```python
"""
file: main_template.py
author: xxxxx
description: template for a command line python script
"""


import os
import sys
import logging
import errno
import time
import traceback
import binascii
from stat import *


class Application:
    """
    This class defines the functionality for the script. It is instantiated in
the global
    processing handler for __main__
    """
    def __init__(self):
        """
        Setups the member variable for the application object.
        """
        import datetime
        from stat import ST_MTIME

        # set the script name variable
        self.scriptname = sys.argv[0]
        # initialize logging, it depends on the script name variable for the
file name
        self.logger()
        # check platform, it depends on logging being initialized
        self.scriptplatform = self.platform()
        # these are needed by self.usage()
        self.versionno = "1.0"
        self.build = str(datetime.date.fromtimestamp(os.stat(self.scriptname)
[ST_MTIME])).replace("-", ".")

        #
        # Member variables for the application.
        #
        self.InputFiles = []

        # Turn this on to true if the unittest() function should run by default
e.g.
        # currently requires the caller to specifically invoke the -u option.
        self.RunUnitTest = True

        return

    # dependencies: self.scriptname
    #
    def logger(self):
        """
        Setups the python logging for application. By default it logs to both
        the console window and to a log file in the current directory.
```

```python
            """
            import logging
            import logging.handlers

            try:
                    (logfile, ext) =
os.path.splitext(os.path.basename(self.scriptname));
                    logging.basicConfig(
                            filename = logfile + ".log",
                            format = '%(asctime)s %(levelname)s %(message)s',
                            datefmt = '%Y-%m-%d %H:%M:%S',
                            filemode = 'a',
                            level = logging.DEBUG)
                    console = logging.StreamHandler()
                    console.setLevel(logging.INFO)
                    formatter = logging.Formatter('%(message)s')
                    console.setFormatter(formatter)
                    logging.getLogger('').addHandler(console)

                    # print a self test logging message
                    logging.info('log file %s.log' % (logfile))
                    logging.info('logging started at %s' % (time.strftime("%Y-%m-%d
%H:%M:%S", time.gmtime())))

            except Exception, e:
                    print >> sys.stderr, 'Failed to initialize logging. ' + str(e)
                    raise e

            return

    # dependencies: self.logger()
    #
    def platform(self):
            import platform

            scriptplatform = platform.system().lower()

            if scriptplatform == 'windows' or scriptplatform == 'microsoft':
                    logging.info("platform %s" % (scriptplatform))
                    return 'windows'
            elif ("linux" == scriptplatform):
                    strDistro = os.popen('lsb_release -i').read()
                    import re
                    p = re.compile('Distributor ID:\s+(.*)')
                    arrDistro = p.findall(strDistro)
                    linuxdistro = arrDistro[0].lower()
                    logging.info("platform %s" % (linuxdistro))
                    if (not len(linuxdistro) > 0):
                            raise StandardError("Unable to determine linux distribution
on this platform.")

                    if 'ubuntu' == linuxdistro:
                            return 'linux'
                    elif 'fedoracore' == linuxdistro or 'fedora' == linuxdistro:
                            return 'fedora'
            else:
                    logging.info("platform %s" % (scriptplatform))

            return 'other'
```

```python
        # dependencies: self.versionno, self.build
        #
        def version(self):
                """
                Formats the version number of the script as a string.
                """
                return "%s.%s" % (self.versionno, self.build)

        #
        # utility functions
        #

        def environ(self, name, default = None):
                """
                Helper method for looking up environment variables. Writes a warning to
                the application log, if the   environment variable can not be found. It
does
                not propagate the exception if the environment variable does not exist.
                """

                value = ""
                try:
                        value = os.environ[ name ]
                except Exception:
                        logging.warning("Environment variable '%s' does not exist" %
(name))
                        if None != default:
                                value = default

                return value

        def shellspawn(self, binPath, binArgs = None):
                """
                Runs a shell command and does not wait for the command to complete.
                """

                if None != binArgs and len(binArgs) > 0:
                        # verify that the binary file does exist
                        if not os.path.exists(binPath):
                                logging.warning('Unable to locate binary image \'%s\'!' %
binPath)

                        # build the command line for the binary file
                        commandLine = "%s %s" % (binPath, binArgs)

                        # execute the command line
                        logging.info("shellspawn: %s" % (commandLine))
                        spawnArgs = binArgs.split(" ")
                        os.spawnv(os.P_NOWAIT, binPath, spawnArgs)
                else:
                        # build the command line for the binary file
                        commandLine = "%s" % (binPath)

                        # execute the command line
                        logging.info("shellspawn: %s" % (commandLine))
                        os.spawnl(os.P_NOWAIT, binPath)

                return True
```

```python
    def shellexec(self, binPath, binArgs = None):
        """
        Runs a shell command and waits for the command to complete before
returning.
        """

        if None != binArgs and len(binArgs) > 0:
            # verify that the binary file does exist
            if not os.path.exists(binPath):
                logging.warning('Unable to locate binary image \'%s\'!' %
binPath)

            # build the command line for the binary file
            commandLine = "\"%s\" %s" % (binPath, binArgs)
        else:
            # build the command line for the binary file
            commandLine = "%s" % (binPath)

        # execute the command line
        logging.info("shellexec: %s" % (commandLine))
        #retcode = os.system(commandLine)
        strStdOut = os.popen("\"%s\"" % (commandLine)).read()
        if None != strStdOut and len(strStdOut) > 0:
            logging.info(strStdOut)

        return True

    def copyExistingFile(self, srcFile, dstFile):
        """
        Copies a file from the source to the destination. Imports the python
modules
        locally so it is a nice copy and paste function. It logs warnings or
errors
        if it can not find the file as expected. A successful operation
generates no
        logs.
