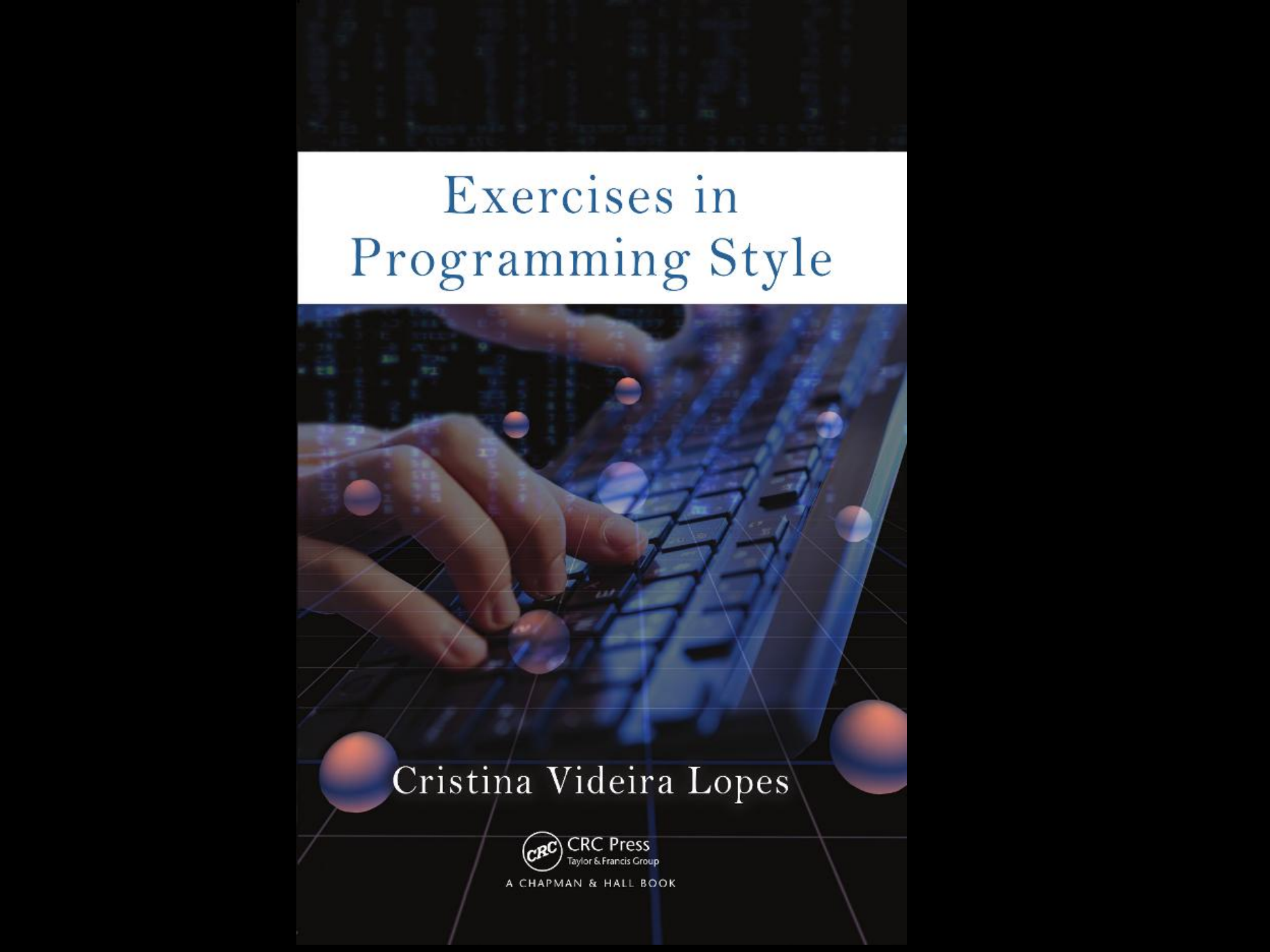


The background of the book cover is a dark, blue-toned image. It features a close-up of a person's hands typing on a computer keyboard. Overlaid on this image are several semi-transparent, glowing spheres in shades of blue and purple. A faint grid pattern is also visible across the lower portion of the cover. At the top, there is a white rectangular area containing the title.

Exercises in Programming Style

Cristina Videira Lopes



CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

Art History, Simplified



DaVinci



El Greco



Rembrandt



Hals



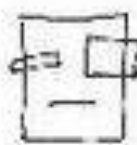
Van Gogh



Seurat



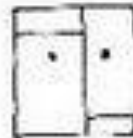
Munch



Braque



Picasso



Mondrian



Malevich



Gericault



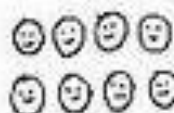
Wood



Miro



Kahlo



Warhol



Rothko



Pollock



Kline



Dali



Johns



Close



Keane



Kinkadee



Mingo

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** →

Term Frequency

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

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

...

@cristalopes #style1 *name*

STYLE #1

```

1 import sys, string
2 # the global list of [word, frequency] pairs
3 word_freqs = []
4 # the list of stop words
5 with open('../stop_words.txt') as f:
6     stop_words = f.read().split(',')
7 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] >
40                             word_freqs[n][1]:
41                             # swap
42                             word_freqs[n], word_freqs[
43                                 pair_index] = word_freqs[
44                                     pair_index], word_freqs[n]
45                             pair_index = n
46                     # Let's reset
47                     start_char = None
48                 i += 1
49
50 for tf in word_freqs[0:25]:
51     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] >
40                             word_freqs[n][1]:
41                             # swap
42                             word_freqs[n], word_freqs[
43                                 pair_index] = word_freqs[
44                                     pair_index], word_freqs[n]
45                     pair_index = n
46                 # Let's reset
```

```

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

```

```

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

```

```

11     for c in line:

```

```

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

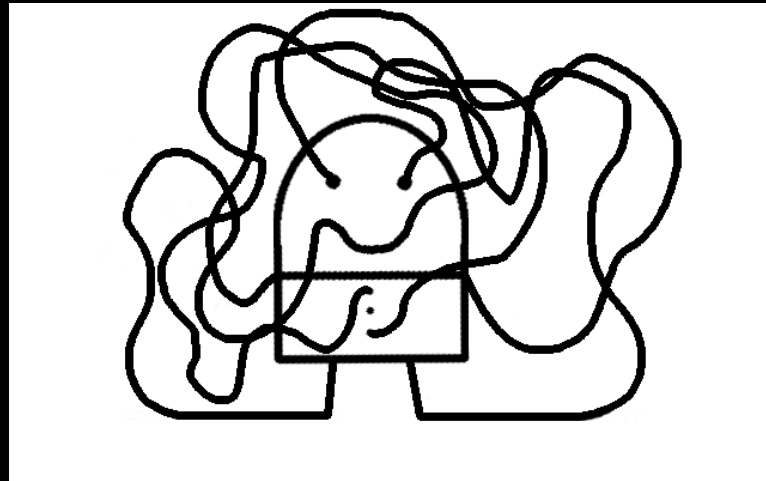
```

