Art History, Simplified

DaVinci  El Greco  Rembrandt  Hals  Van Gogh
Seurat  Munch  Braque  Picasso  Mondrian
Matevich  Gericault  Wood  Miro  Kahlo
Warhol  Rothko  Pollock  Kline  Dali
Johns  Close  Keane  Kinkade  Mingo
Rules and constraints in software construction

PROGRAMMING STYLES
Programming Styles

- Ways of expressing tasks
- Exist at all scales
- Recur in multiple scales
- Codified in PLs
Why Are Styles Important?

- Many
- Common vocabularies
- Basic frames of reference
- Some better than others
  - Depending on many things!
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)
The story:

Term Frequency
given a text file,
output a list of the 25
most frequently-occurring
non stop, words, ordered by
decreasing frequency
Exercises in Programming Style

The story: Pride and Prejudice

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
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
...

TF
http://github.com/crista/
exercises-in-programming-style
@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])
                            if 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))

# 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+1)):
                                if word_freqs[n][1] > word_freqs[pair_index][1]:
                                    # swap
                                    word_freqs[n], word_freqs[  
                                        pair_index] = word_freqs[  
                                        pair_index], word_freqs[n]
                        # Let's reset
                        pair_index = n
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]:
                                word_freqs[n], word_freqs[
                                    pair_index] = word_freqs[
                                    pair_index], word_freqs[n]
                                # 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 Main Characteristics

- No abstractions
- No use of libraries
Style #1 Main Characteristics

▷ No abstractions
▷ No use of libraries

Monolith

@cristalopes #style1 name
Style #1 Main Characteristics

- No abstractions
- No use of libraries

Brain-dump Style

@cristalopes #style1 name
@cristalopes #style2 name

STYLE #2
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]])

Credit: Laurie Tratt, Kings College London
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]]])
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 "%s - %s" % (x, words.count(x))
for x in unique_words[:25])
Style #2 Main Characteristics

- No [named] abstractions
- Very few [long] lines of code
- Advanced libraries / constructs
Style #2 Main Characteristics

- No [named] abstractions
- Very few [long] lines of code
- Advanced libraries / constructs

Code Golf Style

@cristalopes #style2 name
Style #2 Main Characteristics

- No [named] abstractions
- Very few [long] lines of code
- Advanced libraries / constructs

Try Hard Style

@cristalopes #style2 name
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():
    """
    # 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]
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():
    """Counts the number of times each word appears in the file and returns a list of tuples containing the word
    and its frequency. The list is sorted by frequency in descending order.
    ""
    global word_freqs
    word_freqs = {}
    for w in words:
        if w in word_freqs:
            word_freqs[w] += 1
        else:
            word_freqs[w] = 1
    global word_freqs
    word_freqs = [(w, freq) for w, freq in sorted(word_freqs.items(), key=lambda x: x[1], reverse=True)]

def sort():
    """Sorts word_freqs by frequency"
    ""
    global word_freqs
    word_freqs.sort(key=lambda x: x[1], reverse=True)

def 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 Main Characteristics

- Procedural abstractions
  - maybe input, no output
- Shared state
- Commands
Style #3 Main Characteristics

➢ Procedural abstractions
  • maybe input, no output
➢ Shared state
➢ Commands

Cook Book Style

@cristalopes #style3 name
@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(r'[^\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.items(), key=operator.itemgetter(1), reverse=True)

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 alphanumeric 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 words split()"
    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()
    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_freqs:
        if w in word_freqs[w] += 1
    return ...

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

for tf in wfreqs[0:25]:
    print tf[0], ' - ', tf[1]
Style #4 Main Characteristics

- Function abstractions
  - \( f: \text{Input} \rightarrow \text{Output} \)
- No shared state
- Function composition \( f \circ g \)
Style #4 Main Characteristics

▲ Function abstractions
  • $f$: Input $\rightarrow$ Output

▲ No shared state

▲ Function composition $f \circ g$

@cristalopes #style5 name

STYLE #5
import sys, re, operator, string

# The functions

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

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

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

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

# Takes a list of words and returns a copy with all stop
# words removed
def remove_stop_words(word_list, func):
    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],
            #
# The main function

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

def no_op(a, func):
    return a

# The main function

word_freqs = {}
for w in word_list:
    if w in word_freqs:
        word_freqs[w] += 1
    else:
        word_freqs[w] = 1

return func(normalize(sys.argv[1], filter_chars),
            #
            #
            word_freqs = read_file(sys.argv[1], filter_chars),
            #
            for tf in word_freqs[0:25]:
                print tf[0], '-', tf[1]
```python
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 of words with frequencies and returns a dictionary
    associating each word with its frequency.
    """
    # Etc.

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.items(), 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]
```

Functions take one additional parameter, `f`

- called at the end
- given what would normally be the return value plus the next function
Style #5 Main Characteristics

