Data Mining using Python — code comments

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

November 29, 2017

Code comments

Random comments on code provided by students.

With thanks to Vladimir Keleshev and others for tips.

argparse?

```
import argparse
```

argparse?

```
import argparse
```

Use docopt. :-)

argparse?

```
import argparse
```

Use docopt. :-)

```
import docopt
```

http://docopt.org/

Vladimir Keleshev's video: PyCon UK 2012: Create *beautiful* command-line interfaces with Python

You will get the functionality and the documentation in one go.

Comments and a function declaration?

```
# Get comments of a subreddit, returns a list of strings
def get_subreddit_comments(self, subreddit, limit=None):
```

Comments and a function declaration?

```
# Get comments of a subreddit, returns a list of strings
def get_subreddit_comments(self, subreddit, limit=None):
```

Any particular reason for not using docstrings?

```
def get_subreddit_comments(self, subreddit, limit=None):
    """Get comments of a subreddit and return a list of strings."""
```

. . . and use Vladimir Keleskev's Python program pep257 to check docstring format convention (PEP 257 "Docstring Conventions").

Please do:

```
$ sudo pip install pep257
$ pep257 yourpythonmodule.py
```

Names

```
req = requests.get(self.video_url.format(video_id), params=params)
```

Names

```
req = requests.get(self.video_url.format(video_id), params=params)
```

The returned object from `requests.get` is a `Response` object (actually a `requests.model.Response` object).

A more appropriate name would be `response`:

```
response = requests.get(self.video_url.format(video_id), params=params)
```

Names

```
req = requests.get(self.video_url.format(video_id), params=params)
```

The returned object from `requests.get` is a `Response` object (actually a `requests.model.Response` object).

A more appropriate name would be `response`:

```
response = requests.get(self.video_url.format(video_id), params=params)
```

And what about this:

```
next_url = [n["href"] for n in r["feed"]["link"] if n["rel"] == "next"]
```

Names

```
req = requests.get(self.video_url.format(video_id), params=params)
```

The returned object from `requests.get` is a `Response` object (actually a `requests.model.Response` object).

A more appropriate name would be `response`:

```
response = requests.get(self.video_url.format(video_id), params=params)
```

And what about this:

```
next_url = [n["href"] for n in r["feed"]["link"] if n["rel"] == "next"]
```

Single character names are difficult for the reader to understand.

Single characters should perhaps only be used for indices and for abstract mathematical objects, e.g., matrix where the matrix can contain 'general' data.

More names

```
with open(WORDS_PATH,'a+') as w:
    ...
```

More names

```
with open(WORDS_PATH,'a+') as w:
    ...
```

`WORDS_PATH` is a file name not a path (or a path name).

Enumerable constants

Enumerable constants

Use `Enum` class in `enum` module from the `enum34` package.

pi

```
import math
pi = math.pi # Define pi
```

pi

```
import math
pi = math.pi # Define pi
```

What about

```
from math import pi
```

Assignment

```
words=[]
words = single_comment.split()
```

Assignment

```
words=[]
words = single_comment.split()
```

`words` is set to an empty list and then immediately overwritten!

URL and CSV

```
def get_csv_from_url(self, url):
    request = urllib2.Request(url)
    try:
        response = urllib2.urlopen(request)
        self.company_list = pandas.DataFrame({"Companies" : \
        [line for line in response.read().split("\r\n")  \
        if (line != '' and line != "Companies") ]})
        print "Fetching data from " + url
    except urllib2.HTTPError, e:
        print 'HTTPError = ' + str(e.code)
    ...
```

URL and CSV

```
def get_csv_from_url(self, url):
    request = urllib2.Request(url)
    try:
        response = urllib2.urlopen(request)
        self.company_list = pandas.DataFrame({"Companies" : \
        [line for line in response.read().split("\r\n")  \
        if (line != '' and line != "Companies") ]})
        print "Fetching data from " + url
    except urllib2.HTTPError, e:
        print 'HTTPError = ' + str(e.code)
    ...
```

Pandas `read_csv` will also do URLs:

```
def get_company_list_from_url(self, url):
    self.company_list = pandas.read_csv(url)
```

Also note: issues of exception handling, logging and documentation.

Sorting

```
def SortList(l):
    ...

def FindClosestValue(v,l):
    ...

...
SortList(a)
VaIn = FindClosestValue(int(Value), a)
```

Sorting

```
def SortList(l):
    ...

def FindClosestValue(v,l):
    ...


...
SortList(a)
VaIn = FindClosestValue(int(Value), a)
```

Reinventing the wheel? Google: "find closest value in list python" yields several suggestions, if unsorted:

```
min(my_list, key=lambda x: abs(x - my_number))
```

and if sorted:

```
from bisect import bisect_left
```

Sorting 2

Returning the key associated with the maximum value in a dict:

```
return sorted(likelihoods, key=likelihoods.get,
              reverse=True)[0]
```

Sorting 2

Returning the key associated with the maximum value in a dict:

```
return sorted(likelihoods, key=likelihoods.get,
              reverse=True)[0]
```

Sorting is $O(N \log N)$ while finding the maximum is $O(N)$, so this should (hopefully) be faster:

```
return max(likelihoods, key=likelihoods.get)
```

Word tokenization

Splitting a string into a list of words and processing each word:

```
for word in re.split("\W+", sentence)
```

Word tokenization

Splitting a string into a list of words and processing each word:

```
for word in re.split("\W+", sentence)
```

Maybe \W+ is not necessarily particularly good?

