#Modded just for Kodi

############################################################################

# /T /I #

# / |/ | .-~/ #

# T\ Y I |/ / \_ #

# /T | \I | I Y.-~/ #

# I l /I T\ | | l | T / #

# T\ | \ Y l /T | \I l \ ` l Y #

# \_\_ | \l \l \I l \_\_l l \ ` \_. | #

# \ ~-l `\ `\ \ \ ~\ \ `. .-~ | #

# \ ~-. "-. ` \ ^.\_ ^. "-. / \ | #

#.--~-.\_ ~- ` \_ ~-\_.-"-." .\_ /.\_ ." ./ #

# >--. ~-. .\_ ~>-" "\ 7 7 ] #

#^.\_\_\_~"--.\_ ~-{ .-~ . `\ Y . / | #

# <\_\_ ~"-. ~ /\_/ \ \I Y : | #

# ^-.\_\_ ~(\_/ \ >.\_: | l\_\_\_\_\_\_ #

# ^--.,\_\_\_.-~" /\_/ ! `-.~"--l\_ / ~"-. #

# (\_/ . ~( /' "~"--,Y -=b-. \_) #

# (\_/ . \ : / l c"~o \ #

# \ / `. . .^ \\_.-~"~--. ) #

# (\_/ . ` / / ! )/ #

# / / \_. '. .': / ' #

# ~(\_/ . / \_ ` .-<\_ #

# /\_/ . ' .-~" `. / \ \ ,z=. Python Team #

# ~( / ' : | K "-.~-.\_\_\_\_\_\_// Original Author #

# "-,. l I/ \\_ \_\_{--->.\_(==. #

# //( \ < ~"~" // #

# /' /\ \ \ ,v=. (( Fire TV Guru #

# .^. / /\ " }\_\_ //===- ` Modded for Kodi 18 #

# / / ' ' "-.,\_\_ {---(==- #

# .^ ' : T ~" ll #

# / . . . : | :! \ #

# (\_/ / | | j-" ~^ #

# ~-<\_(\_.^-~" #

# #

############################################################################

import struct, os, time, sys, shutil

import binascii, cStringIO, stat

import io

import re

import string

try:

import zlib

crc32 = zlib.crc32

except ImportError:

zlib = None

crc32 = binascii.crc32

import xbmc

KODIV = float(xbmc.getInfoLabel("System.BuildVersion")[:4])

if KODIV > 17:

import xbmcaddon, xbmcgui, xbmcplugin #FTG mod for Kodi 18

else:pass

\_\_all\_\_ = ["BadZipfile", "error", "ZIP\_STORED", "ZIP\_DEFLATED", "is\_zipfile",

"ZipInfo", "ZipFile", "PyZipFile", "LargeZipFile" ]

class BadZipfile(Exception):

pass

class LargeZipFile(Exception):

"""

Raised when writing a zipfile, the zipfile requires ZIP64 extensions

and those extensions are disabled.

"""

error = BadZipfile

ZIP64\_LIMIT = (1 << 31) - 1

ZIP\_FILECOUNT\_LIMIT = (1 << 16) - 1

ZIP\_MAX\_COMMENT = (1 << 16) - 1

ZIP\_STORED = 0

ZIP\_DEFLATED = 8

structEndArchive = "<4s4H2LH"

stringEndArchive = "PK\005\006"

sizeEndCentDir = struct.calcsize(structEndArchive)

\_ECD\_SIGNATURE = 0

\_ECD\_DISK\_NUMBER = 1

\_ECD\_DISK\_START = 2

\_ECD\_ENTRIES\_THIS\_DISK = 3

\_ECD\_ENTRIES\_TOTAL = 4

\_ECD\_SIZE = 5

\_ECD\_OFFSET = 6

\_ECD\_COMMENT\_SIZE = 7

\_ECD\_COMMENT = 8

\_ECD\_LOCATION = 9

structCentralDir = "<4s4B4HL2L5H2L"

stringCentralDir = "PK\001\002"

sizeCentralDir = struct.calcsize(structCentralDir)

\_CD\_SIGNATURE = 0

\_CD\_CREATE\_VERSION = 1

\_CD\_CREATE\_SYSTEM = 2

\_CD\_EXTRACT\_VERSION = 3

\_CD\_EXTRACT\_SYSTEM = 4

\_CD\_FLAG\_BITS = 5

\_CD\_COMPRESS\_TYPE = 6

\_CD\_TIME = 7

\_CD\_DATE = 8

\_CD\_CRC = 9

\_CD\_COMPRESSED\_SIZE = 10

\_CD\_UNCOMPRESSED\_SIZE = 11

\_CD\_FILENAME\_LENGTH = 12

\_CD\_EXTRA\_FIELD\_LENGTH = 13

\_CD\_COMMENT\_LENGTH = 14

\_CD\_DISK\_NUMBER\_START = 15

\_CD\_INTERNAL\_FILE\_ATTRIBUTES = 16

\_CD\_EXTERNAL\_FILE\_ATTRIBUTES = 17

\_CD\_LOCAL\_HEADER\_OFFSET = 18

structFileHeader = "<4s2B4HL2L2H"

stringFileHeader = "PK\003\004"

sizeFileHeader = struct.calcsize(structFileHeader)

