karld Documentation

Release 0.2.10

John W Lockwood IV

October 07, 2015

Contents

Contents:

CHAPTER 1

Getting Started with karld

You have some things you want to do with some data you have. Maybe it's in a couple of files, or one big file and you need to clean it up and extract just part of it or maybe you also need to merge multiple kinds of files based on a common part such as an email address. It's not already indexed in a database either so you can't just do a SQL statement to get the results. You could manipulate it with python, but putting all the data in big dictionaries with email as the key and then iterating over one doing lookups on the other proves to be slow and can only be done with limited size data.

karld is here to help. First off, the name karld was chosen because it sounds like knarled, but it's knot.

CHAPTER 2

Examples

2.1 Split data

From the example directory available by cloning the repository at https://github.com/johnwlockwood/karl_data.

Once cloned to your local system, cd into the karld project directory and run

    python setup.py install

This will install karld. Then cd into the example directory and run:

    python split_multiline.py

This will read multiline/data.csv and produce split_data_ml and split_data_ml_pipe. Run it and compare the input and output. Checkout the source.

2.1.1 Split csv files

Use split_file to split up your data files or use split_csv_file to split up csv files which may have multi-line fields to ensure they are not broken up.:

```
import os

import karld

big_file_names = [
    "bigfile1.csv",
    "bigfile2.csv",
    "bigfile3.csv"
]

data_path = os.path.join('path','to','data', 'root')


def main():
    for filename in big_file_names:
        # Name the directory to write the split files into based
        # on the name of the file.
        out_dir = os.path.join(data_path, 'split_data', filename.replace('.csv', ''))

        # Split the file, with a default max_lines=200000 per shard of the file.
        karld.io.split_csv_file(os.path.join(data_path, filename), out_dir)
```

```
if __name__ == "__main__":
    main()
```

When you're generating data and want to shard it out to files based on quantity, use one of the split output functions such as `split_file_output_csv`, `split_file_output` or `split_file_output_json`:

```
import os
import pathlib

import karld


def main():
    """
    Python 2 version
    """

    items = (str(x) + os.linesep for x in range(2000))

    out_dir = pathlib.Path('shgen')
    karld.io.ensure_dir(str(out_dir))

    karld.io.split_file_output('big_data', items, str(out_dir))


if __name__ == "__main__":
    main()
```

2.1.2 CSV serializable data

```
import pathlib

import karld


def main():
    """
    From a source of data, shard it to csv files.
    """
    if karld.is_py3():
        third = chr
    else:
        third = unichr

    # Your data source
    items = ((x, x + 1, third(x + 10)) for x in range(2000))

    out_dir = pathlib.Path('shard_out_csv')

    karld.io.ensure_dir(str(out_dir))

    karld.io.split_file_output_csv('big_data.csv', items, str(out_dir))


if __name__ == "__main__":
    main()
```

2.1.3 Rows of json serializable data

```
import pathlib

import karld


def main():
    """
    From a source of data, shard it to csv files.
    """
    if karld.is_py3():
        third = chr
    else:
        third = unichr

    # Your data source
    items = ((x, x + 1, third(x + 10)) for x in range(2000))

    out_dir = pathlib.Path('shard_out_json')

    karld.io.ensure_dir(str(out_dir))

    karld.io.split_file_output_json('big_data.json', items, str(out_dir))


if __name__ == "__main__":
    main()
```

2.2 Consume data

2.2.1 Consume the contents of a csv file iteratively.

```
from __future__ import print_function
from operator import itemgetter

import pathlib

import karld


def main():
    """
    Iterate over a the row of a csv file, extracting the data
    you desire.
    """
    data_file_path = pathlib.Path('test_data/things_kinds/data_0.csv')

    rows = karld.io.i_get_csv_data(str(data_file_path))

    kinds = set(map(itemgetter(1), rows))

    for kind in kinds:
        print(kind)
```

```
if __name__ == "__main__":
    main()
```

2.2.2 Consume many csv files iteratively as one stream.

```
from __future__ import print_function
from itertools import chain

try:
    from itertools import imap
except ImportError:
    # if python 3
    imap = map

import karld

from karld.path import i_walk_csv_paths


def main():
    """
    Consume many csv files as if one.
    """
    import pathlib

    input_dir = pathlib.Path('test_data/things_kinds')

    # # Use a generator expression
    # iterables = (karld.io.i_get_csv_data(data_path)
    #              for data_path in i_walk_csv_paths(str(input_dir)))

    # # or a generator map.
    iterables = imap(karld.io.i_get_csv_data,
                     i_walk_csv_paths(str(input_dir)))

    items = chain.from_iterable(iterables)

    for item in items:
        print(item[0], item[1])


if __name__ == "__main__":
    main()
```

The clean.py example shows processing multiple csv files in parallel.