Comparison with NLTK's word tokenizer:

```
>>> import nltk, re
>>> sentence = "In a well-behaved manner"
>>> [word for word in re.split("\W+", sentence)]
['In', 'a', 'well', 'behaved', 'manner']
>>> nltk.word_tokenize(sentence)
['In', 'a', 'well-behaved', 'manner']
```

POS-tagging

```
import re
sentences = """Sometimes it may be good to take a close look at the
documentation. Sometimes you will get surprised."""
words = [word for sentence in nltk.sent_tokenize(sentences)
              for word in re.split('\W+', sentence)]
nltk.pos_tag(words)
```

POS-tagging

```
import re
sentences = """Sometimes it may be good to take a close look at the
documentation. Sometimes you will get surprised."""
words = [word for sentence in nltk.sent_tokenize(sentences)
              for word in re.split('\W+', sentence)]
nltk.pos_tag(words)
```

```
>>> nltk.pos_tag(words)[12:15]
[('documentation', 'NN'), ('', 'NN'), ('Sometimes', 'NNP')]

>>> map(lambda s: nltk.pos_tag(nltk.word_tokenize(s)), nltk.sent_tokenize(sentences))
[[('Sometimes', 'RB'), ('it', 'PRP'), ('may', 'MD'), ('be', 'VB'),
('good', 'JJ'), ('to', 'TO'), ('take', 'VB'), ('a', 'DT'), ('close',
'JJ'), ('look', 'NN'), ('at', 'IN'), ('the', 'DT'), ('documentation',
'NN'), ('.', '.')], [('Sometimes', 'RB'), ('you', 'PRP'), ('will',
'MD'), ('get', 'VB'), ('surprised', 'VBN'), ('.', '.')]]
```

Note the period which is tokenized. "Sometimes" looks like a proper noun because of the initial capital letter.

Exception

```
class LongMessageException(Exception):
    pass
```

Exception

```
class LongMessageException(Exception):
    pass
```

Yes, we can! It is possible to define you own exceptions!

Exception 2

```
if self.db is None:
    raise Exception('No database engine attached to this instan
```

Exception 2

```
if self.db is None:
    raise Exception('No database engine attached to this instan
```

Derive your own class so the user of your module can distinguish between errors.

Exception 3

```
try:
    if data["feed"]["entry"]:
        for item in data["feed"]["entry"]:
            return_comments.append(item["content"])
except KeyError:
    sys.exc_info()[0]
```

Exception 3

```
try:
    if data["feed"]["entry"]:
        for item in data["feed"]["entry"]:
            return_comments.append(item["content"])
except KeyError:
    sys.exc_info()[0]
```

`sys.exc_info()[0]` just ignores the exception. Either you should pass it, log it or actually handle it, here using the `logging` module:

```
import logging

try:
    if data["feed"]["entry"]:
        for item in data["feed"]["entry"]:
            return_comments.append(item["content"])
except KeyError:
    logging.exception("Unhandled feed item")
```

Trying to import a url library?

```
try:
    import urllib3 as urllib
except ImportError:
    try:
        import urllib2 as urllib
    except ImportError:
        import urllib as urllib
```

Trying to import a url library?

```
try:
    import urllib3 as urllib
except ImportError:
    try:
        import urllib2 as urllib
    except ImportError:
        import urllib as urllib
```

This is a silly example. This code was from the lecture slides and was meant to be a demonstration (I thought I was paedagodic). Just import one of them. And urllib and urllib2 is in PSL so it is not likely that you cannot import them.

Just write:

```
import urllib
```

Import failing?

```
try:
    import BeautifulSoup as bs
except ImportError, message:
    print "There was an error loading BeautifulSoup: %s" % message
```

Import failing?

```
try:
    import BeautifulSoup as bs
except ImportError, message:
    print "There was an error loading BeautifulSoup: %s" % message
```

But ehhh. . . you are using BeautifulSoup further down in the code so it will fail then and then raise an exception that is difficult to understand.

Globbing import

```
from youtube import *
```

Globbing import

```
from youtube import *
```

It is usually considered good style to only import the names you need to avoid "polluting" your name space.

Better:

```
import youtube
```

alternatively:

```
from youtube import YouTubeScraper
```

Importing files in a directory tree

```
import sys
sys.path.append('theproject/lib')
sys.path.append('theproject/view_objects')
sys.path.append('theproject/models')
from user_builder import UserBuilder
```

(`user_builder` is somewhere in the directory tree)

Importing files in a directory tree

```
import sys
sys.path.append('theproject/lib')
sys.path.append('theproject/view_objects')
sys.path.append('theproject/models')
from user_builder import UserBuilder
```

(`user_builder` is somewhere in the directory tree)

It is better to define `__init__.py` files in each directory containing the imports with the names that needs to be exported.

