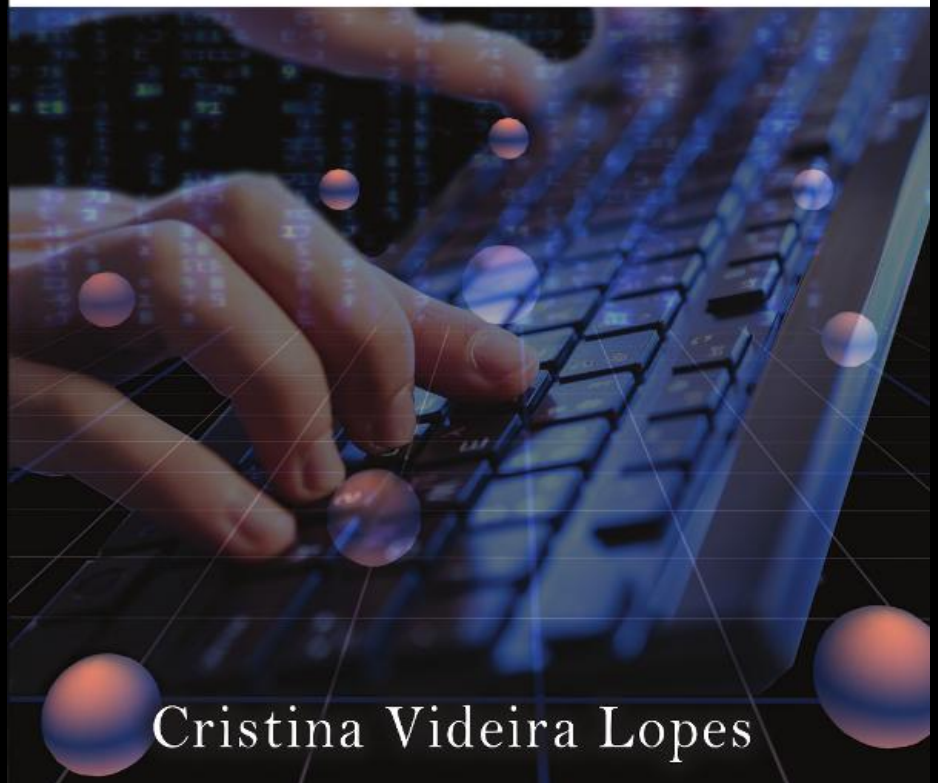
Exercises in Programming Style



Cristina Videira Lopes

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Art History, Simplified

Rules and constraints in software construction

PROGRAMMING STYLES

Programming Styles

- Ways of expressing tasks
- Exist and recur at all scales
- Frozen in Programming Languages

Why Are Styles Important?

- Basic frames of reference for solutions
  - Like scaffolding
- Common vocabularies
  - It's a cultural thing
- Some better than others
  - Depending on many objectives

Programming Styles

How do you teach this?

Raymond Queneau

Queneau's Exercises in Style

- Metaphor
- Surprises
- Dream
- Prognostication
- Hesitation
- Precision
- Negativities
- Asides
- Anagrams
- Logical analysis
- Past
- Present
- …
- (99)

Queneau's "Styles"

- Constraints

- "A Void" (La Disparition) by Georges Perec
  - No letter 'e'

Exercises in Programming Style

The story:

Term Frequency
given a text file,
output a list of the 25
most frequently-occurring
words, ordered by decreasing
frequency

Exercises in Programming Style

The story:

Pride and Prejudice → TF →

mr - 786
elizabeth - 635
very - 488
darcy - 418
such - 395
mrs - 343
much - 329
more - 327
bennet - 323
bingley - 306
jane - 295
miss - 283
one - 275
know - 239
before - 229
herself - 227
though - 226
well - 224
never - 220
…

Term Frequency
given a text file,
output a list of the 25
most frequently-occurring
words, ordered by decreasing
frequency

@cristalopes #style1 *name*

STYLE #1

```
import sys, string
# the global list of [word, frequency] pairs
word_freqs = []
# the list of stop words
with open('../stop_words.txt') as f:
    stop_words = f.read().split(',')
stop_words.extend(list(string.ascii_lowercase))

# iterate through the file one line at a time
for line in open(sys.argv[1]):
    start_char = None
    i = 0
    for c in line:
        if start_char == None:
            if c.isalnum():
                # We found the start of a word
                start_char = i
        else:
            if not c.isalnum():
                # We found the end of a word. Process it
                found = False
                word = line[start_char:i].lower()
                # Ignore stop words
                if word not in stop_words:
                    pair_index = 0
                    # Let's see if it already exists
                    for pair in word_freqs:
                        if word == pair[0]:
                            pair[1] += 1
                            found = True
                            found_at = pair_index
                            break
                        pair_index += 1
                    if not found:
                        word_freqs.append([word, 1])
                    elif len(word_freqs) > 1:
                        # We may need to reorder
                        for n in reversed(range(pair_index)):
                            if word_freqs[pair_index][1] >
                                word_freqs[n][1]:
                                # swap
                                word_freqs[n], word_freqs[
                                    pair_index] = word_freqs[
                                    pair_index], word_freqs[n]
                                pair_index = n
                # Let's reset
                start_char = None
        i += 1

for tf in word_freqs[0:25]:
    print tf[0], ' - ', tf[1]
```

```
# the global list of [word, frequency] pairs
word_freqs = []
# the list of stop words
with open('../stop_words.txt') as f:
    stop_words = f.read().split(',')
stop_words.extend(list(string.ascii_lowercase))
```

```
8
9  # iterate through the file one line at a time
10 for line in open(sys.argv[1]):
11     start_char = None
12     i = 0
13     for c in line:
14         if start_char == None:
15             if c.isalnum():
16                 # We found the start of a word
17                 start_char = i
18         else:
19             if not c.isalnum():
20                 # We found the end of a word. Process it
21                 found = False
22                 word = line[start_char:i].lower()
23                 # Ignore stop words
24                 if word not in stop_words:
25                     pair_index = 0
26                     # Let's see if it already exists
27                     for pair in word_freqs:
28                         if word == pair[0]:
29                             pair[1] += 1
30                             found = True
31                             found_at = pair_index
32                             break
33                         pair_index += 1
34                     if not found:
35                         word_freqs.append([word, 1])
36                     elif len(word_freqs) > 1:
37                         # We may need to reorder
38                         for n in reversed(range(pair_index)):
39                             if word_freqs[pair_index][1] >
                                   word_freqs[n][1]:
40                                 # swap
41                                 word_freqs[n], word_freqs[
                                       pair_index] = word_freqs[
                                       pair_index], word_freqs[n]
42                                 pair_index = n
43                 # Let's reset
```

