='#', encoding='latin-1', header=None,
 names=['timestamp', 'author'] )

# Printing out the first 5 rows
# ... YOUR CODE FOR TASK 2 ...
git_log.head()
```

Out[688]:

	timestamp	author
0	1502826583	Linus Torvalds
1	1501749089	Adrian Hunter
2	1501749088	Adrian Hunter
3	1501882480	Kees Cook
4	1497271395	Rob Clark

In [689]:

```
%%nose


def test_is_pandas_loaded_as_pd():

    try:
        pd # throws NameError
        pd.DataFrame # throws AttributeError
    except NameError:
        assert False, "Module pandas not loaded as pd."
    except AttributeError:
        assert False, "Variable pd is used as short name for another module."


def test_is_git_log_data_frame_existing():

    try:
        # checks implicitly if git_log by catching the NameError exception
        assert isinstance(git_log, pd.DataFrame), "git_log isn't a DataFrame."

    except NameError as e:
        assert False, "Variable git_log doesn't exist."


def test_has_git_log_correct_columns():

    expected = ['timestamp', 'author']
    assert all(git_log.columns.get_values() == expected), \
        "Expected columns are %s" % expected


def test_is_logfile_content_read_in_correctly():

    correct_git_log = pd.read_csv(
        'datasets/git_log.gz',
        sep='#',
        encoding='latin-1',
        header=None,
        names=['timestamp', 'author'])

    assert correct_git_log.equals(git_log), \
        "The content of datasets/git_log.gz wasn't correctly read into git_log. Check t
he parameters of read_csv."
```

Out[689]:

```
4/4 tests passed
```

3. Getting an overview

The dataset contains the information about every single code contribution (a "commit") to the Linux kernel over the last 13 years. We'll first take a look at the number of authors and their commits to the repository.

In [690]:

```
# calculating number of commits
number_of_commits =git_log['timestamp'].size

# calculating number of authors
number_of_authors = git_log.dropna(how='any')['author'].unique().size

# printing out the results
print("%s authors committed %s code changes." % (number_of_authors, number_of_commits))
```

```
17385 authors committed 699071 code changes.
```

In [691]:

```
%%nose

def test_basic_statistics():
    assert number_of_commits == len(git_log), \
    "The number of commits should be right."
    assert number_of_authors == len(git_log['author'].dropna().unique()), \
    "The number of authors should be right."
```

Out[691]:

```
1/1 tests passed
```

4. Finding the TOP 10 contributors

There are some very important people that changed the Linux kernel very often. To see if there are any bottlenecks, we take a look at the TOP 10 authors with the most commits.

In [692]:

```
# Identifying the top 10 authors
author_commits = git_log.groupby('author').count().sort_values(by='timestamp', ascendin
g=False)
top_10_authors = author_commits.head(10)

# Listing contents of 'top_10_authors'
top_10_authors
```

Out[692]:

	timestamp
author	
Linus Torvalds	23361
David S. Miller	9106
Mark Brown	6802
Takashi Iwai	6209
Al Viro	6006
H Hartley Sweeten	5938
Ingo Molnar	5344
Mauro Carvalho Chehab	5204
Arnd Bergmann	4890
Greg Kroah-Hartman	4580

```
In [693]:
```

```
%%nose


def test_is_series_or_data_frame():

    assert isinstance(top_10_authors, pd.Series) or isinstance(top_10_authors, pd.DataF
rame), \
    "top_10_authors isn't a Series or DataFrame, but of type %s." % type(top_10_author
s)


def test_is_result_structurally_alright():

    top10 = top_10_authors.squeeze()
    # after a squeeze(), the DataFrame with one Series should be converted to a Series
    assert isinstance(top10, pd.Series), \
    "top_10_authors should only contain the data for authors and the number of commit
s."


def test_is_right_number_of_entries():

    expected_number_of_entries = 10
    assert len(top_10_authors.squeeze()) is expected_number_of_entries, \
    "The number of TOP 10 entries should be %r. Be sure to store the result into the 't
op_10_authors' variable." % expected_number_of_entries


def test_is_expected_top_author():

    expected_top_author = "Linus Torvalds"
    assert top_10_authors.squeeze().index[0] == expected_top_author, \
    "The number one contributor should be %s." % expected_top_author


def test_is_expected_top_commits():
    expected_top_commits = 23361
    assert top_10_authors.squeeze()[0] == expected_top_commits, \
    "The number of the most commits should be %r." % expected_top_commits
```

```
Out[693]:

5/5 tests passed
```

5. Wrangling the data

For our analysis, we want to visualize the contributions over time. For this, we use the information in the `timestamp` column to create a time series-based column.