Style #1 constraints

- ▷ No abstractions
- ▷ No use of library functions

Style #1 constraints

- ▷ No abstractions
- ▷ No use of library functions



Monolithic Style

@cristalopes #style1 *name*

@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)))

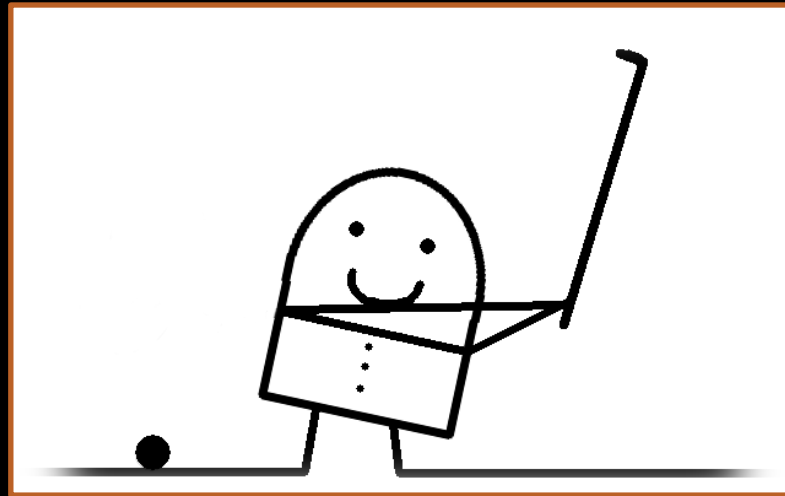
print "\n".join(["%s - %s" % (x, words.count(x))
                for x in unique_words[:25]])
```

Style #2 constraints

- ▶ As few lines of code as possible

Style #2 constraints

- ▶ As few lines of code as possible

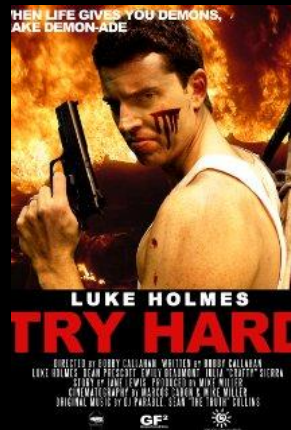


Code Golf Style

@cristalopes #style2 name

Style #2 constraints

▷ As few lines of code as possible



Try Hard Style

@cristalopes #style2 name

@cristalopes #style3 *name*

STYLE #3

```

1 import sys, string
2
3 # The shared mutable data
4 data = []
5 words = []
6 word_freqs = []
7
8 #
9 # The functions
10 #
11 def read_file(path_to_file):
12     """
13     Takes a path to a file and assigns the entire
14     contents of the file to the global variable data
15     """
16     global data
17     f = open(path_to_file)
18     data = data + list(f.read())
19     f.close()
20
21 def filter_chars_and_normalize():
22     """
23     Replaces all nonalphanumeric chars in data with white space
24     """
25     global data
26     for i in range(len(data)):
27         if not data[i].isalnum():
28             data[i] = ' '
29         else:
30             data[i] = data[i].lower()
31
32 def scan():
33     """
34     Scans data for words, filling the global variable words
35     """
36     global data
37     global words
38     data_str = ''.join(data)
39     words = words + data_str.split()
40
41 def remove_stop_words():
42     global words
43     f = open('../stop_words.txt')
44     stop_words = f.read().split(',')
45     f.close()
46     # add single-letter words
47     stop_words.extend(list(string.ascii_lowercase))
48     indices = []
49     for i in range(len(words)):
50         if words[i] in stop_words:
51             indices.append(i)
52     for i in reversed(indices):
53         words.pop(i)
54

```

```

55 def frequencies():
56     """
57     Creates a list of pairs associating
58     words with frequencies
59     """
60     global words
61     global word_freqs
62     for w in words:
63         keys = [wd[0] for wd in word_freqs]
64         if w in keys:
65             word_freqs[keys.index(w)][1] += 1
66         else:
67             word_freqs.append([w, 1])
68
69 def sort():
70     """
71     Sorts word_freqs by frequency
72     """
73     global word_freqs
74     word_freqs.sort(lambda x, y: cmp(y[1], x[1]))
75
76
77 #
78 # The main function
79 #
80 read_file(sys.argv[1])
81 filter_chars_and_normalize()
82 scan()
83 remove_stop_words()
84 frequencies()
85 sort()
86
87 for tf in word_freqs[0:25]:
88     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))
    indices = []
    for i in range(len(words)):
        if words[i] in stop_words:
            indices.append(i)
    for i in reversed(indices):
        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]:
```

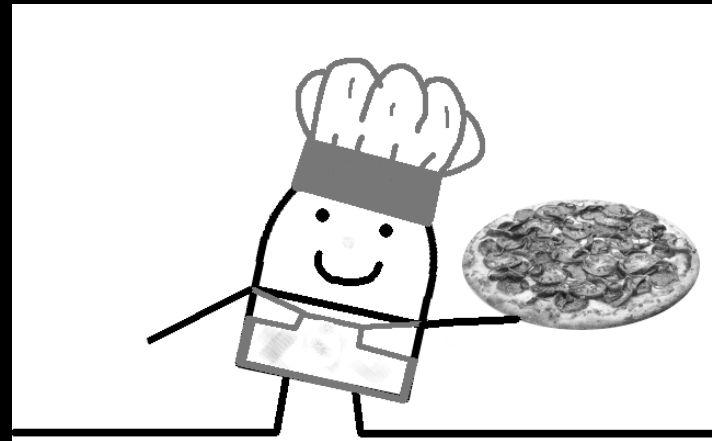
```
    print tf[0], ' - ', tf[1]
```

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