        """

        import shutil
        import stat
        from stat import S_ISDIR

        # verify that the binary file does exist
        if not os.path.exists(srcFile):
            logging.warning('MISSING: \'%s\'' % srcFile)
            return False

        # lookup the path attributes
        attrib = os.stat(srcFile)

        # if the file is read-only
        if (stat.S_ISDIR(attrib.st_mode)):
            logging.error('Source file \'%s\' is a directory!' % srcFile)
            return False

        # check if the dst file already exists
        if os.path.exists(dstFile):
            logging.warning('REPLACING: \'%s\'' % dstFile)
```

```python
            shutil.copyfile(srcFile, dstFile)

            # verify that the dst file now exists
            if not os.path.exists(dstFile):
                    logging.error('MISSING: \'%s\'' % dstFile)
                    return False

            return True

    #
    # Application functionality
    #

    def unittest(self):
            """
            This is the default action for the application. It should generally be
a self
            test.
            """

            blockSizeInBytes = 1024

            for f in self.InputFiles:
                        print("file: %s" % (f))

                        # lookup the path attributes
                        attrib = os.stat(f)

                        cb = attrib.st_size
                        print("size: %d" % (cb))
                        for i in xrange(0, cb, blockSizeInBytes):
                                blockLength = blockSizeInBytes
                                if (i + blockSizeInBytes > cb):
                                        blockLength = cb - i
                                #print("block %d[%d]" % (i, blockLength))
                                nf = "%s-%d(%d).exe" %
(os.path.splitext(os.path.basename(f))[0], i, blockLength)
                                #print("%s\n" % nf)

                                print("writing %s" % nf)
                                fd1 = open(f, "rb")
                                fd2 = open(nf, "wb")
                                randBlock = os.urandom(blockLength)

                                if (i > 0):
                                        d1 = fd1.read(i)
                                        fd2.write(d1)
                                        fd2.write(randBlock)

                                else:
                                        fd2.write(randBlock)

                                fd1.seek(i + blockLength, os.SEEK_SET)
                                cbRemain = cb - (i + blockLength)
                                d2 = fd1.read(cbRemain)
                                fd2.write(d2)

                                fd1.close()
```

```python
                                fd2.close()

                return

        def usage(self):
                """
                Prints the help text for the application.
                """
                print >> sys.stderr, os.path.basename(self.scriptname)
                print >> sys.stderr, 'Version:', self.version()
                print >> sys.stderr, ''
                print >> sys.stderr, 'This application ...'
                print >> sys.stderr, ''
                print >> sys.stderr, 'Usage:', os.path.basename(self.scriptname), '
[-?]'
                print >> sys.stderr, ''
                print >> sys.stderr, '    -? Prints this message.'
                print >> sys.stderr, '     -o Specifies the output folder or files.'
                print >> sys.stderr, '     -t Specifies the test folder.'
                print >> sys.stderr, ''
                print >> sys.stderr, 'Examples:'
                print >> sys.stderr, ''
                print >> sys.stderr, os.path.basename(self.scriptname), '-i foo'
                print >> sys.stderr, ''
                return

        def main(self, argv):
                """
                Entry point for the application. Command line processing should be done
here.
                Note the usage() function is nearby and should be updated as well.
                """
                self.logger()

                # parse the command line using getopt
                import getopt
                long_opts = []
                long_opts.append('help')
                long_opts.append('unittest')
                long_opts.append('input=')
                long_opts.append('output=')
                short_opts = ''
                # default the short option string based on the long options if not
specified
                if 0 == len(short_opts):
                        sopts = []
                        for lopt in long_opts:
                                if 0 == len(lopt):
                                        continue
                                sopts.append(lopt[0])
                                if '=' == lopt[len(lopt) - 1]:
                                        sopts.append(':')
                        short_opts = ''.join(sopts)

                import getopt
                (opts, args) = getopt.getopt(argv[1:], short_opts, long_opts)
                for opt in opts:
                        if opt[0] == '-?' or opt[0] == '--help':
                                return self.usage()
```

```python
                    if opt[0] == '-u' or opt[0] == '--unittest':
                        self.RunUnitTest = True
                    if opt[0] == '-o' or opt[0] == '--output':
                        if not os.path.exists(opt[1]):
                            if not os.mkdir(opt[1]):
                                print >> sys.stderr, ('Could not create
folder \'%s\'!' % opt[1])

                                return
                        self.OutputDirectory = opt[1]
                    if opt[0] == '-i' or opt[0] == '--input':
                        if os.path.exists(opt[1]):
                            self.InputFiles.append(opt[1])
                        else:
                            print >> sys.stderr, ('File \'%s\' not found!' %
opt[1])

                            return

            # if specified run the unit test algorithm
            if self.RunUnitTest:
                self.unittest()

            # otherwise validate the command line and do something useful

            return

# call the script main function
if __name__ == "__main__":
    application = Application()
    sys.exit(application.main(sys.argv))
```