Function declaration

```
def __fetch_page_local(self, course_id, course_dir):
```

Function declaration

```
def __fetch_page_local(self, course_id, course_dir):
```

Standard naming (for "public" method) (Beazley and Jones, 2013):

```
def fetch_page_local(self, course_id, course_dir):
```

Standard naming (for "internal" method):

```
def _fetch_page_local(self, course_id, course_dir):
```

Standard naming (for "internal" method with name mangling. Is that what you want? Or have you been coding too much in Java?):

```
def __fetch_page_local(self, course_id, course_dir):
```

Making sense of `__repr__`

```
class CommentSentiment(base):
    ...
    def __repr__(self):
        return self.positive
```

Making sense of `__repr__`

```
class CommentSentiment(base):
    ...
    def __repr__(self):
        return self.positive
```

The `__repr__` should present something usefull for the developer that uses the class, e.g., its name! Here only the value of an attribute is printed.

Vladimir Keleshev's suggestion:

```
def __repr__(self):
    return '%s(id=%r, video_id=%r, positive=%r)' % (
        self.__class__.__name__, self.id, self.video_id, self.positive)
```

This will print out

```
>>> comment_sentiment = CommentSentiment(video_id=12, positive=True)
>>> comment_sentiment
CommentSentiment(id=23, video_id=12, positive=True)
```

Strings?

```
userName = str(user[u'name'].encode('ascii', 'ignore'))
```

Strings?

```
userName = str(user[u'name'].encode('ascii', 'ignore'))
```

Ehhh... Why not just `user['name']`? What does `str` do? Redundant!?

Strings?

```
userName = str(user[u'name'].encode('ascii', 'ignore'))
```

Ehhh. . . Why not just `user['name']`? What does `str` do? Redundant!?

You really need to make sure you understand the distinction between ASCII byte strings, UTF-8 byte strings and Unicode strings.

You should consider for each variable in your program what is the most appropriate type and when it makes sense to convert it.

Usually: On the web data comes as UTF-8 byte strings that you would need in Python 2 to convert to Unicode strings. After you have done the processing in Unicode you may what to write out the results. This will mostly be in UTF-8.

See slides on encoding.

A URL?

```
baselink = ("http://www.kurser.dtu.dk/search.aspx"
                          "?YearGroup=2013-2014"
                          "&btnSearch=Search"
                          "&menulanguage=en-GB"
                          "&txtSearchKeyword=%s")
```

A URL?

```
baselink = ("http://www.kurser.dtu.dk/search.aspx"
                            "?YearGroup=2013-2014"
                            "&btnSearch=Search"
                            "&menulanguage=en-GB"
                            "&txtSearchKeyword=%s")
```

"2013-2014" looks like something likely to change in the future. Maybe it would be better to make it a parameter.

Also note that the `get` method in the `requests` module has the `param` input argument, which might be better for URL parameters.

“Constants”

```
print(c.fetch_comments("RiQYcw-u18I"))
```

“Constants”

```
print(c.fetch_comments("RiQYcw-u18I"))
```

Don’t put such “pseudoconstants” in a reuseable module, — unless they are examples.

Put them in data files, configuration files or as script input arguments.

Configuration

Use configuration files for 'changing constants', e.g., API keys.

There are two modules `config` and `ConfigParser`/`configparser`. `configparser` (Python 3) can parse portable windows-like configuration file like:

```
[requests]
user_agent = fnielsenbot
from = faan@dtu.dk

[twitter]
consumer_key =  HFDFDF45454HJHJH
consumer_secret = kjhkjsdhfksjdfhf3434jhjhjh34h3
access_token =  kjh234kj2h34
access_secret = kj23h4k2h34k23h4
```

Constructing a path

```
FILE_PATH = "%s" + os.sep + "%s.txt"
current_file_path = FILE_PATH % (directory, filename)
```

Constructing a path

```
FILE_PATH = "%s" + os.sep + "%s.txt"
current_file_path = FILE_PATH % (directory, filename)
```

Yes! `os.sep` is file independent.

Constructing a path

```
FILE_PATH = "%s" + os.sep + "%s.txt"
current_file_path = FILE_PATH % (directory, filename)
```

Yes! `os.sep` is file independent.