```

1 import sys, re, operator, string
2
3 #
4 # The functions
5 #
6 def read_file(path_to_file):
7     """
8     Takes a path to a file and returns the entire
9     contents of the file as a string
10    """
11    f = open(path_to_file)
12    data = f.read()
13    f.close()
14    return data
15
16 def filter_chars(str_data):
17     """
18     Takes a string and returns a copy with all nonalphanumeric
19     chars replaced by white space
20    """
21    pattern = re.compile('[\W_]+')
22    return pattern.sub(' ', str_data)
23
24 def normalize(str_data):
25     """
26     Takes a string and returns a copy with all chars in lower case
27    """
28    return str_data.lower()
29
30 def scan(str_data):
31     """
32     Takes a string and scans for words, returning
33     a list of words.
34    """
35    return str_data.split()
36
37 def remove_stop_words(word_list):
38     """
39     Takes a list of words and returns a copy with all stop
40     words removed
41    """
42    f = open('../stop_words.txt')
43    stop_words = f.read().split(',')
44    f.close()
45    # add single-letter words
46    stop_words.extend(list(string.ascii_lowercase))
47    return [w for w in word_list if not w in stop_words]
48
49 def frequencies(word_list):
50     """
51     Takes a list of words and returns a dictionary associating
52     words with frequencies of occurrence
53    """
54    word_freqs = {}

```

```

55    for w in word_list:
56        if w in word_freqs:
57            word_freqs[w] += 1
58        else:
59            word_freqs[w] = 1
60    return word_freqs
61
62 def sort(word_freq):
63     """
64     Takes a dictionary of words and their frequencies
65     and returns a list of pairs where the entries are
66     sorted by frequency
67    """
68    return sorted(word_freq.iteritems(), key=operator.itemgetter(
69        1), reverse=True)
69
70
71 #
72 # The main function
73 #
74 word_freqs = sort(frequencies(remove_stop_words(scan(normalize(
75     filter_chars(read_file(sys.argv[1]))))))))
76
77 for tf in word_freqs[0:25]:
78     print tf[0], ' - ', tf[1]

```

```

1 import sys, re, operator, string
2
3 #
4 # The functions
5
6 def read_file(path):
7     """
8     Takes a path to a file and returns the entire
9     contents of the file as a string
10     """
11     f = open(path_to_file)
12
13     return ...
14
15 def filter(str_data):
16     """
17     Takes a string and returns a copy with all nonalphanumeric
18     chars replaced by white space
19     """
20
21     return ... re(['\W_']+')
22     str_data)
23
24 def normalize(str_data):
25     """
26     Returns a copy with all chars in lower case
27     """
28
29     return ... er()
30
31 def scan(str_data):
32     """
33     Takes a string and scans for words, returning
34     a list of words
35     """
36     return str_data.split()
37
38 def rem_stop_words(wordl):
39     """
40     Takes a list of words and returns a copy with all stop
41     words removed
42     """
43     f = open('../stop_words.txt')
44     stop_words = f.read().split(',')
45     f.close()
46
47     return ... words
48     list(string.ascii_lowercase))
49     word_list if not w in stop_words]
50
51 def frequencies(wordl):
52     """
53     Takes a list of words and returns a dictionary associating
54     words with frequencies of occurrence
55     """
56     word_freqs = {}

```

```

55 for w in word_list:
56     if w in word_freqs:
57         word_freqs[w] += 1
58
59 return ... - 1
60
61 def sort(word_freqs):
62     """
63     Returns a list of pairs where the entries are
64     sorted by frequency
65     """
66
67     return ... req.iteritems(), key=operator.itemgetter
68     (1))
69
70

```

```

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

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

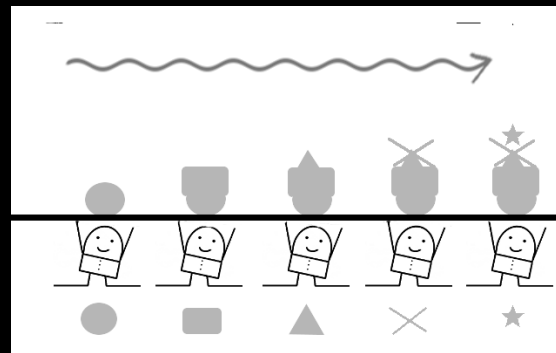
```


Style #4 constraints

- ▷ Function abstractions
 - $f: \text{Input} \rightarrow \text{Output}$
- ▷ No shared state
- ▷ Function composition $f \circ g$

Style #4 constraints

- ▷ Function abstractions
 - $f: \text{Input} \rightarrow \text{Output}$
- ▷ No shared state
- ▷ Function composition $f \circ g$



Pipeline Style

@cristalopes #style4 name

@cristalopes #style5 *name*

STYLE #5

```

1 import sys, re, operator, string
2
3 #
4 # The functions
5 #
6 def read_file(path_to_file, func):
7     """
8     Takes a path to a file and returns the entire
9     contents of the file as a string
10    """
11    f = open(path_to_file)
12    data = f.read()
13    f.close()
14    return func(data, normalize)
15
16 def filter_chars(str_data, func):
17     """
18     Takes a string and returns a copy with all nonalphanumeric
19     chars
20     replaced by white space
21    """
22    pattern = re.compile('[\W_]+')
23    return func(pattern.sub(' ', str_data), scan)
24
25 def normalize(str_data, func):
26     """
27     Takes a string and returns a copy with all characters in lower
28     case
29    """
30    return func(str_data.lower(), remove_stop_words)
31
32 def scan(str_data, func):
33     """
34     Takes a string and scans for words, returning
35     a list of words.
36    """
37    return func(str_data.split(), frequencies)
38
39 def remove_stop_words(word_list, func):
40     """ Takes a list of words and returns a copy with all stop
41     words removed """
42    f = open('../stop_words.txt')
43    stop_words = f.read().split(',')
44    f.close()
45    # add single-letter words
46    stop_words.extend(list(string.ascii_lowercase))
47    return func([w for w in word_list if not w in stop_words],
48                sort)
49
50 def frequencies(word_list, func):
51     """
52     Takes a list of words and returns a dictionary associating
53     words with frequencies of occurrence
54    """

```

```

51 word_freqs = {}
52 for w in word_list:
53     if w in word_freqs:
54         word_freqs[w] += 1
55     else:
56         word_freqs[w] = 1
57 return func(word_freqs, no_op)
58
59 def sort(word_freq, func):
60     """
61     Takes a dictionary of words and their frequencies
62     and returns a list of pairs where the entries are
63     sorted by frequency
64    """
65    return func(sorted(word_freq.iteritems(), key=operator.
66                        itemgetter(1), reverse=True), None)
67
68 def no_op(a, func):
69     return a
70
71 #
72 # The main function
73 #
74 word_freqs = read_file(sys.argv[1], filter_chars)
75
76 for tf in word_freqs[0:25]:
77     print tf[0], ' - ', tf[1]

```

```
1 import sys, re, operator, string
```

```
2 #
```

```
3 # The functions
```

```
4 #
```

```
5
6
7 def read_file(path, func):
8     ...
9
10    return func(..., normalize)
```

```
11
12
13 def filter_chars(data, func):
14     ...
15
16    return func(..., scan)
```