\_FH\_SIGNATURE = 0

\_FH\_EXTRACT\_VERSION = 1

\_FH\_EXTRACT\_SYSTEM = 2

\_FH\_GENERAL\_PURPOSE\_FLAG\_BITS = 3

\_FH\_COMPRESSION\_METHOD = 4

\_FH\_LAST\_MOD\_TIME = 5

\_FH\_LAST\_MOD\_DATE = 6

\_FH\_CRC = 7

\_FH\_COMPRESSED\_SIZE = 8

\_FH\_UNCOMPRESSED\_SIZE = 9

\_FH\_FILENAME\_LENGTH = 10

\_FH\_EXTRA\_FIELD\_LENGTH = 11

structEndArchive64Locator = "<4sLQL"

stringEndArchive64Locator = "PK\x06\x07"

sizeEndCentDir64Locator = struct.calcsize(structEndArchive64Locator)

structEndArchive64 = "<4sQ2H2L4Q"

stringEndArchive64 = "PK\x06\x06"

sizeEndCentDir64 = struct.calcsize(structEndArchive64)

\_CD64\_SIGNATURE = 0

\_CD64\_DIRECTORY\_RECSIZE = 1

\_CD64\_CREATE\_VERSION = 2

\_CD64\_EXTRACT\_VERSION = 3

\_CD64\_DISK\_NUMBER = 4

\_CD64\_DISK\_NUMBER\_START = 5

\_CD64\_NUMBER\_ENTRIES\_THIS\_DISK = 6

\_CD64\_NUMBER\_ENTRIES\_TOTAL = 7

\_CD64\_DIRECTORY\_SIZE = 8

\_CD64\_OFFSET\_START\_CENTDIR = 9

def \_check\_zipfile(fp):

try:

if \_EndRecData(fp):

return True # file has correct magic number

except IOError:

pass

return False

def is\_zipfile(filename):

"""Quickly see if a file is a ZIP file by checking the magic number.

The filename argument may be a file or file-like object too.

"""

result = False

try:

if hasattr(filename, "read"):

result = \_check\_zipfile(fp=filename)

else:

with open(filename, "rb") as fp:

result = \_check\_zipfile(fp)

except IOError:

pass

return result

def \_EndRecData64(fpin, offset, endrec):

"""

Read the ZIP64 end-of-archive records and use that to update endrec

"""

try:

fpin.seek(offset - sizeEndCentDir64Locator, 2)

except IOError:

return endrec

data = fpin.read(sizeEndCentDir64Locator)

if len(data) != sizeEndCentDir64Locator:

return endrec

sig, diskno, reloff, disks = struct.unpack(structEndArchive64Locator, data)

if sig != stringEndArchive64Locator:

return endrec

if diskno != 0 or disks != 1:

raise BadZipfile("zipfiles that span multiple disks are not supported")

fpin.seek(offset - sizeEndCentDir64Locator - sizeEndCentDir64, 2)

data = fpin.read(sizeEndCentDir64)

if len(data) != sizeEndCentDir64:

return endrec

sig, sz, create\_version, read\_version, disk\_num, disk\_dir, \

dircount, dircount2, dirsize, diroffset = \

struct.unpack(structEndArchive64, data)

if sig != stringEndArchive64:

return endrec

endrec[\_ECD\_SIGNATURE] = sig

endrec[\_ECD\_DISK\_NUMBER] = disk\_num

endrec[\_ECD\_DISK\_START] = disk\_dir

endrec[\_ECD\_ENTRIES\_THIS\_DISK] = dircount

endrec[\_ECD\_ENTRIES\_TOTAL] = dircount2

endrec[\_ECD\_SIZE] = dirsize

endrec[\_ECD\_OFFSET] = diroffset

return endrec

def \_EndRecData(fpin):

"""Return data from the "End of Central Directory" record, or None.