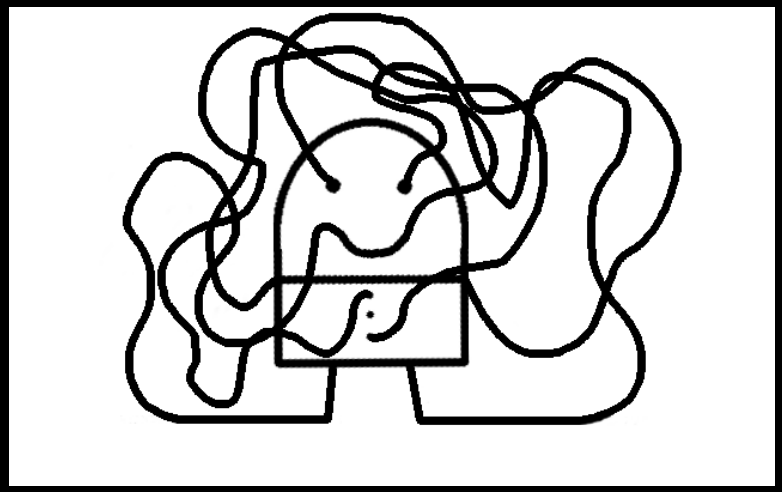
```
import sys, string
# the global list of [word, frequency] pairs
word_freqs = []
# the list of stop words
with open('../stop_words.txt') as f:
    stop_words = f.read().split(',')
stop_words.extend(list(string.ascii_lowercase))

for line in open(sys.argv[1]):
    for c in line:

        if start_char == None:
            if c.isalnum():
                # We found the start of a word
                start_char = i
        else:
            if not c.isalnum():
                # We found the end of a word. Process it
                found = False
                word = line[start_char:i].lower()
                # Ignore stop words
                if word not in stop_words:
                    pair_index = 0
                    # Let's see if it already exists
                    for pair in word_freqs:
                        if word == pair[0]:
                            pair[1] += 1
                            found = True
                            found_at = pair_index
                            break
                        pair_index += 1
                    if not found:
                        word_freqs.append([word, 1])
                    elif len(word_freqs) > 1:
                        # We may need to reorder
                        for n in reversed(range(pair_index)):
                            if word_freqs[pair_index][1] >
                                word_freqs[n][1]:
                                # swap
                                word_freqs[n], word_freqs[
                                    pair_index] = word_freqs[
                                    pair_index], word_freqs[n]
                                pair_index = n
                # Let's reset
                start_char = None
        i += 1

for tf in word_freqs[0:25]:
    print tf[0], ' - ', tf[1]
```

Style #1 constraints

- No abstractions
- No use of library functions

Style #1 constraints

- No abstractions
- No use of library functions



Monolithic Style

@cristalopes #style2 *name*

STYLE #2

```
import re, sys, collections

stopwords = set(open('../stop_words.txt').read().split(','))
words = re.findall('[a-z]{2,}', open(sys.argv[1]).read().lower())
counts = collections.Counter(w for w in words if w not in stopwords)
for (w, c) in counts.most_common(25):
    print w, '-', c
```

Credit: *Laurie Tratt*, Kings College London

```
import re, sys, collections

stopwords=set(open('../stop_words.txt').read().split(','))
words = re.findall('[a-z]{2,}',
                   open(sys.argv[1]).read().lower())
counts = collections.Counter(w for w in words \
                             if w not in stopwords)
for (w, c) in counts.most_common(25):
    print w, '-', c
```

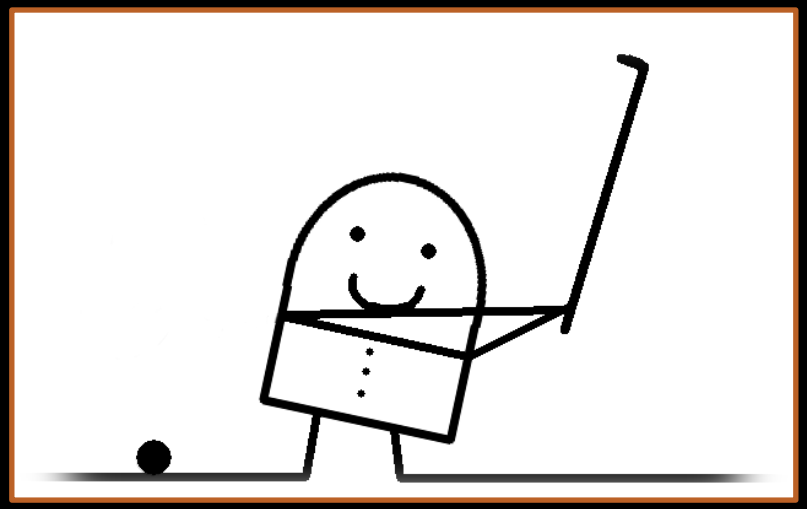
```
import re, string, sys

stops = set(open("../stop_words.txt").read().split(",") +
            list(string.ascii_lowercase))
words = [x.lower() for x in re.split("[^a-zA-Z]+",
                                     open(sys.argv[1]).read())
                  if len(x) > 0 and x.lower() not in stops]
unique_words = list(set(words))
unique_words.sort(lambda x,y:cmp(words.count(y),
                                 words.count(x)))
print "\n".join(["%s - %s" % (x, words.count(x))
                                  for x in unique_words[:25]])
```