```
17
18
19 def normalize(data, func):
20     ...
21
22    return func(..., remove_stops)
```

```
23
24
25 def scan(data, func):
26     ...
27
28    return func(..., frequencies)
```

```
29
30
31 def remove_stops(data, func):
32     ...
33
34    return func(..., sort)
```

```
35
36
37 def frequencies(
38     """
39     Takes a list
40     words with f
41     """
```

Etc.

dictionary associating
e

```
51 word_freqs = {}
52 for w in word_list:
53     if w in word_freqs:
54         word_freqs[w] += 1
55     else:
56         word_freqs[w] = 1
57 return func(word_freqs, no_op)
58
59 def sort(word_freq, func):
60     """
61     Takes a dictionary of words and their frequencies
62     and returns a list of pairs where the entries are
63     sorted by frequency
64     """
65     return func(sorted(word_freq.iteritems(), key=operator.
66                        itemgetter(1), reverse=True), None)
67
68 def no_op(a, func):
69     return a
```

Main

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

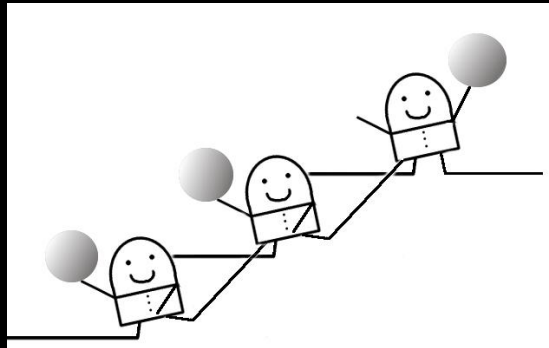
```
for tf in w_freqs[0:25]:
    print tf[0], ' - ', tf[1]
```

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


```

1 import sys, re, operator, string
2 from abc import ABCMeta
3
4 #
5 # The classes
6 #
7 class TFEExercise(object):
8     __metaclass__ = ABCMeta
9
10    def info(self):
11        return self.__class__.__name__ + ": No major data
            structure"
12
13    class DataStorageManager(TFEExercise):
14        """ Models the contents of the file """
15        _data = ''
16        def __init__(self, path_to_file):
17            f = open(path_to_file)
18            self._data = f.read()
19            f.close()
20            self.__filter_chars()
21            self.__normalize()
22
23        def __filter_chars(self):
24            """
25            Takes a string and returns a copy with all nonalphanumeric
                chars
                replaced by white space
            """
26            pattern = re.compile('[\W_]+')
27            self._data = pattern.sub(' ', self._data)
28
29        def __normalize(self):
30            """
31            Takes a string and returns a copy with all characters in
                lower case
            """
32            self._data = self._data.lower()
33
34        def words(self):
35            """
36            Returns the list words in storage
            """
37            data_str = ''.join(self._data)
38            return data_str.split()
39
40        def info(self):
41            return self.__class__.__name__ + ": My major data
                structure is a " + self._data.__class__.__name__
42
43    class StopWordManager(TFEExercise):
44        """ Models the stop word filter """
45        _stop_words = []
46        def __init__(self):

```

```

51        f = open('../stop_words.txt')
52        self._stop_words = f.read().split(',')
53        f.close()
54        # add single-letter words
55        self._stop_words.extend(list(string.ascii_lowercase))
56
57    def is_stop_word(self, word):
58        return word in self._stop_words
59
60    def info(self):
61        return self.__class__.__name__ + ": My major data
            structure is a " + self._stop_words.__class__.__name__
62
63    class WordFrequencyManager(TFEExercise):
64        """ Keeps the word frequency data """
65        _word_freqs = {}
66
67        def increment_count(self, word):
68            if word in self._word_freqs:
69                self._word_freqs[word] += 1
70            else:
71                self._word_freqs[word] = 1
72
73        def sorted(self):
74            return sorted(self._word_freqs.iteritems(), key=operator.
                itemgetter(1), reverse=True)
75
76        def info(self):
77            return self.__class__.__name__ + ": My major data
                structure is a " + self._word_freqs.__class__.__name__
78
79    class WordFrequencyController(TFEExercise):
80        def __init__(self, path_to_file):
81            self._storage_manager = DataStorageManager(path_to_file)
82            self._stop_word_manager = StopWordManager()
83            self._word_freq_manager = WordFrequencyManager()
84
85        def run(self):
86            for w in self._storage_manager.words():
87                if not self._stop_word_manager.is_stop_word(w):
88                    self._word_freq_manager.increment_count(w)
89
90            word_freqs = self._word_freq_manager.sorted()
91            for tf in word_freqs[0:25]:
92                print tf[0], ' - ', tf[1]
93
94    #
95    # The main function
96    #
97    WordFrequencyController(sys.argv[1]).run()

```

```

1 import sys, re, operator, string
2 from abc import ABCMeta
3
4 #
5 # The classes
6
7
8 class TFEercise():
9
10     def info(self):
11         """
12         structure is a " + self.__class__.__name__ + ": No major data
13         """
14
15 class DataStorageManager(TFEercise):
16
17     _data = ''
18     def __init__(self, path_to_file):
19         f = open(path_to_file)
20         self._data = f.read()
21         f.close()
22         self.__filter_chars()
23         self.__normalize()
24
25     def __filter_chars(self):
26         """
27         Takes a string and returns a copy with all nonalphanumeric
28         chars
29         replaced by white space
30         """
31         pattern = re.compile('[\W_]+')
32         self._data = pattern.sub(' ', self._data)
33
34     def __normalize(self):
35         """
36         Takes a string and returns a copy with all characters in
37         lower case
38         """
39         self._data = self._data.lower()
40
41     def words(self):
42         """
43         Returns the list words in storage
44         """
45         data_str = ''.join(self._data)
46
47     def info(self):
48         """
49         structure is a " + self.__class__.__name__ + ": My major data
50         """
51
52 class StopWordManager(TFEercise):
53
54     """ Models the stop word filter """
55     _stop_words = []
56     def __init__(self):

```