But so is `os.path.join`:

```
from os.path import join
join(directory, filename + '.txt')
```

Building a URL

```
request_url = self.BASE_URL + \
    '?' + self.PARAM_DEV_KEY + \
    '=' + self.developer_key + \
    '&' + self.PARAM_PER_PAGE + \
    '=' + str(amount) + \
    '&' + self.PARAM_KEYWORDS + \
    '=' + ','.join(keywords)
```

Building a URL

```
request_url = self.BASE_URL + \
    '?' + self.PARAM_DEV_KEY + \
    '=' + self.developer_key + \
    '&' + self.PARAM_PER_PAGE + \
    '=' + str(amount) + \
    '&' + self.PARAM_KEYWORDS + \
    '=' + ','.join(keywords)
```

Use instead the params keyword in the `requests.get` function, as special characters need to be escaped in URL, e.g.,

```
>>> response = requests.get('http://www.dtu.dk', params={'q': u'æ ø å'})
>>> response.url
u'http://www.dtu.dk/?q=%C3%A6+%C3%B8+%C3%A5'
```

The double break out

```
import Image

image = Image.open("/usr/lib/libreoffice/program/about.png")
message = "Hello, world"

mat = image.load()
x,y = image.size
count = 0
done = False
for i in range(x):
    for j in range(y):
        mat[i,j] = (mat[i,j][2], mat[i,j][0], mat[i,j][1])
        count = count + 1
        if count == len(message):
            done = True
            break
    if done:
        break
```

(modified from the original)

The double break out

```
import Image
from itertools import product

image = Image.open("/usr/lib/libreoffice/program/about.png")
message = "Hello, world"

mat = image.load()
for count, (i, j) in enumerate(product(*map(range, image.size))):
    mat[i,j] = (mat[i,j][2], mat[i,j][0], mat[i,j][1])
    if count == len(message):
        break
```

Fewer lines avoiding the double break, but less readable perhaps? So not necessarily better? In Python itertools module there are lots of interesting functions for iterators. Take a look.

Note that `count = count + 1` can also be written `count += 1`

Not everything you read on the Internet ...

```
def remove_html_markup(html_string):
    tag = False
    quote = False
    out = ""
    for char in html_string:
        if char == "<" and not quote:
            tag = True
        elif char == '>' and not quote:
            tag = False
        elif (char == '"' or char == "'") and tag:
            quote = not quote
        elif not tag:
            out = out + char
    return out
```

"Borrowed from: http://stackoverflow.com/a/14464381/368379"

Not everything you read on the Internet . . .

```
def remove_html_markup(html_string):
    tag = False
    quote = False
    out = ""
    for char in html_string:
        if char == "<" and not quote:
            tag = True
        elif char == '>' and not quote:
            tag = False
        elif (char == '"' or char == "'") and tag:
            quote = not quote
        elif not tag:
            out = out + char
    return out
```

"Borrowed from: http://stackoverflow.com/a/14464381/368379"

The Stackoverflow website is a good resource, but please be critical about the suggested solutions.

Not everything you read on the Internet ...

```
def remove_html_markup(html_string):
    tag = False
    quote = False
    out = ""
    for char in html_string:
        if char == "<" and not quote:
            tag = True
        elif char == '>' and not quote:
            tag = False
        elif (char == '"' or char == "'") and tag:
            quote = not quote
        elif not tag:
            out = out + char
    return out
```

```
>>> s = """<a title="DTU's homepage" href="http://dtu.dk">DTU</a>"""
>>> remove_html_markup(s)
''
```

Not everything you read on the Internet ...

```
def remove_html_markup(html_string):
    tag = False
    quote = False
    out = ""
    for char in html_string:
        if char == "<" and not quote:
            tag = True
        elif char == '>' and not quote:
            tag = False
        elif (char == '"' or char == "'") and tag:
            quote = not quote
        elif not tag:
            out = out + char
    return out
```

You got BeautifulSoup, NLTK, etc., e.g.,

```
>>> import nltk
>>> nltk.clean_html(s)
'DTU'
```

Documentation

```
def update(self):
    """
    Execute an update where the currently selected course from the history is displayed an
    """
```

Documentation

```
def update(self):
    """
    Execute an update where the currently selected course from the history is displayed an
    """
```

Docstrings should be considered to be read on an 80-column terminal:

```
def update(self):
    """Update the currently selected courses.

    Update the currently selected course from the history is displayed
    and the buttons for back and forward are up.

    """
```

Style Guide for Python Code: "For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters." (van Rossum et al., 2013).

Run Vladimir Keleshev's pep257 program on your code!

For loop with counter

```
tcount = 0
for t in stream:
    if tcount >= 1000:
        break
    dump(t)
    tcount += 1
```

For loop with counter

```
tcount = 0
for t in stream:
    if tcount >= 1000:
        break
    dump(t)
    tcount += 1
```

Please at least enumerate:

```
for tcount, t in enumerate(stream):
    if tcount >= 1000:
        break
    dump(t)
```

For loop with counter

```
tcount = 0
for t in stream:
    if tcount >= 1000:
        break
    dump(t)
    tcount += 1
```

Please at least enumerate:

```
for tcount, t in enumerate(stream):
    if tcount >= 1000:
        break
    dump(t)
```

and don't use that short variable names, — like "t"!