Style #2 constraints

- As few lines of code as possible

Style #2 constraints

- As few lines of code as possible

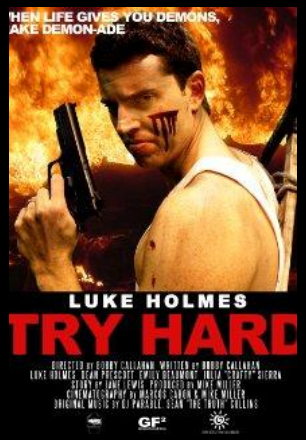


Code Golf Style

@cristalopes #style2 *name*

Style #2 constraints

- As few lines of code as possible



Try Hard Style

@cristalopes #style2 *name*

@cristalopes #style3 *name*

STYLE #3

```
import sys, string

# The shared mutable data
data = []
words = []
word_freqs = []

#
# The functions
#
def read_file(path_to_file):
    """
    Takes a path to a file and assigns the entire
    contents of the file to the global variable data
    """
    global data
    f = open(path_to_file)
    data = data + list(f.read())
    f.close()

def filter_chars_and_normalize():
    """
    Replaces all nonalphanumeric chars in data with white space
    """
    global data
    for i in range(len(data)):
        if not data[i].isalnum():
            data[i] = ' '
        else:
            data[i] = data[i].lower()

def scan():
    """
    Scans data for words, filling the global variable words
    """
    global data
    global words
    data_str = ''.join(data)
    words = words + data_str.split()

def remove_stop_words():
    global words
    f = open('../stop_words.txt')
    stop_words = f.read().split(',')
    f.close()
    # add single-letter words
    stop_words.extend(list(string.ascii_lowercase))
    indeces = []
    for i in range(len(words)):
        if words[i] in stop_words:
            indeces.append(i)
    for i in reversed(indeces):
        words.pop(i)

def frequencies():
    """
    Creates a list of pairs associating
    words with frequencies
    """
    global words
    global word_freqs
    for w in words:
        keys = [wd[0] for wd in word_freqs]
        if w in keys:
            word_freqs[keys.index(w)][1] += 1
        else:
            word_freqs.append([w, 1])

def sort():
    """
    Sorts word_freqs by frequency
    """
    global word_freqs
    word_freqs.sort(lambda x, y: cmp(y[1], x[1]))


#
# The main function
#
read_file(sys.argv[1])
filter_chars_and_normalize()
scan()
remove_stop_words()
frequencies()
sort()

for tf in word_freqs[0:25]:
    print tf[0], ' - ', tf[1]
```

```
data=[]
words=[]
freqs=[]
```

```
def read_file(path):
    """
    Takes a path to a file and assigns the entire
    contents of the file to the global variable data
    """
    global data
    f = open(path_to_file)
    data = data + list(f.read())
```

```
def filter_normalize():
    """
    Replaces all nonalphanumeric chars in data with white space
    """
    global data
    for i in range(len(data)):
        if not data[i].isalnum():
            data[i] = ' '
        else:
            data[i] = data[i].lower()
```

```
def scan():
    """
    Scans data for words, filling the global variable words
    """
    global data
    global words
    data_str = ''.join(data)
    words = words + data_str.split()
```

```
def rem_stop_words():
    f = open('../stop_words.txt')
    stop_words = f.read().split(',')
    f.close()
    # add single-letter words
    stop_words.extend(list(string.ascii_lowercase))
    indeces = []
    for i in range(len(words)):
        if words[i] in stop_words:
            indeces.append(i)
    for i in reversed(indeces):
        words.pop(i)
```

```
def frequencies():
    """
    words with frequencies
    """
    global words
    global word_freqs
    for w in words:
        keys = [wd[0] for wd in word_freqs]
        if w in keys:
            word_freqs[keys.index(w)][1] += 1
        else:
            word_freqs.append([w, 1])
```

```
def sort():
    """
    Sorts word_freqs by frequency
    """
    global word_freqs
    word_freqs.sort(lambda x, y: cmp(y[1], x[1]))
```

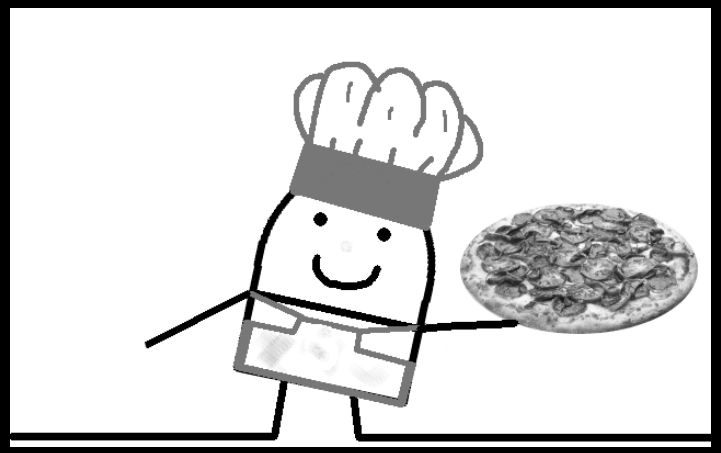
```
#
# Main
#
read_file(sys.argv[1])
filter_normalize()
scan()
rem_stop_words()
frequencies()
sort()

for tf in word_freqs[0:25]:
    print tf[0], ' - ', tf[1]
```

Style #3 constraints

- Procedural abstractions
  - maybe input, no output
- Shared state
- Larger problem solved by applying procedures, one after the other, changing the shared state

Style #3 constraints

- Procedural abstractions
  - maybe input, no output
- Shared state
- Series of commands



Cook Book Style

@cristalopes #style4 *name*

STYLE #4

```
import sys, re, operator, string

#
# The functions
#
def read_file(path_to_file):
    """
    Takes a path to a file and returns the entire
    contents of the file as a string
    """
    f = open(path_to_file)
    data = f.read()
    f.close()
    return data

def filter_chars(str_data):
    """
    Takes a string and returns a copy with all nonalphanumeric
    chars replaced by white space
    """
    pattern = re.compile('[\W_]+')
    return pattern.sub(' ', str_data)

def normalize(str_data):
    """
    Takes a string and returns a copy with all chars in lower case
    """
    return str_data.lower()

def scan(str_data):
    """
    Takes a string and scans for words, returning
    a list of words.
    """
    return str_data.split()

def remove_stop_words(word_list):
    """
    Takes a list of words and returns a copy with all stop
    words removed
    """
    f = open('../stop_words.txt')
    stop_words = f.read().split(',')
    f.close()
    # add single-letter words
    stop_words.extend(list(string.ascii_lowercase))
    return [w for w in word_list if not w in stop_words]

def frequencies(word_list):
    """
    Takes a list of words and returns a dictionary associating
    words with frequencies of occurrence
    """
    word_freqs = {}
    for w in word_list:
        if w in word_freqs:
            word_freqs[w] += 1
        else:
            word_freqs[w] = 1
    return word_freqs

def sort(word_freq):
    """
    Takes a dictionary of words and their frequencies
    and returns a list of pairs where the entries are
    sorted by frequency
    """
    return sorted(word_freq.iteritems(), key=operator.itemgetter
        (1), reverse=True)


#
# The main function
#
word_freqs = sort(frequencies(remove_stop_words(scan(normalize(
    filter_chars(read_file(sys.argv[1])))))))

for tf in word_freqs[0:25]:
    print tf[0], ' - ', tf[1]
```

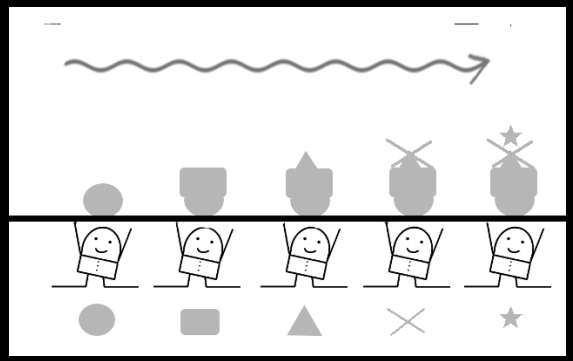
```
import sys, re, operator, string

#
# The functions
def read_file(path):
    """
    Takes a path to a file and returns the entire
    contents of the file as a string
    """
    f = open(path_to_file)

    return ...

def filter(str_data):
    """
    Takes a string and returns a copy with all nonalphanumeric
    chars replaced by white space
    """
    return ...

def normalize(str_data):
    """
    Takes a string and returns a copy with all chars in lower case
    """
    return ...

def scan(str_data):
    """
    Takes a string and scans for words, returning
    a list of the words in the string
    """
    return ...

def rem_stop_words(wordl):
    """
    Takes a list of words and returns a copy with all stop
    words removed
    """
    f = open('../stop_words.txt')
    stop_words = f.read().split(',')
    f.close()
    # add single-letter words
    return ...

def frequencies(wordl):
    """
    Takes a list of words and returns a dictionary associating
    words with frequencies of occurrence
    """
    word_freqs = {}
    for w in word_list:
        if w in word_freqs:
            word_freqs[w] += 1

    return ...

def sort(word_freqs):
    """
    Takes a dictionary of words and their frequencies
    and returns a list of pairs where the entries are
    sorted by frequency
    """
    return ...

#
# Main
#
wfreqs=st(fq(r(sc(n(fc(rf(sys.argv[1])))))))

for tf in wfreqs[0:25]:
    print tf[0], ' - ', tf[1]
```