```

57         f = open('../stop_words.txt')
58         self._stop_words = f.read().split(',')
59         f.close()
60         # add single-letter words
61
62     def is_stop_word(self, word):
63         return word in self._stop_words
64
65     def info(self):
66         """
67         structure is a " + self._stop_words.__class__.__name__ + ": My major data
68         """
69
70 class WordFreqManager(TFEercise):
71
72     word_freqs = {}
73
74     def inc_count(self, word):
75         if word in self.word_freqs:
76             self.word_freqs[word] += 1
77         else:
78             self.word_freqs[word] = 1
79
80     def sorted(self):
81         return sorted(self.word_freqs.items(), key=operator.
82             itemgetter(1), reverse=True)
83
84     def info(self):
85         """
86         structure is a " + self.word_freqs.__class__.__name__ + ": My major data
87         """
88
89 class WordFreqController(TFEercise):
90
91     def __init__(self, path_to_file):
92         self._storage_manager = DataStorageManager(path_to_file)
93         self._stop_word_manager = StopWordManager()
94         self._word_freq_manager = WordFrequencyManager()
95
96     def run(self):
97         words = self._storage_manager.words():
98         for w in words:
99             if not self._stop_word_manager.is_stop_word(w):
100                 self._word_freq_manager.increment_count(w)
101
102         word_freqs = self._word_freq_manager.sorted()
103         for tf in word_freqs[0:25]:
104             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

```

1 import sys, re, operator, string
2
3 class DataStorageManager():
4     """ Models the contents of the file """
5     _data = ''
6
7     def dispatch(self, message):
8         if message[0] == 'init':
9             return self._init(message[1])
10        elif message[0] == 'words':
11            return self._words()
12        else:
13            raise Exception("Message not understood " + message
14                             [0])
15
16    def _init(self, path_to_file):
17        f = open(path_to_file)
18        self._data = f.read()
19        f.close()
20        pattern = re.compile('[\W_]+')
21        self._data = pattern.sub(' ', self._data).lower()
22
23    def _words(self):
24        """
25        Returns the list words in storage
26        """
27        data_str = ''.join(self._data)
28        return data_str.split()
29
30 class StopWordManager():
31     """ Models the stop word filter """
32     _stop_words = []
33
34     def dispatch(self, message):
35         if message[0] == 'init':
36             return self._init()
37         elif message[0] == 'is_stop_word':
38             return self._is_stop_word(message[1])
39         else:
40             raise Exception("Message not understood " + message
41                              [0])
42
43    def _init(self):
44        f = open('../stop_words.txt')
45        self._stop_words = f.read().split(',')
46        f.close()
47        self._stop_words.extend(list(string.ascii_lowercase))
48
49    def _is_stop_word(self, word):
50        return word in self._stop_words
51
52 class WordFrequencyManager():
53     """ Keeps the word frequency data """

```

```

54     _word_freqs = {}
55
56    def dispatch(self, message):
57        if message[0] == 'increment_count':
58            return self._increment_count(message[1])
59        elif message[0] == 'sorted':
60            return self._sorted()
61        else:
62            raise Exception("Message not understood " + message
63                             [0])
64
65    def _increment_count(self, word):
66        if word in self._word_freqs:
67            self._word_freqs[word] += 1
68        else:
69            self._word_freqs[word] = 1
70
71    def _sorted(self):
72        return sorted(self._word_freqs.iteritems(), key=operator.
73                       itemgetter(1), reverse=True)
74
75 class WordFrequencyController():
76
77    def dispatch(self, message):
78        if message[0] == 'init':
79            return self._init(message[1])
80        elif message[0] == 'run':
81            return self._run()
82        else:
83            raise Exception("Message not understood " + message
84                             [0])
85
86    def _init(self, path_to_file):
87        self._storage_manager = DataStorageManager()
88        self._stop_word_manager = StopWordManager()
89        self._word_freq_manager = WordFrequencyManager()
90        self._storage_manager.dispatch(['init', path_to_file])
91        self._stop_word_manager.dispatch(['init'])
92
93    def _run(self):
94        for w in self._storage_manager.dispatch(['words']):
95            if not self._stop_word_manager.dispatch(['is_stop_word', w]):
96                self._word_freq_manager.dispatch(['increment_count', w])
97
98        word_freqs = self._word_freq_manager.dispatch(['sorted'])
99        for tf in word_freqs[0:25]:
100            print tf[0], ' - ', tf[1]
101
102 #
103 # The main function
104 #
105 wfcontroller = WordFrequencyController()
106 wfcontroller.dispatch(['init', sys.argv[1]])
107 wfcontroller.dispatch(['run'])

```

```

1 import sys, re, operator, string
2
3 class DataStorageManager():
4
5     def dispatch(self, message):
6
7         if message[0] == 'init':
8             return self._init(message[1])
9         elif message[0] == 'words':
10             return self._words()
11         else:
12             raise Exception("Message not understood " + message
13                               [0])
14
15     def _init(self, path_to_file):
16         f = open(path_to_file)
17         self._data = f.read()
18         f.close()
19         pattern = re.compile('[\W_]+')
20         self._data = pattern.sub(' ', self._data).lower()
21
22     def _words(self):
23         """
24         Returns the list words in storage
25         """
26         data_str = ''.join(self._data)
27         return data_str.split()
28

```

```

29 class StopWordManager():
30
31     def dispatch(self, message):
32
33         if message[0] == 'init':
34             return self._init()
35         elif message[0] == 'is_stop_word':
36             return self._is_stop_word(message[1])
37         else:
38             raise Exception("Message not understood " + message
39                               [0])
40
41     def _init(self):
42         f = open('../stop_words.txt')
43         self._stop_words = f.read().split(',')
44         f.close()
45         self._stop_words.extend(list(string.ascii_lowercase))
46
47     def _is_stop_word(self, word):
48         return word in self._stop_words
49

```

```

50 class WordFrequencyManager():
51

```

```

52     word_freqs = {}
53
54     def dispatch(self, message):
55
56         return self._increment_count(message[1])
57         elif message[0] == 'sorted':
58             return self._sorted()
59         else:
60             raise Exception("Message not understood " + message
61                               [0])
62
63     def _increment_count(self, word):
64         if word in self._word_freqs:
65             self._word_freqs[word] += 1
66         else:
67             self._word_freqs[word] = 1
68
69     def _sorted(self):
70         return sorted(self._word_freqs.iteritems(), key=operator.
71                       itemgetter(1), reverse=True)
72

```

```

73 class WordFrequencyController():
74
75     def dispatch(self, message):
76
77         return self._init(message[1])
78         elif message[0] == 'run':
79             return self._run()
80         else:
81             raise Exception("Message not understood " + message
82                               [0])
83
84     def _init(self, path_to_file):
85         self._storage_manager = DataStorageManager()
86         self._stop_word_manager = StopWordManager()
87         self._word_freq_manager = WordFrequencyManager()
88         self._storage_manager.dispatch(['init', path_to_file])
89         self._stop_word_manager.dispatch(['init'])
90
91     def _run(self):
92         for w in self._storage_manager.dispatch(['words']):
93             if not self._stop_word_manager.dispatch(['is_stop_word', w]):
94                 self._word_freq_manager.dispatch(['increment_count', w])
95
96         word_freqs = self._word_freq_manager.dispatch(['sorted'])
97