For loop with counter 2

`len` together with `for` is often suspicious. Made-up example:

```
from nltk.corpus import shakespeare

tokens = shakespeare.words('hamlet.xml')
words = []
for n in range(len(tokens)):
    if tokens[n].isalpha():
        words.append(tokens[n].lower())
```

For loop with counter 2

`len` together with `for` is often suspicious. Made-up example:

```
from nltk.corpus import shakespeare

tokens = shakespeare.words('hamlet.xml')
words = []
for n in range(len(tokens)):
    if tokens[n].isalpha():
        words.append(tokens[n].lower())
```

Better and cleaner:

```
tokens = shakespeare.words('hamlet.xml')
words = []
for token in tokens:
    if token.isalpha():
        words.append(token.lower())
```

For loop with counter 2

`len` together with `for` is usually suspicious. Made-up example:

```
from nltk.corpus import shakespeare

tokens = shakespeare.words('hamlet.xml')
words = []
for n in range(len(tokens)):
    if tokens[n].isalpha():
        words.append(tokens[n].lower())
```

Better and cleaner:

```
tokens = shakespeare.words('hamlet.xml')
for token in tokens:
    if tokens.isalpha():
        words.append(tokens.lower())
```

Or with a generator comprehension (alternatively list comprehension):

```
tokens = shakespeare.words('hamlet.xml')
words = (token.lower() for token in tokens if token.isalpha())
```

For loop with counter 3

```
for c_no, value in enumerate(a_list):
    # more code here.
    c_no += 1
```

For loop with counter 3

```
for c_no, value in enumerate(a_list):
    # more code here.
    c_no += 1
```

The first variable `c_no` is increase automatically. Just write:

```
for c_no, value in enumerate(a_list):
    # more code here.
```

Caching results

You want to cache results that takes long time to fetch or compute:

```
def get_all_comments(self):
    self.comments = computation_that_takes_long_time()
    return self.comments

def get_all_comments_from_last_call(self):
    return self.comments
```

Caching results

You want to cache results that takes long time to fetch or compute:

```
def get_all_comments(self):
    self.comments = computation_that_takes_long_time()
    return self.comments

def get_all_comments_from_last_call(self):
    return self.comments
```

This can be done more elegantly with a lazy property:

```
import lazy

@lazy
def all_comments(self):
    comments = computation_that_takes_long_time()
    return comments
```

All those Python versions . . . !

```
import sysconfig
if float(sysconfig.get_python_version()) < 3.1:
    exit('your version of python is below 3.1')
```

All those Python versions . . . !

```
import sysconfig
if float(sysconfig.get_python_version()) < 3.1:
    exit('your version of python is below 3.1')
```

Are there any particular reason why it shouldn't work with previous versions of Python?

Try install tox that will allow you to test your code with multiple versions of Python.

. . . and please be careful with handling version numbering: conversion to float will not work with, e.g., "3.2.3". See `pkg_resources.parse_version`.

Iterable

```
for kursus in iter(kursusInfo.keys()):
    # Here is some extra code
```

Iterable

```
for kursus in iter(kursusInfo.keys()):
    # Here is some extra code
```

Dictionary keys are already iterable

```
for kursus in kursusInfo.keys():
    # Here is some extra code
```

Iterable

```
for kursus in iter(kursusInfo.keys()):
    # Here is some extra code
```

Dictionary keys are already iterable

```
for kursus in kursusInfo.keys():
    # Here is some extra code
```

. . . and you can actually make it yet shorter.

Iterable

```
for kursus in iter(kursusInfo.keys()):
    # Here is some extra code
```

Dictionary keys are already iterable

```
for kursus in kursusInfo.keys():
    # Here is some extra code
```

. . . and you can actually make it yet shorter.

```
for kursus in kursusInfo:
    # Here is some extra code
```

Getting those items

```
endTime = firstData["candles"][-1].__getitem__("time")
```

Getting those items

```
endTime = firstData["candles"][-1].__getitem__("time")
```

There is no need to use magic methods directly `.__getitem__("time")` is the same as `["time"]`

```
endTime = firstData["candles"][-1]["time"]
```

Checkin of .pyc files

```
$ git add *.pyc
```

Checkin of .pyc files

```
$ git add *.pyc
```

*.pyc files are byte code files generated from *.py files. Do not check these files into the revision control system.

Put them in `.gitignore`:

```
*.pyc
```

Checkin of .pyc files

```
$ git add *.pyc
```

*.pyc files are byte code files generated from *.py files. Do not check these files into the revision control system.

Put them in `.gitignore` together with others

```
*.pyc
.tox
__pycache__
```

And many more, see an example of .gitignore.

I18N

Module with a docstring

```
"""
@author: Finn Årup Nielsen
"""

# Code below.
```

I18N

Module with a docstring

```
"""
@author: Finn Årup Nielsen
"""

# Code below.
```

Python 2 is by default ASCII, — not UTF-8.

```
# -*- coding: utf-8 -*-
u"""
@author: Finn Årup Nielsen
"""

# Code below.
```

I18N

Module with a docstring

```
"""
@author: Finn Årup Nielsen

# Code below.
"""
```

Note that you might run into Python 2/3 output encoding problem with

```
>>> import mymodule
>>> help(mymodule)
...
UnicodeEncodeError: 'ascii' codec can't encode character
u'\xc5' in position 68: ordinal not in range(128)
```

If the user session is in an ascii session an encoding exception is raised.