Style #4 constraints

- Function abstractions
  - *f: Input → Output*
- No shared state
- Function composition $f \circ g$

Style #4 constraints

- Function abstractions
  - *f: Input → Output*
- No shared state
- Function composition *f º g*



Pipeline Style

@cristalopes #style5 *name*

STYLE #5

```
import sys, re, operator, string

#
# The functions
#
def read_file(path_to_file, func):
    """
    Takes a path to a file and returns the entire
    contents of the file as a string
    """
    f = open(path_to_file)
    data = f.read()
    f.close()
    return func(data, normalize)

def filter_chars(str_data, func):
    """
    Takes a string and returns a copy with all nonalphanumeric
        chars
    replaced by white space
    """
    pattern = re.compile('[\W_]+')
    return func(pattern.sub(' ', str_data), scan)

def normalize(str_data, func):
    """
    Takes a string and returns a copy with all characters in lower
        case
    """
    return func(str_data.lower(), remove_stop_words)

def scan(str_data, func):
    """
    Takes a string and scans for words, returning
    a list of words.
    """
    return func(str_data.split(), frequencies)

def remove_stop_words(word_list, func):
    """ Takes a list of words and returns a copy with all stop
        words removed """
    f = open('../stop_words.txt')
    stop_words = f.read().split(',')
    f.close()
    # add single-letter words
    stop_words.extend(list(string.ascii_lowercase))
    return func([w for w in word_list if not w in stop_words],
        sort)

def frequencies(word_list, func):
    """
    Takes a list of words and returns a dictionary associating
    words with frequencies of occurrence
    """
    word_freqs = {}
    for w in word_list:
        if w in word_freqs:
            word_freqs[w] += 1
        else:
            word_freqs[w] = 1
    return func(word_freqs, no_op)

def sort(word_freq, func):
    """
    Takes a dictionary of words and their frequencies
    and returns a list of pairs where the entries are
    sorted by frequency
    """
    return func(sorted(word_freq.iteritems(), key=operator.
        itemgetter(1), reverse=True), None)

def no_op(a, func):
    return a

#
# The main function
#
word_freqs = read_file(sys.argv[1], filter_chars)

for tf in word_freqs[0:25]:
    print tf[0], ' - ', tf[1]
```

```
import sys, re, operator, string

#
# The functions
#

def read_file(path, func):
    ...
    return func(…, normalize)

def filter_chars(data, func):
    ...
    return func(…, scan)

def normalize(data, func):
    ...
    return func(…,remove_stops)

def scan(data, func):
    ...
    return func(…, frequencies)

def remove_stops(data, func):
    ...
    return func(…, sort)

def frequencies(
    """
    Takes a list                    dictionary associating
    words with f                    e
    """
    word_freqs = {}
    for w in word_list:
        if w in word_freqs:
            word_freqs[w] += 1
        else:
            word_freqs[w] = 1
    return func(word_freqs, no_op)

def sort(word_freq, func):
    """
    Takes a dictionary of words and their frequencies
    and returns a list of pairs where the entries are
    sorted by frequency
    """
    return func(sorted(word_freq.iteritems(), key=operator.
        itemgetter(1), reverse=True), None)

def no_op(a, func):
    return a

# Main
w_freqs=read_file(sys.argv[1],
                  filter_chars)

for tf in w_freqs[0:25]:
    print tf[0], ' - ', tf[1]
```

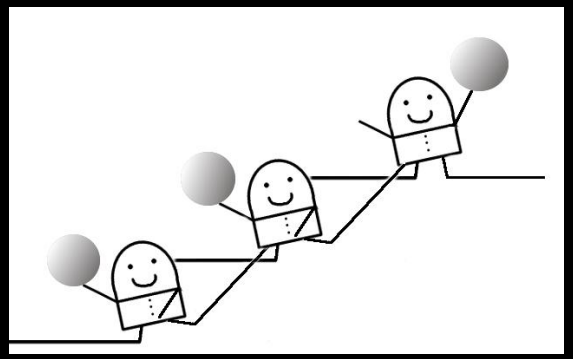
Etc.

Style #5 constraints

- Functions take one additional parameter, f
  - called at the end
  - given what would normally be the return value plus the next function

Style #5 constraints

- Functions take one additional parameter, f
  - called at the end
  - given what would normally be the return value plus the next function



Kick forward Style

@cristalopes #style5 *name*

@cristalopes #style6 *name*

STYLE #6

```
import sys, re, operator, string
from abc import ABCMeta

#
# The classes
#
class TFExercise(object):
    __metaclass__ = ABCMeta

    def info(self):
        return self.__class__.__name__ + ": No major data
            structure"

class DataStorageManager(TFExercise):
    """ Models the contents of the file """
    _data = ''
    def __init__(self, path_to_file):
        f = open(path_to_file)
        self._data = f.read()
        f.close()
        self.__filter_chars()
        self.__normalize()

    def __filter_chars(self):
        """
        Takes a string and returns a copy with all nonalphanumeric
             chars
        replaced by white space
        """
        pattern = re.compile('[\W_]+')
        self._data = pattern.sub(' ', self._data)

    def __normalize(self):
        """
        Takes a string and returns a copy with all characters in
            lower case
        """
        self._data = self._data.lower()

    def words(self):
        """
        Returns the list words in storage
        """
        data_str = ''.join(self._data)
        return data_str.split()

    def info(self):
        return self.__class__.__name__ + ": My major data
            structure is a " + self._data.__class__.__name__

class StopWordManager(TFExercise):
    """ Models the stop word filter """
    _stop_words = []
    def __init__(self):
        f = open('../stop_words.txt')
        self._stop_words = f.read().split(',')
        f.close()
        # add single-letter words
        self._stop_words.extend(list(string.ascii_lowercase))

    def is_stop_word(self, word):
        return word in self._stop_words

    def info(self):
        return self.__class__.__name__ + ": My major data
            structure is a " + self._stop_words.__class__.__name__

class WordFrequencyManager(TFExercise):
    """ Keeps the word frequency data """
    _word_freqs = {}

    def increment_count(self, word):
        if word in self._word_freqs:
            self._word_freqs[word] += 1
        else:
            self._word_freqs[word] = 1

    def sorted(self):
        return sorted(self._word_freqs.iteritems(), key=operator.
            itemgetter(1), reverse=True)

    def info(self):
        return self.__class__.__name__ + ": My major data
            structure is a " + self._word_freqs.__class__.__name__


class WordFrequencyController(TFExercise):
    def __init__(self, path_to_file):
        self._storage_manager = DataStorageManager(path_to_file)
        self._stop_word_manager = StopWordManager()
        self._word_freq_manager = WordFrequencyManager()

    def run(self):
        for w in self._storage_manager.words():
            if not self._stop_word_manager.is_stop_word(w):
                self._word_freq_manager.increment_count(w)

        word_freqs = self._word_freq_manager.sorted()
        for tf in word_freqs[0:25]:
            print tf[0], ' - ', tf[1]

#
# The main function
#
WordFrequencyController(sys.argv[1]).run()
```

```
import sys, re, operator, string
from abc import ABCMeta

#
# The classes
```

