photosbyjeremy add files `901332b` on May 7

1 contributor

325 lines (263 sloc) 11.8 KB

```
# -*- coding: utf-8 -*-
# PyExifTool <http://github.com/smarnach/pyexiftool>
# Copyright 2012 Sven Marnach

# This file is part of PyExifTool.
#
# PyExifTool is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the licence, or
# (at your option) any later version, or the BSD licence.
#
# PyExifTool is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#
# See COPYING.GPL or COPYING.BSD for more details.

"""
PyExifTool is a Python library to communicate with an instance of Phil
Harvey's excellent ExifTool_ command-line application.  The library
provides the class :py:class:`ExifTool` that runs the command-line
tool in batch mode and features methods to send commands to that
program, including methods to extract meta-information from one or
more image files.  Since ``exiftool`` is run in batch mode, only a
single instance needs to be launched and can be reused for many
queries.  This is much more efficient than launching a separate
process for every single query.

.. _ExifTool: http://www.sno.phy.queensu.ca/~phil/exiftool/

The source code can be checked out from the github repository with

::

    git clone git://github.com/smarnach/pyexiftool.git

Alternatively, you can download a tarball_.  There haven't been any
releases yet.

.. _tarball: https://github.com/smarnach/pyexiftool/tarball/master

PyExifTool is licenced under GNU GPL version 3 or later.

Example usage::

    import exiftool

    files = ["a.jpg", "b.png", "c.tif"]
    with exiftool.ExifTool() as et:
        metadata = et.get_metadata_batch(files)
    for d in metadata:
        print("{:20.20} {:20.20}".format(d["SourceFile"],
                                         d["EXIF:DateTimeOriginal"]))
"""

```

```
from __future__ import unicode_literals

import sys
import subprocess
import os
import json
import warnings
import codecs

try:        # Py3k compatibility
    basestring
except NameError:
    basestring = (bytes, str)

executable = "exiftool"
"""The name of the executable to run.

If the executable is not located in one of the paths listed in the
``PATH`` environment variable, the full path should be given here.
"""

# Sentinel indicating the end of the output of a sequence of commands.
# The standard value should be fine.
sentinel = b"{ready}"

# The block size when reading from exiftool.  The standard value
# should be fine, though other values might give better performance in
# some cases.
block_size = 4096

# This code has been adapted from Lib/os.py in the Python source tree
# (sha1 265e36e277f3)
def _fscodec():
    encoding = sys.getfilesystemencoding()
    errors = "strict"
    if encoding != "mbcs":
        try:
            codecs.lookup_error("surrogateescape")
        except LookupError:
            pass
        else:
            errors = "surrogateescape"

    def fsencode(filename):
        """
        Encode filename to the filesystem encoding with 'surrogateescape' error
        handler, return bytes unchanged. On Windows, use 'strict' error handler if
        the file system encoding is 'mbcs' (which is the default encoding).
        """
        if isinstance(filename, bytes):
            return filename
        else:
            return filename.encode(encoding, errors)

    return fsencode

fsencode = _fscodec()
del _fscodec

class ExifTool(object):
    """Run the `exiftool` command-line tool and communicate to it.

    You can pass the file name of the ``exiftool`` executable as an
    argument to the constructor.  The default value ``exiftool`` will
    only work if the executable is in your ``PATH``.

    Most methods of this class are only available after calling
```

```
    :py:meth:`start()`, which will actually launch the subprocess.  To
    avoid leaving the subprocess running, make sure to call
    :py:meth:`terminate()` method when finished using the instance.
    This method will also be implicitly called when the instance is
    garbage collected, but there are circumstance when this won't ever
    happen, so you should not rely on the implicit process
    termination.  Subprocesses won't be automatically terminated if
    the parent process exits, so a leaked subprocess will stay around
    until manually killed.

    A convenient way to make sure that the subprocess is terminated is
    to use the :py:class:`ExifTool` instance as a context manager::

        with ExifTool() as et:
            ...

    .. warning:: Note that there is no error handling.  Nonsensical
       options will be silently ignored by exiftool, so there's not
       much that can be done in that regard.  You should avoid passing
       non-existent files to any of the methods, since this will lead
       to undefied behaviour.

    .. py:attribute:: running

       A Boolean value indicating whether this instance is currently
       associated with a running subprocess.
    """

    def __init__(self, executable_=None):
        if executable_ is None:
            self.executable = executable
        else:
            self.executable = executable_
        self.running = False

    def start(self):
        """Start an ``exiftool`` process in batch mode for this instance.

        This method will issue a ``UserWarning`` if the subprocess is
        already running.  The process is started with the ``-G`` and
        ``-n`` as common arguments, which are automatically included
        in every command you run with :py:meth:`execute()`.
        """
        if self.running:
            warnings.warn("ExifTool already running; doing nothing.")
            return
        with open(os.devnull, "w") as devnull:
            self._process = subprocess.Popen(
                [self.executable, "-stay_open", "True",  "-@", "-",
                 "-common_args", "-G", "-n"],
                stdin=subprocess.PIPE, stdout=subprocess.PIPE,
                stderr=devnull)
        self.running = True

    def terminate(self):
        """Terminate the ``exiftool`` process of this instance.

        If the subprocess isn't running, this method will do nothing.
        """
        if not self.running:
            return
        self._process.stdin.write(b"-stay_open\nFalse\n")
        self._process.stdin.flush()
        self._process.communicate()
        del self._process
        self.running = False