```
In [694]:
```

```python
# converting the timestamp column
# ... YOUR CODE FOR TASK 5 ...
git_log['timestamp'] = pd.to_datetime(git_log['timestamp'], unit='s')
# summarizing the converted timestamp column
# ... YOUR CODE FOR TASK 5 ...
git_log.describe()
```

```
Out[694]:
```

	timestamp	author
count	699071	699070
unique	668448	17385
top	2008-09-04 05:30:19	Linus Torvalds
freq	99	23361
first	1970-01-01 00:00:01	NaN
last	2037-04-25 08:08:26	NaN

```
In [695]:
```

```python
%%nose

def test_timestamps():

    START_DATE = '1970-01-01 00:00:01'
    assert START_DATE in str(git_log['timestamp'].min()), \
    'The first timestamp should be %s.' % START_DATE

    END_DATE = '2037-04-25 08:08:26'
    assert END_DATE in str(git_log['timestamp'].max()), \
    'The last timestamp should be %s.' % END_DATE
```

```
Out[695]:

1/1 tests passed
```

6. Treating wrong timestamps

As we can see from the results above, some contributors had their operating system's time incorrectly set when they committed to the repository. We'll clean up the `timestamp` column by dropping the rows with the incorrect timestamps.

In [696]:

```
# determining the first real commit timestamp
first_commit_timestamp = git_log[git_log['author']=='Linus Torvalds'].sort_values(by='t
imestamp').head(1).reset_index(drop=True).iloc[0,0]
# determining the last sensible commit timestamp
last_commit_timestamp =  pd.to_datetime('today')

# filtering out wrong timestamps
corrected_log = git_log[(git_log['timestamp'] >= first_commit_timestamp) & (git_log['ti
mestamp'] <= last_commit_timestamp)]
# summarizing the corrected timestamp column
# ... YOUR CODE FOR TASK 6 ...
corrected_log['timestamp'].describe()
```

Out[696]:

```
count                   698569
unique                  667977
top        2008-09-04 05:30:19
freq                        99
first      2005-04-16 22:20:36
last       2017-10-03 12:57:00
Name: timestamp, dtype: object
```

In [697]:

```
%%nose

def test_corrected_timestamps():

    FIRST_REAL_COMMIT = '2005-04-16 22:20:36'
    assert FIRST_REAL_COMMIT in str(corrected_log['timestamp'].min()), \
    'The first real commit timestamp should be %s.' % FIRST_REAL_COMMIT

    LAST_REAL_COMMIT = '2017-10-03 12:57:00'
    assert LAST_REAL_COMMIT in str(corrected_log['timestamp'].max()), \
    'The last real commit timestamp should be %s.' % LAST_REAL_COMMIT
```

Out[697]:

```
1/1 tests passed
```

7. Grouping commits per year

To find out how the development activity has increased over time, we'll group the commits by year and count them up.

In [698]:

```
# Counting the no. commits per year
commits_per_year = corrected_log.groupby(
    pd.Grouper(
        key='timestamp',
        freq='AS'
        )
    ).count()

# Listing the first rows
# ... YOUR CODE FOR TASK 7 ...
commits_per_year.head()
```

Out[698]:

	author
timestamp	
2005-01-01	16229
2006-01-01	29255
2007-01-01	33759
2008-01-01	48847
2009-01-01	52572

In [699]:

```
%%nose

def test_number_of_commits_per_year():

    YEARS = 13
    assert len(commits_per_year) == YEARS, \
    'Number of years should be %s.' % YEARS


def test_new_beginning_of_git_log():

    START = '2005-01-01 00:00:00'
    assert START in str(commits_per_year.index[0]), \
    'DataFrame should start at %s' % START
```

Out[699]:

```
2/2 tests passed
```

8. Visualizing the history of Linux

Finally, we'll make a plot out of these counts to better see how the development effort on Linux has increased over the the last few years.