class TFExercise():

def info(self):

```
            structure"
```

class DataStorageManager(TFExercise):

```
    _data = ''
    def __init__(self, path_to_file):
        f = open(path_to_file)
        self._data = f.read()
        f.close()
        self.__filter_chars()
        self.__normalize()

    def __filter_chars(self):
        """
        Takes a string and returns a copy with all nonalphanumeric
             chars
        replaced by white space
        """
        pattern = re.compile('[\W_]+')
        self._data = pattern.sub(' ', self._data)

    def __normalize(self):
        """
        Takes a string and returns a copy with all characters in
            lower case
        """
        self._data = self._data.lower()
```

def words(self):

```
        Returns the list words in storage
        """
        data_str = ''.join(self._data)
```

def info(self):

```
        return self.__class__.__name__ + ": My major data
```

class StopWordManager(TFExercise):

```
    """ Models the stop word filter """
    _stop_words = []
    def __init__(self):
        f = open('../stop_words.txt')
        self._stop_words = f.read().split(',')
        f.close()
        # add single-letter words
```

def is_stop_word(self, word):

```
        return word in self._stop_words
```

def info(self):

```
            structure is a " + self._stop_words.__class__.__name__
```

class WordFreqManager(TFExercise):

def inc_count(self, word):

```
            self._word_freqs[word] += 1
        else:
            self._word_freqs[word] = 1
```

def sorted(self):

```
            itemgetter(1), reverse=True)
```

def info(self):

```
            structure is a " + self._word_freqs.__class__.__name__
```

class WordFreqController(TFExercise):

```
    def __init__(self, path_to_file):
        self._storage_manager = DataStorageManager(path_to_file)
        self._stop_word_manager = StopWordManager()
        self._word_freq_manager = WordFrequencyManager()
```

def run(self):

```
        for tf in word_freqs[0:25]:
            print tf[0], ' - ', tf[1]
```

```
# Main
WordFreqController(sys.argv[1]).run()
```

Style #6 constraints

- Things, things and more things!
  - Capsules of data and procedures
- Data is never accessed directly
- Capsules can reappropriate procedures from other capsules

Style #6 constraints

- Things, things and more things!
  - Capsules of data and procedures
- Data is never accessed directly
- Capsules can reappropriate procedures from other capsules

Kingdom of Nouns Style

@cristalopes #style6 *name*

@cristalopes #style7 *name*

STYLE #7

```
import sys, re, operator, string

class DataStorageManager():
    """ Models the contents of the file """
    _data = ''

    def dispatch(self, message):
        if message[0] == 'init':
            return self._init(message[1])
        elif message[0] == 'words':
            return self._words()
        else:
            raise Exception("Message not understood " + message
                [0])

    def _init(self, path_to_file):
        f = open(path_to_file)
        self._data = f.read()
        f.close()
        pattern = re.compile('[\W_]+')
        self._data = pattern.sub(' ', self._data).lower()

    def _words(self):
        """
        Returns the list words in storage
        """
        data_str = ''.join(self._data)
        return data_str.split()


class StopWordManager():
    """ Models the stop word filter """
    _stop_words = []

    def dispatch(self, message):
        if message[0] == 'init':
            return self._init()
        elif message[0] == 'is_stop_word':
            return self._is_stop_word(message[1])
        else:
            raise Exception("Message not understood " + message
                [0])

    def _init(self):
        f = open('../stop_words.txt')
        self._stop_words = f.read().split(',')
        f.close()
        self._stop_words.extend(list(string.ascii_lowercase))

    def _is_stop_word(self, word):
        return word in self._stop_words

class WordFrequencyManager():
    """ Keeps the word frequency data """
    _word_freqs = {}

    def dispatch(self, message):
        if message[0] == 'increment_count':
            return self._increment_count(message[1])
        elif message[0] == 'sorted':
            return self._sorted()
        else:
            raise Exception("Message not understood " + message
                [0])

    def _increment_count(self, word):
        if word in self._word_freqs:
            self._word_freqs[word] += 1
        else:
            self._word_freqs[word] = 1

    def _sorted(self):
        return sorted(self._word_freqs.iteritems(), key=operator.
            itemgetter(1), reverse=True)

class WordFrequencyController():

    def dispatch(self, message):
        if message[0] == 'init':
            return self._init(message[1])
        elif message[0] == 'run':
            return self._run()
        else:
            raise Exception("Message not understood " + message
                [0])

    def _init(self, path_to_file):
        self._storage_manager = DataStorageManager()
        self._stop_word_manager = StopWordManager()
        self._word_freq_manager = WordFrequencyManager()
        self._storage_manager.dispatch(['init', path_to_file])
        self._stop_word_manager.dispatch(['init'])

    def _run(self):
        for w in self._storage_manager.dispatch(['words']):
            if not self._stop_word_manager.dispatch(['is_stop_word
                ', w]):
                self._word_freq_manager.dispatch(['increment_count
                    ', w])

        word_freqs = self._word_freq_manager.dispatch(['sorted'])
        for tf in word_freqs[0:25]:
            print tf[0], ' - ', tf[1]

#
# The main function
#
wfcontroller = WordFrequencyController()
wfcontroller.dispatch(['init', sys.argv[1]])
wfcontroller.dispatch(['run'])
```