```

```

98 # Main
99

```

```

100 wfcntnl = WordFrequencyController()
101 wfcntnl.dispatch(['init', sys.argv[1]])
102 wfcntnl.dispatch(['run'])

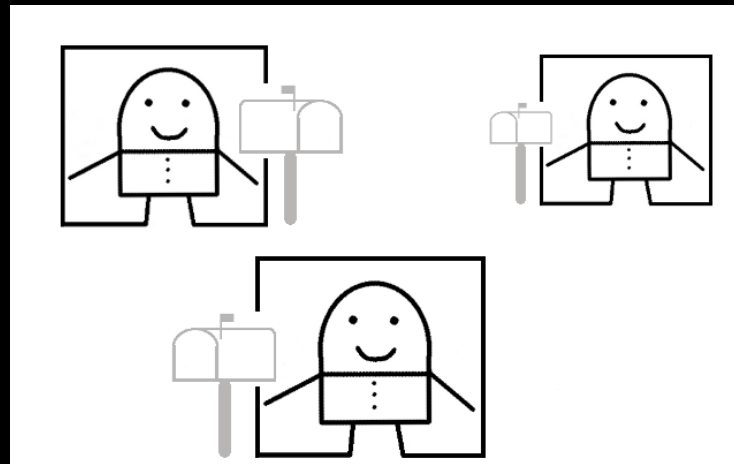
```

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

```

1 import sys, re, operator, string
2
3 #
4 # Functions for map reduce
5 #
6 def partition(data_str, nlines):
7     """
8     Generator function that partitions the input data_str (a big
9     string)
10    into chunks of nlines.
11    """
12    lines = data_str.split('\n')
13    for i in xrange(0, len(lines), nlines):
14        yield '\n'.join(lines[i:i+nlines])
15
16 def split_words(data_str):
17     """
18     Takes a string, filters non alphanumeric characters,
19     normalizes to
20     lower case, scans for words, and filters the stop words.
21     It returns a list of pairs (word, 1), one for each word in the
22     input, so
23     [(w1, 1), (w2, 1), ..., (wn, 1)]
24     """
25
26 def _filter_chars(str_data):
27     """
28     Takes a string and returns a copy with all nonalphanumeric
29     chars
30     replaced by white space
31     """
32     pattern = re.compile('[\W_]+')
33     return pattern.sub(' ', str_data)
34
35 def _normalize(str_data):
36     """
37     Takes a string and returns a copy with all characters in
38     lower case
39     """
40     return str_data.lower()
41
42 def _scan(str_data):
43     """
44     Takes a string and scans for words, returning
45     a list of words.
46     """
47     return str_data.split()
48
49 def _remove_stop_words(word_list):
50     f = open('../stop_words.txt')
51     stop_words = f.read().split(',')
52     f.close()
53     # add single-letter words
54     stop_words.extend(list(string.ascii_lowercase))
55     return [w for w in word_list if not w in stop_words]

```

```

56 # The actual work of splitting the input into words
57 result = []
58 words = _remove_stop_words(_scan(_normalize(_filter_chars(
59     data_str))))
60 for w in words:
61     result.append((w, 1))
62
63 return result
64
65 def count_words(pairs_list_1, pairs_list_2):
66     """
67     Takes a two lists of pairs of the form
68     [(w1, 1), ...]
69     and returns a list of pairs [(w1, frequency), ...],
70     where frequency is the sum of all the reported occurrences
71     """
72     mapping = dict((k, v) for k, v in pairs_list_1)
73     for p in pairs_list_2:
74         if p[0] in mapping:
75             mapping[p[0]] += p[1]
76         else:
77             mapping[p[0]] = 1
78
79 return mapping.items()
80
81 #
82 # Auxiliary functions
83 #
84 def read_file(path_to_file):
85     """
86     Takes a path to a file and returns the entire
87     contents of the file as a string
88     """
89     f = open(path_to_file)
90     data = f.read()
91     f.close()
92     return data
93
94 def sort(word_freq):
95     """
96     Takes a collection of words and their frequencies
97     and returns a collection of pairs where the entries are
98     sorted by frequency
99     """
100    return sorted(word_freq, key=operator.itemgetter(1), reverse=
101        True)
102
103 #
104 # The main function
105 #
106 splits = map(split_words, partition(read_file(sys.argv[1]), 200))
107 splits.insert(0, []) # Normalize input to reduce
108 word_freqs = sort(reduce(count_words, splits))

```

```

1 import sys, re, operator, string
2
3 #
4 # Functions for map reduce
5 #
6 def partition(data_str, nlines):
7     """
8     Generator function that partitions the input data_str (a big
9     string)
10    into chunks of nlines.
11    """
12    lines = data_str.split('\n')
13    for i in xrange(0, len(lines), nlines):
14        yield '\n'.join(lines[i:i+nlines])
15
16 def split_words(data_str):
17     """
18     Takes a string, filters non alphanumeric characters,
19     normalizes to
20     lower case, scans for words, and filters the stop words.
21     It returns a list of pairs (word, 1), one for each word in the
22     input, so
23     [(w1, 1), (w2, 1), ..., (wn, 1)]
24     """
25
26 def _filter_chars(str_data):
27     """
28     Takes a string and returns a copy with all nonalphanumeric
29     chars
30     replaced by white space
31     """
32     pattern = re.compile('[\W_]+')
33     return pattern.sub(' ', str_data)
34
35 def _normalize(str_data):
36     """
37     Takes a string and returns a copy with all characters in
38     lower case
39     """
40     return str_data.lower()
41
42 def _scan(str_data):
43     """
44     Takes a string and scans
45     a list of words.
46     """
47     return str_data.split()
48
49 def _remove_stop_words(words):
50     f = open('../stop_words.txt')
51     stop_words = f.read().split()
52     f.close()
53     # add single-letter words
54     stop_words.extend(list('a-z'))
55     return [w for w in words if w not in stop_words]

```

```

50
51 # The actual work of splitting the input into words
52 result = []
53 words = _remove_stop_words(_scan(_normalize(_filter_chars(
54     data_str))))
55 for w in words:
56     result.append((w, 1))
57
58 return result
59
60 def count_words(pairs_list_1, pairs_list_2):
61     """
62     Takes a two lists of pairs of the form
63     [(w1, 1), ...]
64     and returns a list of pairs [(w1, frequency), ...],
65     where frequency is the sum of all the reported occurrences
66     """
67     mapping = dict((k, v) for k, v in pairs_list_1)
68     for p in pairs_list_2:
69         if p[0] in mapping:
70             mapping[p[0]] += p[1]
71         else:
72             mapping[p[0]] = 1
73
74 return mapping.items()
75
76 #
77 # Auxiliary functions
78 #
79 def read_file(path_to_file):
80     """
81     Takes a path to a file and returns the entire
82     contents of the file as a string
83     """
84     f = open(path_to_file)
85     data = f.read()
86     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]