The data is a list of the nine items in the ZIP "End of central dir"

record followed by a tenth item, the file seek offset of this record."""

fpin.seek(0, 2)

filesize = fpin.tell()

try:

fpin.seek(-sizeEndCentDir, 2)

except IOError:

return None

data = fpin.read()

if (len(data) == sizeEndCentDir and

data[0:4] == stringEndArchive and

data[-2:] == b"\000\000"):

endrec = struct.unpack(structEndArchive, data)

endrec=list(endrec)

endrec.append("")

endrec.append(filesize - sizeEndCentDir)

return \_EndRecData64(fpin, -sizeEndCentDir, endrec)

maxCommentStart = max(filesize - (1 << 16) - sizeEndCentDir, 0)

fpin.seek(maxCommentStart, 0)

data = fpin.read()

start = data.rfind(stringEndArchive)

if start >= 0:

recData = data[start:start+sizeEndCentDir]

if len(recData) != sizeEndCentDir:

return None

endrec = list(struct.unpack(structEndArchive, recData))

commentSize = endrec[\_ECD\_COMMENT\_SIZE]

comment = data[start+sizeEndCentDir:start+sizeEndCentDir+commentSize]

endrec.append(comment)

endrec.append(maxCommentStart + start)

return \_EndRecData64(fpin, maxCommentStart + start - filesize,

endrec)

return None

class ZipInfo (object):

"""Class with attributes describing each file in the ZIP archive."""

\_\_slots\_\_ = (

'orig\_filename',

'filename',

'date\_time',

'compress\_type',

'comment',

'extra',

'create\_system',

'create\_version',

'extract\_version',

'reserved',

'flag\_bits',

'volume',

'internal\_attr',

'external\_attr',

'header\_offset',

'CRC',

'compress\_size',

'file\_size',

'\_raw\_time',

)

def \_\_init\_\_(self, filename="NoName", date\_time=(1980,1,1,0,0,0)):

self.orig\_filename = filename

null\_byte = filename.find(chr(0))

if null\_byte >= 0:

filename = filename[0:null\_byte]

if os.sep != "/" and os.sep in filename:

filename = filename.replace(os.sep, "/")

self.filename = filename

self.date\_time = date\_time

if date\_time[0] < 1980:

raise ValueError('ZIP does not support timestamps before 1980')

self.compress\_type = ZIP\_STORED

self.comment = ""

self.extra = ""

if sys.platform == 'win32':

self.create\_system = 0

else:

self.create\_system = 3

self.create\_version = 20

self.extract\_version = 20

self.reserved = 0

self.flag\_bits = 0

self.volume = 0

self.internal\_attr = 0

self.external\_attr = 0

def FileHeader(self, zip64=None):

"""Return the per-file header as a string."""

dt = self.date\_time

dosdate = (dt[0] - 1980) << 9 | dt[1] << 5 | dt[2]

dostime = dt[3] << 11 | dt[4] << 5 | (dt[5] // 2)

if self.flag\_bits & 0x08:

CRC = compress\_size = file\_size = 0

else:

CRC = self.CRC

compress\_size = self.compress\_size

file\_size = self.file\_size

extra = self.extra

if zip64 is None:

zip64 = file\_size > ZIP64\_LIMIT or compress\_size > ZIP64\_LIMIT

if zip64:

fmt = '<HHQQ'

extra = extra + struct.pack(fmt,

1, struct.calcsize(fmt)-4, file\_size, compress\_size)

if file\_size > ZIP64\_LIMIT or compress\_size > ZIP64\_LIMIT:

if not zip64:

raise LargeZipFile("Filesize would require ZIP64 extensions")

file\_size = 0xffffffff

compress\_size = 0xffffffff

self.extract\_version = max(45, self.extract\_version)

self.create\_version = max(45, self.extract\_version)

filename, flag\_bits = self.\_encodeFilenameFlags()

header = struct.pack(structFileHeader, stringFileHeader,

self.extract\_version, self.reserved, flag\_bits,

self.compress\_type, dostime, dosdate, CRC,

compress\_size, file\_size,

len(filename), len(extra))

return header + filename + extra

def \_encodeFilenameFlags(self):

if isinstance(self.filename, unicode):

try:

return self.filename.encode('ascii'), self.flag\_bits

except UnicodeEncodeError:

return self.filename.encode('utf-8'), self.flag\_bits | 0x800

else:

return self.filename, self.flag\_bits

def \_decodeFilename(self):

if self.flag\_bits & 0x800:

return self.filename.decode('utf-8')

else:

return self.filename

def \_decodeExtra(self):

extra = self.extra

unpack = struct.unpack

while len(extra) >= 4:

tp, ln = unpack('<HH', extra[:4])

if tp == 1:

if ln >= 24:

counts = unpack('<QQQ', extra[4:28])

elif ln == 16:

counts = unpack('<QQ', extra[4:20])

elif ln == 8:

counts = unpack('<Q', extra[4:12])

elif ln == 0:

counts = ()

else:

raise RuntimeError, "Corrupt extra field %s"%(ln,)

idx = 0

if self.file\_size in (0xffffffffffffffffL, 0xffffffffL):

self.file\_size = counts[idx]

idx += 1

if self.compress\_size == 0xFFFFFFFFL:

self.compress\_size = counts[idx]

idx += 1

if self.header\_offset == 0xffffffffL:

old = self.header\_offset

self.header\_offset = counts[idx]

idx+=1

extra = extra[ln+4:]

class \_ZipDecrypter:

"""Class to handle decryption of files stored within a ZIP archive.