2.3 Tap Data

2.3.1 Use Simple functions to get info from a stream of data.

```
from functools import partial
import os
```

```
from karld.iter_utils import i_batch

from karld.loadump import is_file_csv
from karld.run_together import csv_file_consumer
from karld.run_together import pool_run_files_to_files
from karld.tap import Bucket
from karld.tap import stream_tap


def get_fruit(item):
    """Get things that are fruit.

    :returns: thing of item if it's a fruit"""
    if len(item) == 2 and item[1] == u"fruit":
        return item[0]


def get_metal(item):
    """Get things that are metal.

    :returns: thing of item if it's metal"""
    if len(item) == 2 and item[1] == u"metal":
        return item[0]


def certain_kind_tap(data_items):
    """
    :param data_items: A sequence of unicode strings
    """
    fruit_spigot = Bucket(get_fruit)
    metal_spigot = Bucket(get_metal)

    items = stream_tap((fruit_spigot, metal_spigot), data_items)

    for batch in i_batch(100, items):
        tuple(batch)

    return fruit_spigot.contents(), metal_spigot.contents()


def run(in_dir):
    """
    Run the composition of csv_file_consumer and information tap
    with the csv files in the input directory, and collect
    the results from each file and merge them together,
    printing both kinds of results.

    :param in_dir: directory of input csv files.
    """
    files_to_files_runner = pool_run_files_to_files

    results = files_to_files_runner(
        partial(csv_file_consumer, certain_kind_tap),
        in_dir, filter_func=is_file_csv)

    fruit_results = []
    metal_results = []
```

```
    for fruits, metals in results:
        for fruit in fruits:
            fruit_results.append(fruit)

        for metal in metals:
            metal_results.append(metal)

    print("=== fruits ===")
    for fruit in fruit_results:
        print(fruit)

    print("=== metals ===")
    for metal in metal_results:
        print(metal)

if __name__ == "__main__":
    run(os.path.join("test_data", "things_kinds"))
```

CHAPTER 3

karld Package

3.1 karld Package

karld.__init__.**is_py3**()

3.2 _meta Module

3.3 conversion_operators Module

karld.conversion_operators.**apply_conversion_map**(*conversion_map*, *entity*)
returns tuple of conversions

karld.conversion_operators.**apply_conversion_map_map**(*conversion_map*, *entity*)
returns ordered dict of keys and converted values

karld.conversion_operators.**get_number_as_int**(*number*)
Returns the first number from a string.

karld.conversion_operators.**join_stripped_gotten_value**(*sep*, *getters*, *data*)
Join the values, coerced to str and stripped of whitespace padding, from entity, gotten with collection of getters, with the separator.

Parameters

- **sep** (*str*) – Separator of values.
- **getters** – collection of callables takes that data and returns value.
- **data** – argument for the getters

karld.conversion_operators.**join_stripped_values**(*sep*, *collection_getter*, *data*)
Join the values, coerced to str and stripped of whitespace padding, from entity, gotten with collection_getter, with the separator.

Parameters

- **sep** (*str*) – Separator of values.
- **collection_getter** – callable takes that data and returns collection.
- **data** – argument for the collection_getter

3.4 iter_utils Module

karld.iter_utils.**i_batch**(*max_size*, *iterable*)
Generator that iteratively batches items to a max size and consumes the items iterable as each batch is yielded.

Parameters

- **max_size** (*int*) – Max size of each batch.
- **iterable** (*iter*) – An iterable

karld.iter_utils.**yield_getter_of**(*getter_maker*, *iterator*)
Iteratively map iterator over the result of getter_maker.