- Functions take one additional parameter, f
  - called at the end
  - given what would normally be the return value plus the next function
Style #5 Main Characteristics

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

Crochet Style
@cristalopes #style5 name
@cristalopes #style6 name

STYLE #6
import sys, re, operator, string
from abc import ABCMeta

class TFXExercise(object):
    __metaclass__ = ABCMeta

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

class DataStorageManager(TFXExercise):
    """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(TFXExercise):
    """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(TFXExercise):
    """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(TFXExercise):
    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()
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 structure is a "' + self.__stop_words.__class__.__name__ + '

class StopWordManager(TFExercise):
    """ Models the stop word filter """
    __stop_words = []
    def __init__(self):
        pass
    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 WordFreqController(TFExercise):
    def run(self):
        if not self.__stop_word_manager.is_stop_word(w):
            self.__word_freq_manager.increment_count(w)
    def info(self):
        return self.__class__.__name__ + ': My major data structure is a "' + self.__word_freq_manager.__class__.__name__ + '

class WordFreqManager(TFExercise):
    def sorted(self):
        return sorted(self.__word_freqs.items(), key=operator.itemgetter(1), reverse=True)
    def inc_count(self, word):
        if word in self.__word_freqs:
            self.__word_freqs[word] += 1
        else:
            self.__word_freqs[word] = 1
    def info(self):
        return self.__class__.__name__ + ': My major data structure is a "' + self.__word_freqs.__class__.__name__ + '

# Main
WordFreqController(sys.argv[1]).run()
Style #6 Main Characteristics

- Things, things and more things!
- Capsules of data and procedures
- Data is never accessed directly
- Capsules say “I do the same things as that one, and more!”

@cristalopes #style6 name
Style #6 Main Characteristics

- Things, things and more things!
- Capsules of data and procedures
- Data is never accessed directly
- Capsules say “I do the same things as that one, and more!”
@cristalopes #style7 name

STYLE #7
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 range(0, len(lines), nlines):
        yield '\n'.join(lines[i: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 [(w, 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

# 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, []))
def partition(data_str, nlines):
    
    Generator function that partitions the input data_str (a big string) into chunks of nlines.
    
    lines = data_str.split('
')
    for i in xrange(0, len(lines), nlines):
        yield ''.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
    [(wl, 1), (w2, 1), ..., (wn, 1)]

def _filter_chars(str_data):
    
    Takes a string and returns a copy with all nonalphanumeric chars replaced by whitespace
    
    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 a list of words.
    
    return str_data.split()
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_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()
Style #7 Main Characteristics

- Two key abstractions: `map(f, chunks)` and `reduce(g, results)`
Style #7 Main Characteristics

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

iMux Style
@cristalopes #style7 name
@cristalopes #style8 name

STYLE #8
import sys, re, string, sqlite3

# The relational database of this problem consists of 3 tables:
# documents, words, characters

# 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()

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()

# The relational database of this problem consists of 3 tables:
# documents, words, characters
# 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

# 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()

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','r')
    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, char, word_id))
    char_id += 1
    word_id += 1
connection.commit()
c.close()
Style #8 Main Characteristics

- Entities and relations between them
- Query engine
- Declarative queries
Style #8 Main Characteristics

▷ Entities and relations between them
▷ Query engine
  • Declarative queries

<table>
<thead>
<tr>
<th>Z</th>
<th>Model</th>
<th>1 Gyr</th>
<th>4 Gyr</th>
<th>8 Gyr</th>
<th>12 Gyr</th>
<th>17 Gyr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.008</td>
<td>V96</td>
<td>6.24</td>
<td>6.63</td>
<td>6.79</td>
<td>6.88</td>
<td>6.97</td>
</tr>
<tr>
<td>0.008</td>
<td>grid II</td>
<td>5.78</td>
<td>7.21</td>
<td>7.31</td>
<td>7.43</td>
<td>7.48</td>
</tr>
<tr>
<td>0.02</td>
<td>V96</td>
<td>8.32</td>
<td>8.44</td>
<td>8.25</td>
<td>8.22</td>
<td>8.09</td>
</tr>
<tr>
<td>0.02</td>
<td>grid II</td>
<td>6.84</td>
<td>8.57</td>
<td>8.57</td>
<td>8.63</td>
<td>8.57</td>
</tr>
<tr>
<td>0.05</td>
<td>V96</td>
<td>8.50</td>
<td>8.90</td>
<td>8.34</td>
<td>8.08</td>
<td>7.92</td>
</tr>
<tr>
<td>0.05</td>
<td>grid II</td>
<td>7.22</td>
<td>9.92</td>
<td>9.62</td>
<td>9.65</td>
<td>9.63</td>
</tr>
</tbody>
</table>

Tabular Style
@cristalopes #style8 name