Opening a file with with

```
comments_filename = 'somekindoffilename.txt'
words_filename = 'anotherfilename.txt'
text = 'Some kind of text goes here.'
with open(comments_filename, 'a+') as c:
    #Convert text to str and remove newlines
    single_comment = str(text).replace("\n", " ")
    single_comment+='\n'
    c.write(single_comment)
    with open(words_filename,'a+') as w:
        words=[]
        words = single_comment.split()
        for word in words:
            single_word = str(word)
            single_word+='\n'
            w.write(single_word)


w.close()
c.close()
```

Opening a file with with

```
comments_filename = 'somekindoffilename.txt'
words_filename = 'anotherfilename.txt'
text = 'Some kind of text goes here.'
with open(comments_filename, 'a+') as c:
    #Convert text to str and remove newlines
    single_comment = str(text).replace("\n", " ")
    single_comment+='\n'
    c.write(single_comment)
    with open(words_filename,'a+') as w:
        words=[]
        words = single_comment.split()
        for word in words:
            single_word = str(word)
            single_word+='\n'
            w.write(single_word)
```

File identifiers are already closed when the with block has ended.

Opening a file with with

```
comments_filename = 'somekindoffilename.txt'
words_filename = 'anotherfilename.txt'
text = 'Some kind of text goes here.'
with open(comments_filename, 'a+') as c:
    #Convert text to str and remove newlines
    single_comment = str(text).replace("\n", " ")
    single_comment+='\n'
    c.write(single_comment)
    with open(words_filename,'a+') as w:
        words = single_comment.split()
        for word in words:
            single_word = str(word)
            single_word+='\n'
            w.write(single_word)
```

File identifiers are already closed when the with block has ended. Erase redundant assignment.

Opening a file with with

```
comments_filename = 'somekindoffilename.txt'
words_filename = 'anotherfilename.txt'
text = 'Some kind of text goes here.'
with open(comments_filename, 'a+') as c:
    single_comment = text.replace("\n", " ")
    single_comment+='\n'
    c.write(single_comment)
    with open(words_filename,'a+') as w:
        words = single_comment.split()
        for word in words:
            single_word = word
            single_word+='\n'
            w.write(single_word)
```

File identifiers are already closed when the with block has ended. Erase redundant assignment. No need to convert to `str`.

Opening a file with with

```
comments_filename = 'somekindoffilename.txt'
words_filename = 'anotherfilename.txt'
text = 'Some kind of text goes here.'
with open(comments_filename, 'a+') as c:
    single_comment = text.replace("\n", " ")
    c.write(single_comment + '\n')
    with open(words_filename,'a+') as w:
        words = single_comment.split()
        for word in words:
            w.write(word + '\n')
```

File identifiers are already closed when the with block has ended. Erase redundant assignment. No need to convert to `str`. Simplifying.

Opening a file with with

```
comments_filename = 'somekindoffilename.txt'
words_filename = 'anotherfilename.txt'
text = 'Some kind of text goes here.'
with open(comments_filename, 'a+') as comments_file, \
        open(words_filename,'a+') as words_file:
    single_comment = text.replace("\n", " ")
    comments_file.write(single_comment + '\n')
    words = single_comment.split()
    for word in words:
        words_file.write(word + '\n')
```

File identifiers are already closed when the with block has ended. Erase redundant assignment. No need to convert to `str`. Simplifying. Multiple file openings with with.

Opening a file with with

```
comments_filename = 'somekindoffilename.txt'
words_filename = 'anotherfilename.txt'
text = 'Some kind of text goes here.'
with open(comments_filename, 'a+') as f:
    single_comment = text.replace("\n", " ")
    f.write(single_comment + '\n')
with open(words_filename,'a+') as f:
    words = text.split()
    for word in words:
        f.write(word + '\n')
```

File identifiers are already closed when the with block has ended. Erase redundant assignment. No need to convert to `str`. Simplifying. Multiple file openings with with. Alternatively: Split the with blocks.

Splitting words

```
text = 'Some kind of text goes here.'
words = text.split()
```

Splitting words

```
text = 'Some kind of text goes here.'
words = text.split()
```

You get a word with a dot: `here.` because of split on whitespaces.

Splitting words

```
text = 'Some kind of text goes here.'
words = text.split()
```

You get a word with a dot: `here.` because of split on whitespaces.

Write a test!