Parameters

- **getter_maker** – function that returns a getter function.
- **iterator** – An iterator.

karld.iter_utils.**yield_nth_of**(*nth*, *iterator*)
For an iterator that returns sequences, yield the nth value of each.

Parameters

- **nth** (*int*) – Index desired column of each sequence.
- **iterator** – iterator of sequences.

3.5 loadump Module

karld.loadump.**dump_dicts_to_json_file**(*file_name*, *dicts*, *buffering=10485760*)
writes each dictionary in the dicts iterable to a line of the file as json.

NOTE: Deprecated. replaced by write_as_json, to match the signature of write_to_csv.

Parameters **buffering** (*int*) – number of bytes to buffer files

karld.loadump.**ensure_dir**(*directory*)
If directory doesn't exist, make it.

Parameters **directory** (*str*) – path to directory

karld.loadump.**ensure_file_path_dir**(*file_path*)
Ensure the parent directory of the file path.

Parameters **file_path** (*str*) – Path to file.

karld.loadump.**file_path_and_name**(*path*, *base_name*)
Join the path and base_name and yield it and the base_name.

Parameters

- **path** (*str*) – Directory path
- **base_name** (*str*) – File name

Returns *tuple* of file path and file name.

karld.loadump.**i_get_csv_data**(*file_name*, **args*, ***kwargs*)
A generator for reading a csv file.

karld.loadump.**i_get_json_data**(*file_name*, **args*, ***kwargs*)
A generator for reading file with json documents delimited by newlines.

karld.loadump.**i_read_buffered_file**(*file_name*, *buffering=10485760*, *binary=True*, *py3_csv_read=False*, *encoding='utf-8'*)
Generator of lines of a file name, with buffering for speed.

karld.loadump.**i_walk_dir_for_filepaths_names**(*root_dir*)
Walks a directory yielding the paths and names of files.

Parameters **root_dir** (*str*) – path to a directory.

karld.loadump.**i_walk_dir_for_paths_names**(*root_dir*)
Walks a directory yielding the directory of files and names of files.

Parameters **root_dir** (*str*) – path to a directory.

karld.loadump.**identity**(**args*)

karld.loadump.**is_file_csv**(*file_path_name*)
Is the file a csv file? Identify by extension.

Parameters **file_path_name** (*str*) –

karld.loadump.**is_file_json**(*file_path_name*)
Is the file a json file? Identify by extension.

Parameters **file_path_name** (*str*) –

karld.loadump.**raw_line_reader**(*file_object*)

karld.loadump.**split_file**(*file_path*, *out_dir=None*, *max_lines=200000*, *buffering=10485760*, *line_reader=<function raw_line_reader>*, *split_file_writer=<function split_file_output>*, *read_binary=True*)
Opens then shards the file.

Parameters

- **file_path** (*str*) – Path to the large input file.
- **max_lines** (*int*) – Max number of lines in each shard.
- **out_dir** (*str*) – Path of directory to put the shards.
- **buffering** (*int*) – number of bytes to buffer files

karld.loadump.**split_file_output**(*name*, *data*, *out_dir*, *max_lines=1100*, *buffering=10485760*)
Split an iterable lines into groups and write each to a shard.

Parameters

- **name** (*str*) – Each shard will use this in it's name.
- **data** (*iter*) – Iterable of data to write.
- **out_dir** (*str*) – Path to directory to write the shards.
- **max_lines** (*int*) – Max number of lines per shard.
- **buffering** (*int*) – number of bytes to buffer files

karld.loadump.**split_file_output_csv**(*filename*, *data*, *out_dir=None*, *max_lines=1100*, *buffering=10485760*, *write_as_csv=<function write_as_csv>*)

Split an iterable of csv serializable rows of data into groups and write each to a csv shard.

Parameters **buffering** (*int*) – number of bytes to buffer files

karld.loadump.**split_file_output_json**(*filename*, *dict_list*, *out_dir=None*, *max_lines=1100*, *buffering=10485760*)

Split an iterable of JSON serializable rows of data into groups and write each to a shard.

Parameters **buffering** (*int*) – number of bytes to buffer files

karld.loadump.**write_as_csv**(*items*, *file_name*, *append=False*, *line_buffer_size=None*, *buffering=10485760*, *get_csv_row_writer=<function get_csv_row_writer>*)

Writes out items to a csv file in groups.