```

```
    def __enter__(self):
        self.start()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.terminate()

    def __del__(self):
        self.terminate()

    def execute(self, *params):
        """Execute the given batch of parameters with ``exiftool``.

        This method accepts any number of parameters and sends them to
        the attached ``exiftool`` process.  The process must be
        running, otherwise ``ValueError`` is raised.  The final
        ``-execute`` necessary to actually run the batch is appended
        automatically; see the documentation of :py:meth:`start()` for
        the common options.  The ``exiftool`` output is read up to the
        end-of-output sentinel and returned as a raw ``bytes`` object,
        excluding the sentinel.

        The parameters must also be raw ``bytes``, in whatever
        encoding exiftool accepts.  For filenames, this should be the
        system's filesystem encoding.

        .. note:: This is considered a low-level method, and should
           rarely be needed by application developers.
        """
        if not self.running:
            raise ValueError("ExifTool instance not running.")
        self._process.stdin.write(b"\n".join(params + (b"-execute\n",)))
        self._process.stdin.flush()
        output = b""
        fd = self._process.stdout.fileno()
        while not output[-32:].strip().endswith(sentinel):
            output += os.read(fd, block_size)
        return output.strip()[:-len(sentinel)]

    def execute_json(self, *params):
        """Execute the given batch of parameters and parse the JSON output.

        This method is similar to :py:meth:`execute()`.  It
        automatically adds the parameter ``-j`` to request JSON output
        from ``exiftool`` and parses the output.  The return value is
        a list of dictionaries, mapping tag names to the corresponding
        values.  All keys are Unicode strings with the tag names
        including the ExifTool group name in the format <group>:<tag>.
        The values can have multiple types.  All strings occurring as
        values will be Unicode strings.  Each dictionary contains the
        name of the file it corresponds to in the key ``"SourceFile"``.

        The parameters to this function must be either raw strings
        (type ``str`` in Python 2.x, type ``bytes`` in Python 3.x) or
        Unicode strings (type ``unicode`` in Python 2.x, type ``str``
        in Python 3.x).  Unicode strings will be encoded using
        system's filesystem encoding.  This behaviour means you can
        pass in filenames according to the convention of the
        respective Python version – as raw strings in Python 2.x and
        as Unicode strings in Python 3.x.
        """
        params = map(fsencode, params)
        return json.loads(self.execute(b"-j", *params).decode("utf-8"))

    def get_metadata_batch(self, filenames):
        """Return all meta-data for the given files.
```

```
        The return value will have the format described in the
        documentation of :py:meth:`execute_json()`.
        """
        return self.execute_json(*filenames)

    def get_metadata(self, filename):
        """Return meta-data for a single file.

        The returned dictionary has the format described in the
        documentation of :py:meth:`execute_json()`.
        """
        return self.execute_json(filename)[0]

    def get_tags_batch(self, tags, filenames):
        """Return only specified tags for the given files.

        The first argument is an iterable of tags.  The tag names may
        include group names, as usual in the format <group>:<tag>.

        The second argument is an iterable of file names.

        The format of the return value is the same as for
        :py:meth:`execute_json()`.
        """
        # Explicitly ruling out strings here because passing in a
        # string would lead to strange and hard-to-find errors
        if isinstance(tags, basestring):
            raise TypeError("The argument 'tags' must be "
                            "an iterable of strings")
        if isinstance(filenames, basestring):
            raise TypeError("The argument 'filenames' must be "
                            "an iterable of strings")
        params = ["-" + t for t in tags]
        params.extend(filenames)
        return self.execute_json(*params)

    def get_tags(self, tags, filename):
        """Return only specified tags for a single file.

        The returned dictionary has the format described in the
        documentation of :py:meth:`execute_json()`.
        """
        return self.get_tags_batch(tags, [filename])[0]

    def get_tag_batch(self, tag, filenames):
        """Extract a single tag from the given files.

        The first argument is a single tag name, as usual in the
        format <group>:<tag>.

        The second argument is an iterable of file names.

        The return value is a list of tag values or ``None`` for
        non-existent tags, in the same order as ``filenames``.
        """
        data = self.get_tags_batch([tag], filenames)
        result = []
        for d in data:
            d.pop("SourceFile")
            result.append(next(iter(d.values()), None))
        return result

    def get_tag(self, tag, filename):
        """Extract a single tag from a single file.

        The return value is the value of the specified tag, or
        ``None`` if this tag was not found in the file.
```

```
        """
        return self.get_tag_batch(tag, [filename])[0]
```