```

```

1 import sys, re, operator, string
2
3 #
4 # Functions for map reduce
5 #
6 def partition(data_str, nlines):
7     """
8     Generator function that partitions the input data_str (a big
9     string)
10    into chunks of nlines.
11    """
12    lines = data_str.split('\n')
13    for i in xrange(0, len(lines), nlines):
14        yield '\n'.join(lines[i:i+nlines])

```

```

50 # The actual work of splitting the input into words
51 result = []
52 words = _remove_stop_words(_scan(_normalize(_filter_chars(
53     data_str))))
54 for w in words:
55     result.append((w, 1))
56
57 return result
58
59 def count_words(pairs_list_1, pairs_list_2):
60     """
61     Takes a two lists of pairs of the form
62     [(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

```

```

39 # list of words.
40 """
41 return str_data.split()
42
43 def _remove_stop_words(word_list):
44     f = open('../stop_words.txt')
45     stop_words = f.read().split(',')
46     f.close()
47     # add single-letter words
48     stop_words.extend(list(string.ascii_lowercase))
49     return [w for w in word_list if not w in stop_words]

```

```

93 sorted by frequency
94 """
95 return sorted(word_freq, key=operator.itemgetter(1), reverse=
96     True)
97
98 #
99 # The main function
100 #
101 splits = map(split_words, partition(read_file(sys.argv[1]), 200))
102 splits.insert(0, []) # Normalize input to reduce
103 word_freqs = sort(reduce(count_words, splits))

```

```

1 import sys, re, operator, string
2
3 #
4 # Functions for map reduce
5 #
6 def partition(data_str, nlines):
7     """
8     Generator function that partitions the input data_str (a big
9     string)
10    into chunks of size nlines
11    """
12    lines = data_str
13    for i in xrange(0, len(lines), nlines):
14        yield lines[i:i+nlines]
15
16 def split_words(data_str):
17     """
18     Takes a string and returns a list of words.
19     It normalizes the input, i.e. it converts
20     everything to lower case, and it removes
21     punctuation.
22     """
23     # The actual work of splitting the input into words
24     result = []
25     words = _remove_stop_words(_scan(_normalize(_filter_chars(
26         data_str))))
27     for w in words:
28         result.append((w, 1))
29     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()
```

```

30 def _normalize(data_str):
31     """
32     Takes a string and returns a list of words.
33     It normalizes the input, i.e. it converts
34     everything to lower case, and it removes
35     punctuation.
36     """
37     # The actual work of splitting the input into words
38     result = []
39     words = _remove_stop_words(_scan(_normalize(_filter_chars(
40         data_str))))
41     for w in words:
42         result.append((w, 1))
43     return result
44
45 def _remove_stop_words(word_list):
46     f = open('../stop_words.txt')
47     stop_words = f.read().split(',')
48     f.close()
49     # add single-letter words
50     stop_words.extend(list(string.ascii_lowercase))
51     return [w for w in word_list if not w in stop_words]

```

```

52 # The actual work of splitting the input into words
53 result = []
54 words = _remove_stop_words(_scan(_normalize(_filter_chars(
55     data_str))))
56 for w in words:
57     result.append((w, 1))
58 return result
59
60 # The main function
61 #
62 splits = map(split_words, partition(read_file(sys.argv[1]), 200))
63 word_freqs = sort(reduce(count_words, splits))
64
65 sorted by frequency
66 """
67 return sorted(word_freq, key=operator.itemgetter(1), reverse=
68     True)
69
70 #
71 # The main function
72 #
73 splits = map(split_words, partition(read_file(sys.argv[1]), 200))
74 splits.insert(0, []) # Normalize input to reduce
75 word_freqs = sort(reduce(count_words, splits))

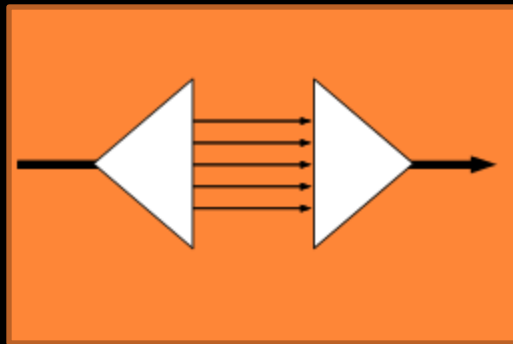
```

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

```

1 import sys, re, string, sqlite3
2
3 #
4 # The relational database of this problem consists of 3 tables:
5 # documents, words, characters
6 #
7 def create_db_schema(connection):
8     c = connection.cursor()
9     c.execute(''''CREATE TABLE documents (id INTEGER PRIMARY KEY
10         AUTOINCREMENT, name)''')
11     c.execute(''''CREATE TABLE words (id, doc_id, value)''')
12     c.execute(''''CREATE TABLE characters (id, word_id, value)''')
13     connection.commit()
14     c.close()
15
16 def load_file_into_database(path_to_file, connection):
17     """ Takes the path to a file and loads the contents into the
18         database """
19
20     def _read_file(path_to_file):
21         """
22         Takes a path to a file and returns the entire contents of
23         the
24         file as a string
25         """
26         f = open(path_to_file)
27         data = f.read()
28         f.close()
29         return data
30
31     def _filter_chars_and_normalize(str_data):
32         """
33         Takes a string and returns a copy with all nonalphanumeric
34         chars
35         replaced by white space, and all characters lower-cased
36         """
37         pattern = re.compile('[\W_]+')
38         return pattern.sub(' ', str_data).lower()
39
40     def _scan(str_data):
41         """ Takes a string and scans for words, returning a list
42             of words. """
43         return str_data.split()
44
45     def _remove_stop_words(word_list):
46         f = open('../stop_words.txt')
47         stop_words = f.read().split(',')
48         f.close()
49         # add single-letter words
50         stop_words.extend(list(string.ascii_lowercase))
51         return [w for w in word_list if not w in stop_words]
52
53 # The actual work of splitting the input into words
54 words = _remove_stop_words(_scan(_filter_chars_and_normalize(
55     _read_file(path_to_file))))