Parameters

- **items** – An iterable collection of collections.
- **file_name** – path to the output file.
- **append** – whether to append or overwrite the file.
- **line_buffer_size** – number of lines to write at a time.
- **buffering** (*int*) – number of bytes to buffer files
- **get_csv_row_writer** – callable that returns a csv row writer function, customize this for non-default options: *custom_writer = partial(get_csv_row_writer, delimiter="|"); write_as_csv(items, 'my_out_file', get_csv_row_writer=custom_writer)*

karld.loadump.**write_as_json**(*items*, *file_name*, *buffering=10485760*)

writes each dictionary in the dicts iterable to a line of the file as json.

Parameters

- **items** – A sequence of json dumpable objects.
- **file_name** – the path of the output file.
- **buffering** (*int*) – number of bytes to buffer files

3.6 merger Module

karld.merger.**get_first_if_any**(*values*)

karld.merger.**get_first_type_instance_of_group**(*instance_type*, *group*)

karld.merger.**i_get_multi_groups**(*iterables*, *key=None*)

karld.merger.**i_merge_group_sorted**(*iterables*, *key=None*)

karld.merger.**merge**(**iterables*, ***kwargs*)

Merge multiple sorted inputs into a single sorted output.

Similar to sorted(itertools.chain(*iterables)) but returns a generator, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

```
>>> list(merge([[2,1],[2,3],[2,5],[2,7]],
[[2,0],[2,2],[2,4],[2,8]],
[[2,5],[2,10],[2,15],[2,20]],
[], [[2,25]]), key=itemgetter(-1))
[0, 1, 2, 3, 4, 5, 5, 7, 8, 10, 15, 20, 25]
```

karld.merger.**sort_iterables**(*iterables*, *key=None*)

karld.merger.**sort_merge_group**(*iterables*, *key=None*)

karld.merger.**sorted_by**(*key*, *items*)

3.7 run_together Module

karld.run_together.**csv_file_consumer**(*csv_rows_consumer*, *file_path_name*)
Consume the file at file_path_name as a csv file, passing it through csv_rows_consumer.

Parameters

- **csv_rows_consumer** (*callable*) – consumes data_items yielding collection for each
- **file_path_name** (*str, str*) – path to input csv file

karld.run_together.**csv_file_to_file**(*csv_rows_consumer*, *out_prefix*, *out_dir*, *file_path_name*)
Consume the file at file_path_name as a csv file, passing it through csv_rows_consumer, writing the results as a csv file into out_dir as the same name, lowered, and prefixed.

Parameters

- **csv_rows_consumer** (*callable*) – consumes data_items yielding collection for each
- **out_prefix** (*str*) – prefix out_file_name
- **out_dir** (*str*) – directory to write output file to
- **file_path_name** (*str, str*) – path to input csv file

karld.run_together.**csv_files_to_file**(*csv_rows_consumer*, *out_prefix*, *out_dir*, *out_file_name*, *file_path_names*)
Consume the file at file_path_name as a csv file, passing it through csv_rows_consumer, writing the results as a csv file into out_dir as the same name, lowered, and prefixed.

Parameters

- **csv_rows_consumer** – consumes data_items yielding collection for each
- **out_prefix** (*str*) – prefix out_file_name
- **out_dir** (*str*) – Directory to write output file to.
- **out_file_name** (*str*) – Output file base name.
- **file_path_names** (*str, str*) – tuple of paths and basenames to input csv files

karld.run_together.**distribute_multi_run_to_runners**(*items_func*, *in_dir*, *reader=None*, *walker=None*, *batch_size=1100*, *filter_func=None*)
With a multi-process pool, map batches of items from multiple files to an items processing function.

The reader callable should be as fast as possible to reduce data feeder cpu usage. It should do the minimal to produce discrete units of data, save any decoding for the items function.

Parameters

- **items_func** – Callable that takes multiple items of the data.
- **reader** – URL reader callable.
- **walker** – A generator that takes the in_dir URL and emits url, name tuples.

- **batch_size** – size of batches.
- **filter_func** – a function that returns True for desired paths names.

karld.run_together.**distribute_run_to_runners**(*items_func*, *in_url*, *reader=None*, *batch_size=1100*)

With a multi-process pool, map batches of items from file to an items processing function.