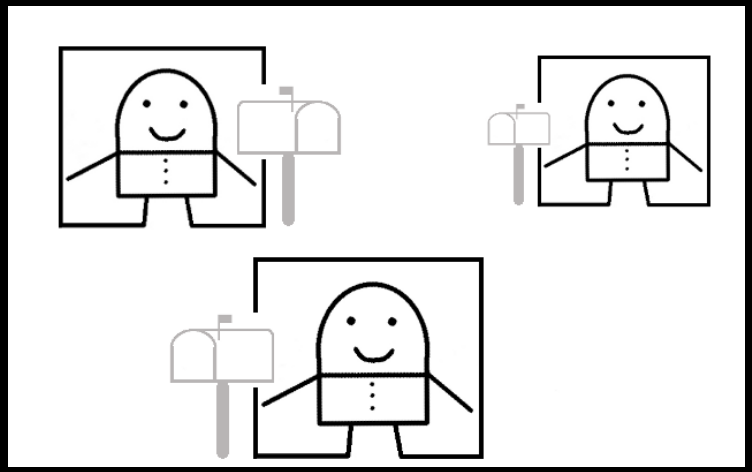
```
import sys, re, operator, string

class DataStorageManager():
    def dispatch(self, message):
        if message[0] == 'init':
            return self._init(message[1])
        elif message[0] == 'words':
            return self._words()
        else:
            raise Exception("Message not understood " + message
                [0])

    def _init(self, path_to_file):
        f = open(path_to_file)
        self._data = f.read()
        f.close()
        pattern = re.compile('[\W_]+')
        self._data = pattern.sub(' ', self._data).lower()

    def _words(self):
        """
        Returns the list words in storage
        """
        data_str = ''.join(self._data)
        return data_str.split()

class StopWordManager():
    def dispatch(self, message):
        if message[0] == 'init':
            return self._init()
        elif message[0] == 'is_stop_word':
            return self._is_stop_word(message[1])
        else:
            raise Exception("Message not understood " + message
                [0])

    def _init(self):
        f = open('../stop_words.txt')
        self._stop_words = f.read().split(',')
        f.close()
        self._stop_words.extend(list(string.ascii_lowercase))

    def _is_stop_word(self, word):
        return word in self._stop_words

class WordFrequencyManager():
    _word_freqs = {}

    def dispatch(self, message):
        if message[0] == 'increment_count':
            return self._increment_count(message[1])
        elif message[0] == 'sorted':
            return self._sorted()
        else:
            raise Exception("Message not understood " + message
                [0])

    def _increment_count(self, word):
        if word in self._word_freqs:
            self._word_freqs[word] += 1
        else:
            self._word_freqs[word] = 1

    def _sorted(self):
        return sorted(self._word_freqs.iteritems(), key=operator.
            itemgetter(1), reverse=True)

class WordFrequencyController():
    def dispatch(self, message):
        if message[0] == 'init':
            return self._init(message[1])
        elif message[0] == 'run':
            return self._run()
        else:
            raise Exception("Message not understood " + message
                [0])

    def _init(self, path_to_file):
        self._storage_manager = DataStorageManager()
        self._stop_word_manager = StopWordManager()
        self._word_freq_manager = WordFrequencyManager()
        self._storage_manager.dispatch(['init', path_to_file])
        self._stop_word_manager.dispatch(['init'])

    def _run(self):
        for w in self._storage_manager.dispatch(['words']):
            if not self._stop_word_manager.dispatch(['is_stop_word
                ', w]):
                self._word_freq_manager.dispatch(['increment_count
                    ', w])

        word_freqs = self._word_freq_manager.dispatch(['sorted'])

# Main
wfcntrl = WordFrequencyController()
wfcntrl.dispatch(['init',sys.argv[1]])
wfcntrl.dispatch(['run'])
```

Style #7 constraints

- (Similar to #6)
- Capsules receive messages via single receiving procedure

Style #7 constraints

- (Similar to #6)
- Capsules receive messages via single receiving procedure



Letterbox Style

@cristalopes #style7 *name*

@cristalopes #style8 *name*

STYLE #8

```python
import sys, re, operator, string

#
# Functions for map reduce
#
def partition(data_str, nlines):
    """
    Generator function that partitions the input data_str (a big string)
    into chunks of nlines.
    """
    lines = data_str.split('\n')
    for i in xrange(0, len(lines), nlines):
        yield '\n'.join(lines[i:i+nlines])

def split_words(data_str):
    """
    Takes a string, filters non alphanumeric characters, normalizes to
    lower case, scans for words, and filters the stop words.
    It returns a list of pairs (word, 1), one for each word in the input, so
    [(w1, 1), (w2, 1), ..., (wn, 1)]
    """
    def _filter_chars(str_data):
        """
        Takes a string and returns a copy with all nonalphanumeric chars
        replaced by white space
        """
        pattern = re.compile('[\W_]+')
        return pattern.sub(' ', str_data)

    def _normalize(str_data):
        """
        Takes a string and returns a copy with all characters in lower case
        """
        return str_data.lower()

    def _scan(str_data):
        """
        Takes a string and scans for words, returning
        a list of words.
        """
        return str_data.split()

    def _remove_stop_words(word_list):
        f = open('../stop_words.txt')
        stop_words = f.read().split(',')
        f.close()
        # add single-letter words
        stop_words.extend(list(string.ascii_lowercase))
        return [w for w in word_list if not w in stop_words]

    # The actual work of splitting the input into words
    result = []
    words = _remove_stop_words(_scan(_normalize(_filter_chars(data_str))))
    for w in words:
        result.append((w, 1))

    return result

def count_words(pairs_list_1, pairs_list_2):
    """
    Takes a two lists of pairs of the form
    [(w1, 1), ...]
    and returns a list of pairs [(w1, frequency), ...],
    where frequency is the sum of all the reported occurrences
    """
    mapping = dict((k, v) for k, v in pairs_list_1)
    for p in pairs_list_2:
        if p[0] in mapping:
            mapping[p[0]] += p[1]
        else:
            mapping[p[0]] = 1

    return mapping.items()

#
# Auxiliary functions
#

def read_file(path_to_file):
    """
    Takes a path to a file and returns the entire
    contents of the file as a string
    """
    f = open(path_to_file)
    data = f.read()
    f.close()
    return data

def sort(word_freq):
    """
    Takes a collection of words and their frequencies
    and returns a collection of pairs where the entries are
    sorted by frequency
    """
    return sorted(word_freq, key=operator.itemgetter(1), reverse=True)


#
# The main function
#
splits = map(split_words, partition(read_file(sys.argv[1]), 200))