Splitting words

```
def split(text):
    words = text.split()
    return words

def test_split():
    text = 'Some kind of text goes here.'
    assert ['Some', 'kind', 'of', 'text', 'goes', 'here'] == split(text)
```

Splitting words

```
def split(text):
    words = text.split()
    return words

def test_split():
    text = 'Some kind of text goes here.'
    assert ['Some', 'kind', 'of', 'text', 'goes', 'here'] == split(text)
```

Test the module with py.test:

```
$ py.test yourmodulewithtestfunctions.py
```

Splitting words

```
def split(text):
    words = text.split()
    return words

def test_split():
    text = 'Some kind of text goes here.'
    assert ['Some', 'kind', 'of', 'text', 'goes', 'here'] == split(text)
```

Test the module with py.test

```
$ py.test yourmodulewithtestfunctions.py
    def test_split():
>       assert ['Some', 'kind', 'of', 'text', 'goes', 'here'] == split(text
E       assert ['Some', 'kin...goes', 'here'] == ['Some', 'kind...oes', 'he
E         At index 5 diff: 'here' != 'here.'
E         Use -v to get the full diff
```

Splitting words

```
from nltk import sent_tokenize, word_tokenize

def split(text):
    words = [word for sentence in sent_tokenize(text)
                  for word in word_tokenize(sentence)]
    return words

def test_split():
    text = 'Some kind of text goes here.'
    assert ['Some', 'kind', 'of', 'text', 'goes', 'here'] == split(text)
```

Test the module with py.test

```
$ py.test yourmodulewithtestfunctions.py
```

Splitting words

```
from nltk import sent_tokenize, word_tokenize

def split(text):
    words = [word for sentence in sent_tokenize(text)
                  for word in word_tokenize(sentence)]
    return words

def test_split():
    text = 'Some kind of text goes here.'
    assert ['Some', 'kind', 'of', 'text', 'goes', 'here'] == split(text)
```

Test the module with py.test

```
$ py.test yourmodulewithtestfunctions.py
    def test_split():
        text = 'Some kind of text goes here.'
>       assert ['Some', 'kind', 'of', 'text', 'goes', 'here'] == split(text
E       assert ['Some', 'kin...goes', 'here'] == ['Some', 'kind..., 'here',
E         Right contains more items, first extra item: '.'
E         Use -v to get the full diff
```

Splitting words

```
from nltk import sent_tokenize, word_tokenize

def split(text):
    words = [word for sentence in sent_tokenize(text)
                  for word in word_tokenize(sentence)
                  if word.isalpha()]
    return words

def test_split():
    text = 'Some kind of text goes here.'
    assert ['Some', 'kind', 'of', 'text', 'goes', 'here'] == split(text)
```

Test the module with py.test

```
$ py.test yourmodulewithtestfunctions.py
```

Splitting words

```
from nltk import sent_tokenize, word_tokenize

def split(text):
    words = [word for sentence in sent_tokenize(text)
                  for word in word_tokenize(sentence)
                  if word.isalpha()]
    return words

def test_split():
    text = 'Some kind of text goes here.'
    assert ['Some', 'kind', 'of', 'text', 'goes', 'here'] == split(text)
```

Test the module with py.test

```
$ py.test yourmodulewithtestfunctions.py
```

Success!

The example continues . . .

What about lower/uppercase case?

What about issues of Unicode/UTF-8?

Should the files really be opened for each comment?

Should individual words really be written one at a time to a file?

A sentiment analysis function

```
# pylint: disable = fixme, line-too-long
#-----------------------------------------------------------------
def afinn(text):
    '''
        AFINN is a list of English words rated for valence with an integer

        This method uses this AFINn list to find the sentiment score of a t

        Parameters
        ----------
        text : Text
            A tweet text

        Returns
        -------
        sum :
            A sentiment score based on the input text
    '''
    afinn = dict(map(lambda(k, v): (k, int(v)), [line.split('\t') for line
    return sum(map(lambda word: afinn.get(word, 0), text.lower().split()))
```

A sentiment analysis function

```
def afinn(text):
    '''
        AFINN is a list of English words rated for valence with an integer

        This method uses this AFINn list to find the sentiment score of a t

        Parameters
        ----------
        text : Text
            A tweet text

        Returns
        -------
        sum :
            A sentiment score based on the input text
    '''
    afinn = dict(map(lambda(k, v): (k, int(v)), [line.split('\t') for line
    return sum(map(lambda word: afinn.get(word, 0), text.lower().split()))
```

If the lines are too long they are too long.

A sentiment analysis function

```
def afinn(text):
    """Return sentiment analysis score for text.

    The AFINN word list is used to score the text. The sum of word
    valences are returned.

    Parameters
    ----------
    text : str
        A tweet text

    Returns
    -------
    sum : int
        A sentiment score based on the input text
    """
    afinn = dict(map(lambda(k, v): (k, int(v)), [line.split('\t') for line
    return sum(map(lambda word: afinn.get(word, 0), text.lower().split()))
```

PEP 257 fixes: Proper types for input and output, short headline, correct indentation. Note extra space need. And perhaps 'References'.

A sentiment analysis function

A closer look on the actual itself:

```
def afinn(text):
    afinn = dict(map(lambda(k, v): (k, int(v)), [line.split('\t') for line
    return sum(map(lambda word: afinn.get(word, 0), text.lower().split()))
```

What is wrong?

A sentiment analysis function

A closer look on the actual itself:

```
def afinn(text):
    afinn = dict(map(lambda(k, v): (k, int(v)), [line.split('\t') for line
    return sum(map(lambda word: afinn.get(word, 0), text.lower().split()))
```

What is wrong?

The word list is read each time the function is called. That's slow.

The word tokenization is bad. It splits on whitespace.