The reader callable should be as fast as possible to reduce data feeder cpu usage. It should do the minimal to produce discrete units of data, save any decoding for the items function.

Parameters

- **items_func** – Callable that takes multiple items of the data.
- **reader** – URL reader callable.
- **in_url** – Url of content
- **batch_size** – size of batches.

karld.run_together.**multi_in_single_out**(*rows_reader*, *rows_writer*, *rows_iter_consumer*, *out_url*, *in_urls_func*)

Multi input combiner.

Parameters

- **rows_reader** – function to read a file path and returns an iterator
- **rows_writer** – function to write values
- **rows_iter_consumer** – function takes iter. of iterators returns iter.
- **out_url** – url for the rows_writer to write to.
- **in_urls_func** – function generates iterator of input urls.

karld.run_together.**pool_run_files_to_files**(*file_to_file*, *in_dir*, *filter_func=None*)

With a multi-process pool, map files in in_dir over file_to_file function.

Parameters

- **file_to_file** – callable that takes file paths.
- **in_dir** – path to process all files from.
- **filter_func** – Takes a tuple of path and base name of a file and returns a bool.

Returns A list of return values from the map.

karld.run_together.**serial_run_files_to_files**(*file_to_file*, *in_dir*, *filter_func=None*)

With a map files in in_dir over the file_to_file function.

Using this to debug your file_to_file function can make it easier.

Parameters

- **file_to_file** – callable that takes file paths.
- **in_dir** – path to process all files from.
- **filter_func** – Takes a tuple of path and base name of a file and returns a bool.

Returns A list of return values from the map.

3.8 unicode_io Module

3.8.1 How To Encoding

If you've tried something like `unicode('')` or `u'hello ' + 'wrld' or ``str(u'wörld')` you will have seen UnicodeDecodeError and UnicodeEncodeError. Likely, you've tried to read csv data from a file and mixed the data with unicode and everything went fine until it got to the line with some word with an accent character and it broke and showed `UnicodeDecodeError: 'ascii' codec can't decode byte ...` What do you do?. You've tried to write sequences of unicode strings to a csv file and gotten `UnicodeEncodeError: 'ascii' codec can't encode character u'\xf6' in position 1: ordinal not in range(128)` What do you do?

Unicode handles characters used by different languages around the world, emojis, curly quotes and other *glyphs*. The textual data in different parts of the world can have various encodings designed to specifically handle their glyphs and unicode can represent them all, but the data must be decoded from that encoding to unicode.

The data was written to the file in a specific encoding, either deliberately or because that was the default for the software. Unfortunately, it's up to the reader of the data to know what the data was encoded in. It can be connected to the language or locale it was created in. Sometimes it can be inferred by the data. Many times it's written in utf-8, which can handle encoding all the different chars that can be in a unicode string. It does this by saving chars like `'¥'`, or in unicode, `u'\xa5'`, as `'\xc2\xa5'`. `u'\xa5'.encode('utf-8')` results in `'\xc2\xa5'`. It uses more space, but can do it. By the way, `'¥'` is possible in this code because the encoding is declared at the top of this file.

String transformation methods, such as upper() or lower() don't work on these chars, like `'î'` or `'ê'` if they are encoded as a utf-8 string, but will work if they are decoded from utf-8 to unicode.

```
>>> print 'î'.upper()
î
>>> print u'î'.upper()
Î
>>> print 'ê'.upper()
ê
>>> print 'ê'.decode('utf-8').upper()
Ê
```

The python 2.7 csv module doesn't work with unicode, so the text it parses must be encoded from unicode to a str using an encoding that will handle all the chars in the text. utf-8 is good choice, and thus is default.

The purpose of this module is to facilitate reading and writing csv data in whatever encoding your data is in.

karld.unicode_io.**csv_reader**(*csv_data*, *dialect=<class csv.excel>*, *encoding='utf-8'*, ***kwargs*)

Csv row generator that re-encodes to unicode from csv data with a given encoding.

Utf-8 data in, unicode out. You may specify a different encoding of the incoming data.