```

```

49
50 # Now let's add data to the database
51 # Add the document itself to the database
52 c = connection.cursor()
53 c.execute("INSERT INTO documents (name) VALUES (?)", (
54     path_to_file,))
55 c.execute("SELECT id from documents WHERE name=?", (
56     path_to_file,))
57 doc_id = c.fetchone()[0]
58
59 # Add the words to the database
60 c.execute("SELECT MAX(id) FROM words")
61 row = c.fetchone()
62 word_id = row[0]
63 if word_id == None:
64     word_id = 0
65
66 for w in words:
67     c.execute("INSERT INTO words VALUES (?, ?, ?)", (word_id,
68         doc_id, w))
69
70 # Add the characters to the database
71 char_id = 0
72 for char in w:
73     c.execute("INSERT INTO characters VALUES (?, ?, ?)", (
74         char_id, word_id, char))
75     char_id += 1
76 word_id += 1
77 connection.commit()
78 c.close()
79
80 #
81 # The main function
82 #
83 connection = sqlite3.connect(':memory:')
84 create_db_schema(connection)
85 load_file_into_database(sys.argv[1], connection)
86
87 # Now, let's query
88 c = connection.cursor()
89 c.execute("SELECT value, COUNT(*) as C FROM words GROUP BY value
90     ORDER BY C DESC")
91
92 for i in range(25):
93     row = c.fetchone()
94     if row != None:
95         print row[0] + ' - ' + str(row[1])
96
97 connection.close()

```

```

1 import sys, re, string, sqlite3
2
3 #
4 # The relational database of this problem consists of 3 tables:
5 # documents, words, characters
6 #
7 def create_db_schema(connection):
8     c = connection.cursor()
9     c.execute(''''CREATE TABLE documents (id INTEGER PRIMARY KEY
10                AUTOINCREMENT, name)''')
11     c.execute(''''CREATE TABLE words (id, doc_id, value)''')
12     c.execute(''''CREATE TABLE characters (id, word_id, value)''')
13     connection.commit()
14     c.close()
15
16 def load_file_into_database(path_to_file, connection):
17     """ Takes the path to a file and loads the contents into the
18         database """
19     def _read_file(path_to_file):
20         """
21         Takes a path to a file and returns the entire contents of
22         the
23         file as a string

```

```

49
50 # Now let's add data to the database
51 # Add the document itself to the database
52 c = connection.cursor()
53 c.execute("INSERT INTO documents (name) VALUES (?)", (
54     path_to_file,))
55 c.execute("SELECT id from documents WHERE name=?", (
56     path_to_file,))
57 doc_id = c.fetchone()[0]
58
59 # Add the words to the database
60 c.execute("SELECT MAX(id) FROM words")
61 row = c.fetchone()
62 word_id = row[0]
63 if word_id == None:
64     word_id = 0
65
66 for w in words:
67     c.execute("INSERT INTO words VALUES (?, ?, ?)", (word_id,
68                                                         doc_id, w))
69
70 # Add the characters to the database
71 char_id = 0
72
73 for char in w:
74     c.execute("INSERT INTO characters VALUES (?, ?, ?)", (
75         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()

```

```
1 import sys, re, string, sqlite3
```

```
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()
```

```
17 def _read_file(path_to_file):
```

```
18     """
```

```
19     Takes a path to a file and returns the entire contents of  
    the
```

```
20     file as a string
```

```
21     """
```

```
22     f = open(path_to_file)
```

```
23     data = f.read()
```

```
24     f.close()
```

```
25     return data
```

```
26
```

```
27 def _filter_chars_and_normalize(str_data):
```

```
28     """
```

```
29     Takes a string and returns a copy with all nonalphanumeric  
    chars
```

```
30     replaced by white space, and all characters lower-cased
```

```
31     """
```

```
32     pattern = re.compile('[\W_]+')
```

```
33     return pattern.sub(' ', str_data).lower()
```

```
34
```

```
35 def _scan(str_data):
```

```
36     """ Takes a string and scans for words, returning a list  
    of words. """
```

```
37     return str_data.split()
```

```
38
```

```
39 def _remove_stop_words(word_list):
```

```
40     f = open('../stop_words.txt')
```

```
41     stop_words = f.read().split(',')
```

```
42     f.close()
```

```
43     # add single-letter words
```

```
44     stop_words.extend(list(string.ascii_lowercase))
```

```
45     return [w for w in word_list if not w in stop_words]
```

```
46
```

```
47     # The actual work of splitting the input into words
```

```
48     words = _remove_stop_words(_scan(_filter_chars_and_normalize(  
        _read_file(path_to_file))))
```

```
49
```

```
50     # Now let's add data to the database
```

```
51     # Add the document itself to the database
```

```
65     # Add the characters to the database
```

```
66     char_id = 0
```

```
67     for char in w:
```

```
68         c.execute("INSERT INTO characters VALUES (?, ?, ?)", (  
            char_id, word_id, char))
```

```
69         char_id += 1
```

```
70         word_id += 1
```

```
71     connection.commit()
```

```
72     c.close()
```

```
73
```

```
74     #
```

```
75     # The main function
```

```
76     #
```

```
77     connection = sqlite3.connect(':memory:')
```

```
78     create_db_schema(connection)
```

```
79     load_file_into_database(sys.argv[1], connection)
```

```
80
```

```
81     # Now, let's query
```

```
82     c = connection.cursor()
```

```
83     c.execute("SELECT value, COUNT(*) as C FROM words GROUP BY value  
    ORDER BY C DESC")
```

```
84     for i in range(25):
```

```
85         row = c.fetchone()
```

```
86         if row != None:
```

```
87             print row[0] + ' - ' + str(row[1])
```

```
88
```

```
89     connection.close()
```

```
1 import sys, re, string, sqlite3
```

```
49
```

```
# Now let's add data to the database
```

```
50
```

```
# Add the document itself to the database
```

```
51
```

```
# 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, c,
```

```
    # Add the characters to the database
```

```
    char_id = 0
```

```
    for char in w:
```

```
        c.execute("INSERT INTO characters VALUES (?, ?, ?)", (c,
```

```
        char_id += 1
```

```
    word_id += 1
```

```
connection.commit()
```

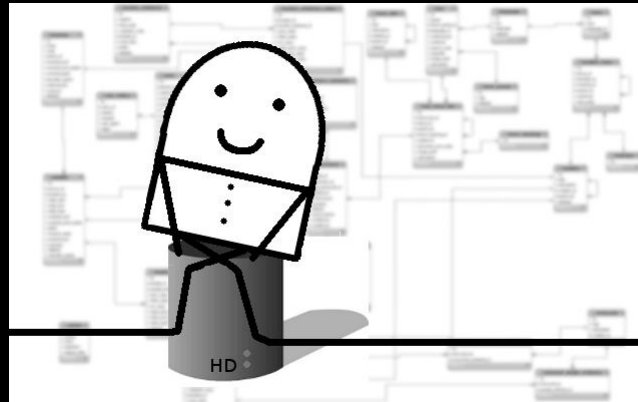
```
c.close()
```

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