splits.insert(0, []) # Normalize input to reduce
word_freqs = sort(reduce(count_words, splits))
```

```
import sys, re, operator, string

#
# Functions for map reduce
#
def partition(data_str, nlines):
    """
    Generator function that partitions the input data_str (a big
        string)
    into chunks of nlines.
    """
    lines = data_str.split('\n')
    for i in xrange(0, len(lines), nlines):
        yield '\n'.join(lines[i:i+nlines])

def split_words(data_str):
    """
    Takes a string, filters non alphanumeric characters,
        normalizes to
    lower case, scans for words, and filters the stop words.
    It returns a list of pairs (word, 1), one for each word in the
         input, so
    [(w1, 1), (w2, 1), ..., (wn, 1)]
    """
    def _filter_chars(str_data):
        """
        Takes a string and returns a copy with all nonalphanumeric
             chars
        replaced by white space
        """
        pattern = re.compile('[\W_]+')
        return pattern.sub(' ', str_data)

    def _normalize(str_data):
        """
        Takes a string and returns a copy with all characters in
            lower case
        """
        return str_data.lower()

    def _scan(str_data):
        """
        Takes a string and scan
        a list of words.
        """
        return str_data.split()

    def _remove_stop_words(word
        f = open('../stop_words
        stop_words = f.read().s
        f.close()
        # add single-letter wo
        stop_words.extend(list(
        return [w for w in word

    # The actual work of splitting the input into words
    result = []
    words = _remove_stop_words(_scan(_normalize(_filter_chars(
        data_str))))
    for w in words:
        result.append((w, 1))

    return result

def count_words(pairs_list_1, pairs_list_2):
    """
    Takes a two lists of pairs of the form
    [(w1, 1), ...]
    and returns a list of pairs [(w1, frequency), ...],
    where frequency is the sum of all the reported occurrences
    """
    mapping = dict((k, v) for k, v in pairs_list_1)
    for p in pairs_list_2:
        if p[0] in mapping:
            mapping[p[0]] += p[1]
        else:
            mapping[p[0]] = 1

    return mapping.items()

#
# Auxiliary functions
#

def read_file(path_to_file):
    """
    Takes a path to a file and returns the entire
    contents of the file as a string
    """
    f = open(path_to_file)
    data = f.read()
    f.close()
```

```
# Main
splits = map(split_words,
             partition(read_file(sys.argv[1]), 200))
splits.insert(0, [])
word_freqs = sort(reduce(count_words, splits))

for tf in word_freqs[0:25]:
    print tf[0], ' - ', tf[1]
```

```
import sys, re, operator, string

#
# Functions for map reduce
#
def partition(data_str, nlines):
    """
    Generator function that partitions the input data_str (a big
        string)
    into chunks of nlines.
    """
    lines = data_str.split('\n')
    for i in xrange(0, len(lines), nlines):
        yield '\n'.join(lines[i:i+nlines])
```

```
    # The actual work of splitting the input into words
    result = []
    words = _remove_stop_words(_scan(_normalize(_filter_chars(
        data_str))))
    for w in words:
        result.append((w, 1))

    return result

def count_words(pairs_list_1, pairs_list_2):
    """
    Takes a two lists of pairs of the form
    [(w1, 1), ...]
```

```
def split_words(data_str)
    """
    Takes a string (many lines), filters, normalizes to
    lower case, scans for words, and filters the stop words.
    Returns a list of pairs (word, 1), so
    [(w1, 1), (w2, 1), ..., (wn, 1)]
    """
    ...
    result = []
    words = _rem_stop_words(_scan(_normalize(_filter(data_str))))
    for w in words:
        result.append((w, 1))
    return result
```

```
        """
        return str_data.split()

    def _remove_stop_words(word_list):
        f = open('../stop_words.txt')
        stop_words = f.read().split(',')
        f.close()
        # add single-letter words
        stop_words.extend(list(string.ascii_lowercase))
        return [w for w in word_list if not w in stop_words]
```

```
    """
    return sorted(word_freq, key=operator.itemgetter(1), reverse=
        True)


#
# The main function
#
splits = map(split_words, partition(read_file(sys.argv[1]), 200))
splits.insert(0, []) # Normalize input to reduce
word_freqs = sort(reduce(count_words, splits))
```

```
import sys, re, operator, string

#
# Functions for map reduce
#
def partition(data_str, nlines):
    """
    Generator function that partitions the input data_str (a big
        string)
    into chunks
    """
    lines = data_
    for i in xran
        yield '\n

def split_words(
    """
    Takes a strin
        normalize
    lower case,
    It returns a
         input,
    [(w1, 1), (w
    """
    def _filter_
        """
        Takes a
             char
        replaced
        """
        pattern =
        return pa

    def _normali
        """
        Takes a
            lowe
        """
        return st

    def _scan(str
        """
        Takes a
        a list of words.
        """
        return str_data.split()

    def _remove_stop_words(word_list):
        f = open('../stop_words.txt')
        stop_words = f.read().split(',')
        f.close()
        # add single-letter words
        stop_words.extend(list(string.ascii_lowercase))
        return [w for w in word_list if not w in stop_words]
```

```
    # The actual work of splitting the input into words
    result = []
    words = _remove_stop_words(_scan(_normalize(_filter_chars(
        data_str))))
    for w in words:
        result.append((w, 1))

    return result
```

```
def count_words(pairs_list_1, pairs_list_2)
    """
    Takes two lists of pairs of the form
    [(w1, 1), ...]
    and returns a list of pairs [(w1, frequency), ...],
    where frequency is the sum of all occurrences
    """
    mapping = dict((k, v) for k, v in pairs_list_1)
    for p in pairs_list_2:
        if p[0] in mapping:
            mapping[p[0]] += p[1]
        else:
            mapping[p[0]] = 1

    return mapping.items()
```