Parameters

- **csv_data** – An iterable of str of the specified encoding.
- **dialect** – csv dialect
- **encoding** – The encoding of the given data.

karld.unicode_io.**get_csv_row_writer**(*stream*, *dialect=<class csv.excel>*, *encoding='utf-8'*, ***kwargs*)

Create a csv, encoding from unicode, row writer.

Use returned callable to write rows of unicode data to a stream, such as a file opened in write mode, in utf-8(or another) encoding.

```
my_row_data = [
    [u'one', u'two'],
    [u'three', u'four'],
]

with open('myfile.csv', 'wt') as myfile:
    unicode_row_writer = get_unicode_row_writer(myfile)
    for row in my_row_data:
        unicode_row_writer(row)
```

karld.unicode_io.**csv_unicode_reader**(*unicode_csv_data*, *dialect=<class csv.excel>*, ***kwargs*)

Generator the reads serialized unicode csv data. Use this if you have a stream of data in unicode and you want to access the rows of the data as sequences encoded as unicode.

Unicode in, unicode out.

Parameters

- **unicode_csv_data** – An iterable of unicode strings.
- **dialect** – csv dialect

3.9 path Module

karld.path.**i_walk_csv_paths**(*input_dir*)

Generator to yield the paths of csv files in the input directory.

Parameters **input_dir** – path to the input directory

karld.path.**i_walk_json_paths**(*input_dir*)

Generator to yield the paths of json files in the input directory.

Parameters **input_dir** – path to the input directory

3.10 io Module

3.11 tap Module

3.12 Subpackages

3.12.1 tests Package

test_conversion_operators Module

class karld.tests.test_conversion_operators.**TestConversion**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_apply_conversion_map()

Ensure all the items in the conversion_map is applied.

test_apply_conversion_map_map()
Ensure all the items in the conversion_map is applied.

test_get_number_as_int()
Ensure a string with a number prefix returns an int.

test_must_int()
Ensure a string not castable to a number raises a ValueError

class karld.tests.test_conversion_operators.**TestGettersValueJoiner**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_join_stripped_gotten_value()
Ensure joiner gets the values from data with the getters, coerce to str, strips padding whitespace and join with the separator.

class karld.tests.test_conversion_operators.**TestValueJoiner**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_join_stripped_values()
Ensure joiner gets the values from data with the getter, coerce to str, strips padding whitespace and join with the separator.

test_loadump Module

test_run_together Module

test_unicode_io Module

class karld.tests.test_unicode_io.**TestCSVToUnicodeReader**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Ensure unicode reader reads contents of a file iteratively and produces unicode sequences.

setUp()

test_iterative()
Ensure unicode reader consumes the data iteratively.

test_unicoded()
Ensure utf-8 strings in the data are converted to unicode sequences.

class karld.tests.test_unicode_io.**TestUnicodeCSVRowWriter**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Ensure get_csv_row_writer returns a function that will write a row of data to a stream.

setUp()

test_write_unicode()
Ensure the function returned from get_csv_row_writer will write a row to the io stream.

class karld.tests.test_unicode_io.**TestUnicodeToUnicodeCSVReader**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Ensure the unicode csv reader is iterative, and consumes an iterator of unicode strings that are delimited and split them into sequences of unicode strings.

test_iterative()
Ensure csv_unicode_reader consumes iteratively.

test_unicode_to_csv_unicode()
Ensure that a stream of unicode strings are converted to sequences by parsing with the csv module.

3.12.2 karld.record_reader Package

karld.record_reader Package

When your data can be divided into logical units, but is each unit takes up varying amounts of multiple lines of a file, use this to consume them in those units. Just provide a function that takes a line and tells if it's a start line or not.

karld.record_reader.__init__.**multi_line_records**(*lines*, *is_line_start=None*)
Iterate over lines, yielding a sequence for group of lines that end where the next multi-line record begins. The beginning of the record is determined by calling the given is_line_delimiter function, which is called on the every line.

Parameters

- **lines** – An iterator of unicode lines.
- **is_line_start** (*callable that returns if a line is the beginning of a record.*) – determine the beginning line of a record.

Yields deque of lines.

app_engine Module

karld.record_reader.app_engine.**is_log_start_line**(*line*)
Is the line the start of a request log from Google App Engine.

Parameters **line** – A string.