UTF-8 encoded file is not handled.

Style: The lines are too long and variable names too short.

The location of the file is hardcoded (the filename cannot be seen because the line is too long).

A sentiment analysis function

Avoid reading the word list file multiple times:

```
class Afinn(object):
    def __init__(self):
        self.load_wordlist()

    def load_wordlist(self, filename='AFINN-111.txt'):
        self.wordlist = ...

    def score(self, text):
        return ...


afinn = Afinn()
afinn.score('This bad text should be sentiment analyzed')
afinn.score('This good text should be sentiment analyzed')
```

Tools to help

Install flake8 with plugins flake8-docstrings, flake8-import-order and pep8-naming to perform style checking:

```
$ flake8 yourmoduledirectory
```

It catches, e.g., naming issue, unused imports, missing documentation or documentation in the wrong format.

`pylint` will catch, e.g., unused arguments.

Install `py.test` and write test functions and then:

```
$ py.test yourmoduledirectory
```

Use Numpy document convention, A Guide to NumPy/SciPy Documentation. See example.py for use. "Examples" section should be doctested.

Testing

Ran both py.test and nosetests with the result:

“All scripts returned ‘Ran 0 tests in 0.000s’ ”

Machine learning

Splitting the data set

When you evaluate the performance of a machine learning algorithm that has been trained/estimated/optimized, then you need to evaluate the performance (e.g., mean square error, accuracy, precision, . . .) on an independent data set to get unbiased results, — or you too optimistic results.

Splitting the data set

When you evaluate the performance of a machine learning algorithm that has been trained/estimated/optimized, then you need to evaluate the performance (e.g., mean square error, accuracy, precision, . . .) on an independent data set to get unbiased results, — or you too optimistic results.

So split the data into a training and a test set, — and perhaps a development set (part of the training set used test, e.g., hyperparameters).

The split can, e.g., be split-half or cross-validation.

Splitting the data set

When you evaluate the performance of a machine learning algorithm that has been trained/estimated/optimized, then you need to evaluate the performance (e.g., mean square error, accuracy, precision, . . .) on an independent data set to get unbiased results, — or you too optimistic results.

So split the data into a training and a test set, — and perhaps a development set (part of the training set used test, e.g., hyperparameters).

The split can, e.g., be split-half or cross-validation.

It is ok to evaluate the performance on the training test: This will usually give an upper bound on the performance of your model, but do not regard this value as relevant for the final performance of the system.

Continuous, ordered and categorical data

If you have a continuous variable as input or output of the model, then it is usually not a good idea to model this as a categorical value.

Default NLTK naïve Bayes classifier handles categorical input and output values and not continuous, so if you have a problem with continous variables do not use that default classifier.

There are amble of different methods, e.g., in sklearn and statsmodels for modeling both continuous and categorical values.

Continuous, ordered and categorical data

This is very wrong:

```
>>> import nltk
>>> classifier = nltk.NaiveBayesClassifier.train([({'a': 1.0}, 'pos'),
                                                  ({'a': -2.0}, 'neg')])
>>> classifier.prob_classify({'a': 1.5}).prob('pos')
0.5
```

This is an example of how it can be done:

```
import pandas as pd
import statsmodels.formula.api as smf
data = pd.DataFrame([{'Value': 1.0, 'Class': 'pos'},
                     {'Value': -2.0, 'Class': 'neg'}])
data['Classint'] = (data['Class'] == 'pos').astype(float)
results = smf.ols('Classint ~ Value', data=data).fit()
```

Example use:

```
>>> results.predict({'Value': 1.5})
array([ 1.16666667])
```

Labels as features!?

Comments from social media (Twitter, YouTube, . . .) can be downloaded and annotated for sentiment automatically using the happy and sad emoticons as labels.

Labels as features!?

Comments from social media (Twitter, YouTube, . . .) can be downloaded and annotated for sentiment automatically using the happy and sad emoticons as labels.

This is a good idea!

Labels as features!?

Comments from social media (Twitter, YouTube, ...) can be downloaded and annotated for sentiment automatically using the happy and sad emoticons as labels.

This is a good idea!

But then do not use the specific emoticons as features. You will very likely get an all too optimistic performance meaure!

When training a machine learning classifier do not use labels as features!

sklearn

```
def sklearn_classifier_Kfold_score(features,targets):
    # gnb = GaussianNB()
    svm = SVC()
```

sklearn

```
def sklearn_classifier_Kfold_score(features,targets):
    # gnb = GaussianNB()
    svm = SVC()
```

If you want to try out several classifier it is possible to parametrize the classifier.

There is a nice scikit-learn example, `plot_classifier_comparison.py`, where multiple classifiers are used on multiple data sets.

To be continued

References

Beazley, D. and Jones, B. K. (2013). *Python Cookbook*. O'Reilly, Sebastopol, third edition.

van Rossum, G., Warsaw, B., and Coglan, N. (2013). Style guide for python code. Python Enhancement Proposals 8, Python Software Foundation. http://www.python.org/dev/peps/pep-0008/.