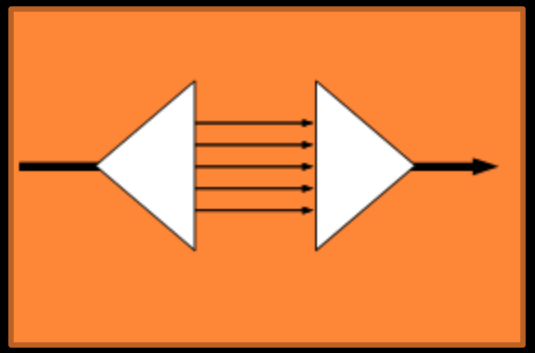
```
    sorted by frequency
    """
    return sorted(word_freq, key=operator.itemgetter(1), reverse=
        True)


#
# The main function
#
splits = map(split_words, partition(read_file(sys.argv[1]), 200))
splits.insert(0, []) # Normalize input to reduce
word_freqs = sort(reduce(count_words, splits))
```

Style #8 constraints

- Two key abstractions: map(f, chunks) and reduce(g, results)

Style #8 constraints

- Two key abstractions: map(f, chunks) and reduce(g, results)



Map-Reduce Style

@cristalopes #style8 *name*

@cristalopes #style9 *name*

STYLE #9

```
import sys, re, string, sqlite3

#
# The relational database of this problem consists of 3 tables:
# documents, words, characters
#
def create_db_schema(connection):
    c = connection.cursor()
    c.execute('''CREATE TABLE documents (id INTEGER PRIMARY KEY
        AUTOINCREMENT, name)''')
    c.execute('''CREATE TABLE words (id, doc_id, value)''')
    c.execute('''CREATE TABLE characters (id, word_id, value)''')
    connection.commit()
    c.close()

def load_file_into_database(path_to_file, connection):
    """ Takes the path to a file and loads the contents into the
        database """
    def _read_file(path_to_file):
        """
        Takes a path to a file and returns the entire contents of
            the
        file as a string
        """
        f = open(path_to_file)
        data = f.read()
        f.close()
        return data

    def _filter_chars_and_normalize(str_data):
        """
        Takes a string and returns a copy with all nonalphanumeric
             chars
        replaced by white space, and all characters lower-cased
        """
        pattern = re.compile('[\W_]+')
        return pattern.sub(' ', str_data).lower()

    def _scan(str_data):
        """ Takes a string and scans for words, returning a list
            of words. """
        return str_data.split()

    def _remove_stop_words(word_list):
        f = open('../stop_words.txt')
        stop_words = f.read().split(',')
        f.close()
        # add single-letter words
        stop_words.extend(list(string.ascii_lowercase))
        return [w for w in word_list if not w in stop_words]

    # The actual work of splitting the input into words
    words = _remove_stop_words(_scan(_filter_chars_and_normalize(
        _read_file(path_to_file))))

    # Now let's add data to the database
    # Add the document itself to the database
    c = connection.cursor()
    c.execute("INSERT INTO documents (name) VALUES (?)", (
        path_to_file,))
    c.execute("SELECT id from documents WHERE name=?", (
        path_to_file,))
    doc_id = c.fetchone()[0]

    # Add the words to the database
    c.execute("SELECT MAX(id) FROM words")
    row = c.fetchone()
    word_id = row[0]
    if word_id == None:
        word_id = 0
    for w in words:
        c.execute("INSERT INTO words VALUES (?, ?, ?)", (word_id,
            doc_id, w))
        # Add the characters to the database
        char_id = 0
        for char in w:
            c.execute("INSERT INTO characters VALUES (?, ?, ?)", (
                char_id, word_id, char))
            char_id += 1
        word_id += 1
    connection.commit()
    c.close()

#
# The main function
#
connection = sqlite3.connect(':memory:')
create_db_schema(connection)
load_file_into_database(sys.argv[1], connection)

# Now, let's query
c = connection.cursor()
c.execute("SELECT value, COUNT(*) as C FROM words GROUP BY value
    ORDER BY C DESC")
for i in range(25):
    row = c.fetchone()
    if row != None:
        print row[0] + ' - '  + str(row[1])

connection.close()
```

```
import sys, re, string, sqlite3

#
# The relational database of this problem consists of 3 tables:
# documents, words, characters
#
def create_db_schema(connection):
    c = connection.cursor()
    c.execute('''CREATE TABLE documents (id INTEGER PRIMARY KEY
        AUTOINCREMENT, name)''')
    c.execute('''CREATE TABLE words (id, doc_id, value)''')
    c.execute('''CREATE TABLE characters (id, word_id, value)''')
    connection.commit()
    c.close()

def load_file_into_database(path_to_file, connection):
    """ Takes the path to a file and loads the contents into the
        database """
    def _read_file(path_to_file):
        """
        Takes a path to a file and returns the entire contents of
            the
        file as a string
```

```
    # Now let's add data to the database
    # Add the document itself to the database
    c = connection.cursor()
    c.execute("INSERT INTO documents (name) VALUES (?)", (
        path_to_file,))
    c.execute("SELECT id from documents WHERE name=?", (
        path_to_file,))
    doc_id = c.fetchone()[0]

    # Add the words to the database
    c.execute("SELECT MAX(id) FROM words")
    row = c.fetchone()
    word_id = row[0]
    if word_id == None:
        word_id = 0
    for w in words:
        c.execute("INSERT INTO words VALUES (?, ?, ?)", (word_id,
            doc_id, w))
        # Add the characters to the database
        char_id = 0
        for char in w:
            c.execute("INSERT INTO characters VALUES (?, ?, ?)", (
                char_id, word_id, char))
```

```
# Main
connection = sqlite3.connect(':memory:')
create_db_schema(connection)
load_file_into_database(sys.argv[1], connection)

# Now, let's query
c = connection.cursor()
c.execute("SELECT value, COUNT(*) as C FROM words GROUP BY value ORDER BY C DESC")
for i in range(25):
    row = c.fetchone()
    if row != None:
        print row[0] + ' - '  + str(row[1])

connection.close()
```

```
def create_db_schema(connection):
    c = connection.cursor()
    c.execute('''CREATE TABLE documents(id PRIMARY KEY AUTOINCREMENT, name)'''
    c.execute('''CREATE TABLE words(id, doc_id, value)''')
    c.execute('''CREATE TABLE characters(id, word_id, value)''')
    connection.commit()
    c.close()
```

```
import sys, re, string, sqlite3

    def _read_file(path_to_file):
        """
        Takes a path to a file and returns the entire contents of
            the
        file as a string
        """
        f = open(path_to_file)
        data = f.read()
        f.close()
        return data

    def _filter_chars_and_normalize(str_data):
        """
        Takes a string and returns a copy with all nonalphanumeric
             chars
        replaced by white space, and all characters lower-cased
        """
        pattern = re.compile('[\W_]+')
        return pattern.sub(' ', str_data).lower()

    def _scan(str_data):
        """ Takes a string and scans for words, returning a list
            of words. """
        return str_data.split()

    def _remove_stop_words(word_list):
        f = open('../stop_words.txt')
        stop_words = f.read().split(',')
        f.close()
        # add single-letter words
        stop_words.extend(list(string.ascii_lowercase))
        return [w for w in word_list if not w in stop_words]

    # The actual work of splitting the input into words
    words = _remove_stop_words(_scan(_filter_chars_and_normalize(
        _read_file(path_to_file))))
```

```
        # Add the characters to the database
        char_id = 0
        for char in w:
            c.execute("INSERT INTO characters VALUES (?, ?, ?)", (
                char_id, word_id, char))
            char_id += 1
        word_id += 1
    connection.commit()
    c.close()

#
# The main function
#
connection = sqlite3.connect(':memory:')
create_db_schema(connection)
load_file_into_database(sys.argv[1], connection)

# Now, let's query
c = connection.cursor()
c.execute("SELECT value, COUNT(*) as C FROM words GROUP BY value
    ORDER BY C DESC")
for i in range(25):
    row = c.fetchone()
    if row != None:
        print row[0] + ' - '  + str(row[1])

connection.close()
```

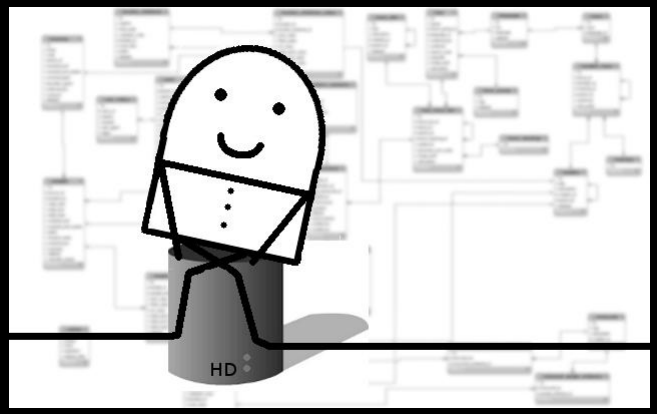
```
# Now let's add data to the database
# Add the document itself to the database
c = connection.cursor()
c.execute("INSERT INTO documents (name) VALUES (?)", (path_to_
c.execute("SELECT id from documents WHERE name=?", (path_to_fi
doc_id = c.fetchone()[0]

# Add the words to the database
c.execute("SELECT MAX(id) FROM words")
row = c.fetchone()
word_id = row[0]
if word_id == None:
    word_id = 0
for w in words:
    c.execute("INSERT INTO words VALUES (?, ?, ?)", (word_id, 
    # Add the characters to the database
    char_id = 0
    for char in w:
        c.execute("INSERT INTO characters VALUES (?, ?, ?)", (
        char_id += 1
    word_id += 1
connection.commit()
c.close()
```

Style #9 constraints

- Entities and relations between them
- Query engine
  - Declarative queries

Style #9 constraints

- Entities and relations between them
- Query engine
  - Declarative queries



Persistent Tables Style

@cristalopes #style9 *name*

Take Home

- Many ways of solving problems
  - Know them, assess them
  - What are you trying to optimize?
- Constraints are important for communication
  - Make them explicit
- Don’t be hostage of one way of doing things

@cristalopes