Returns True if the line doesn't start with a tab.

karld.record_reader.app_engine.**log_reader**(*file_path*)
Iterate over request logs as written by a Google App Engine app.

Parameters **file_path** – Path to an App Engine log file.

Returns An iterator of multi-line log records.

CHAPTER 4

example scripts

4.1 split_multiline Module

Run this script first to split the example data, which has multiple lines in some fields.

split_multiline.**main**()

4.2 clean Module

Run this script to 'clean' the split up data.

clean.**contrived_cleaner**(*data_items*)

Sort the data by the second row, enumerate it, apply title case to every field and include the original index and sorted in the in the row.

Parameters **data_items** – A sequence of unicode strings

clean.**main**(**args*)

Try it:

```
python clean.py
```

or:

```
python clean.py --pool True
```

or:

```
python clean.py --in-dir split_data_ml/data --out-dir my_clean_data
```

or:

```
python clean.py --pool True --in-dir split_data_ml/data
```

clean.**run**(*in_dir*, *out_dir*, *pool*)

4.3 split_non_multiline Module

Run this script first to split the example data that does not have any multiple line in fields.

split_non_multiline.**main**()

4.4 shard_data Module

Shard out data to files.

shard_data.**main**()
Python 2 version

4.5 shard_to_csv Module

Shard out data to csv files.

shard_to_csv.**main**()
From a source of data, shard it to csv files.

4.6 shard_to_json Module

Shard out data to files as rows of JSON.

shard_to_json.**main**()
From a source of data, shard it to csv files.

4.7 consume_csv_file Module

Iteratively consume csv file.

consume_csv_file.**main**()
Iterate over a the row of a csv file, extracting the data you desire.

4.8 consume_many_csv_files Module

Consume the items of a directory of csv files as if they were one file.

consume_many_csv_files.**main**()
Consume many csv files as if one.

4.9 concat_csv_files Module

Concatenate all the csv files in a directory together.

concat_csv_files.**main**()
Concatenate csv files together in no particular order.

4.10 merge_small_csv_files Module

Merge a number of homogeneous small csv files on a key. Small means they all together fit in your computer's memory.

merge_small_csv_files.**main**()
Merge a number of homogeneous small csv files on a key. Small means they all together fit in your computer's memory.

4.11 tap_example Module

Uses tap to get information from a stream of data in csv files.

4.12 stream_searcher Module

Uses tap to get information from a stream of data in csv files in designated directory with optional multi-processing.

stream_searcher.**certain_kind_tap**(*data_items*)
As the stream of data items go by, get different kinds of information from them, in this case, the things that are fruit and metal, collecting each kind with a different spigot.

stream_tap doesn't consume the data_items iterator by itself, it's a generator and must be consumed by something else. In this case, it's consuming the items by casting the iterator to a tuple, but doing it in batches.

Since each batch is not referenced by anything the memory can be freed by the garbage collector, so no matter the size of the data_items, only a little memory is needed. The only things retained are the results, which should just be a subset of the items and in this case, the getter functions only return a portion of each item it matches.

Parameters **data_items** – A sequence of unicode strings

stream_searcher.**get_fruit**(*item*)
Get things that are fruit.

Returns thing of item if it's a fruit

stream_searcher.**get_metal**(*item*)
Get things that are metal.

Returns thing of item if it's metal

stream_searcher.**main**(**args*)
Try it:

```
python stream_searcher.py
```

or:

```
python stream_searcher.py --pool True
```

or:

```
python stream_searcher.py --in-dir test_data/things_kinds
```

or:

```
python stream_searcher.py --pool True --in-dir test_data/things_kinds
```

stream_searcher.**run**(*in_dir*, *pool*)
Run the composition of csv_file_consumer and information tap with the csv files in the input directory, and collect the results from each file and merge them together, printing both kinds of results.

stream_searcher.**run_distribute**(*in_path*)

Run the composition of csv_file_consumer and information tap with the csv files in the input directory, and collect the results from each file and merge them together, printing both kinds of results.

stream_searcher.**run_distribute_multi**(*in_dir*)

Run the composition of csv_file_consumer and information tap with the csv files in the input directory, and collect the results from each file and merge them together, printing both kinds of results.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

c

k

m

s

L

M

P

R

S

T

W

Y