ZIP supports a password-based form of encryption. Even though known

plaintext attacks have been found against it, it is still useful

to be able to get data out of such a file.

Usage:

zd = \_ZipDecrypter(mypwd)

plain\_char = zd(cypher\_char)

plain\_text = map(zd, cypher\_text)

"""

def \_GenerateCRCTable():

"""Generate a CRC-32 table.

ZIP encryption uses the CRC32 one-byte primitive for scrambling some

internal keys. We noticed that a direct implementation is faster than

relying on binascii.crc32().

"""

poly = 0xedb88320

table = [0] \* 256

for i in range(256):

crc = i

for j in range(8):

if crc & 1:

crc = ((crc >> 1) & 0x7FFFFFFF) ^ poly

else:

crc = ((crc >> 1) & 0x7FFFFFFF)

table[i] = crc

return table

crctable = \_GenerateCRCTable()

def \_crc32(self, ch, crc):

"""Compute the CRC32 primitive on one byte."""

return ((crc >> 8) & 0xffffff) ^ self.crctable[(crc ^ ord(ch)) & 0xff]

def \_\_init\_\_(self, pwd):

self.key0 = 305419896

self.key1 = 591751049

self.key2 = 878082192

for p in pwd:

self.\_UpdateKeys(p)

def \_UpdateKeys(self, c):

self.key0 = self.\_crc32(c, self.key0)

self.key1 = (self.key1 + (self.key0 & 255)) & 4294967295

self.key1 = (self.key1 \* 134775813 + 1) & 4294967295

self.key2 = self.\_crc32(chr((self.key1 >> 24) & 255), self.key2)

def \_\_call\_\_(self, c):

"""Decrypt a single character."""

c = ord(c)

k = self.key2 | 2

c = c ^ (((k \* (k^1)) >> 8) & 255)

c = chr(c)

self.\_UpdateKeys(c)

return c

compressor\_names = {

0: 'store',

1: 'shrink',

2: 'reduce',

3: 'reduce',

4: 'reduce',

5: 'reduce',

6: 'implode',

7: 'tokenize',

8: 'deflate',

9: 'deflate64',

10: 'implode',

12: 'bzip2',

14: 'lzma',

18: 'terse',

19: 'lz77',

97: 'wavpack',

98: 'ppmd',

}

class ZipExtFile(io.BufferedIOBase):

"""File-like object for reading an archive member.

Is returned by ZipFile.open().

"""

MAX\_N = 1 << 31 - 1

MIN\_READ\_SIZE = 4096

PATTERN = re.compile(r'^(?P<chunk>[^\r\n]+)|(?P<newline>\n|\r\n?)')

def \_\_init\_\_(self, fileobj, mode, zipinfo, decrypter=None,

close\_fileobj=False):

self.\_fileobj = fileobj

self.\_decrypter = decrypter

self.\_close\_fileobj = close\_fileobj

self.\_compress\_type = zipinfo.compress\_type

self.\_compress\_size = zipinfo.compress\_size

self.\_compress\_left = zipinfo.compress\_size

if self.\_compress\_type == ZIP\_DEFLATED:

self.\_decompressor = zlib.decompressobj(-15)

elif self.\_compress\_type != ZIP\_STORED:

descr = compressor\_names.get(self.\_compress\_type)

if descr:

raise NotImplementedError("compression type %d (%s)" % (self.\_compress\_type, descr))

else:

raise NotImplementedError("compression type %d" % (self.\_compress\_type,))

self.\_unconsumed = ''

self.\_readbuffer = ''

self.\_offset = 0

self.\_universal = 'U' in mode

self.newlines = None

if self.\_decrypter is not None:

self.\_compress\_left -= 12

self.mode = mode

self.name = zipinfo.filename

if hasattr(zipinfo, 'CRC'):

self.\_expected\_crc = zipinfo.CRC

self.\_running\_crc = crc32(b'') & 0xffffffff

else:

self.\_expected\_crc = None

def readline(self, limit=-1):

"""Read and return a line from the stream.

If limit is specified, at most limit bytes will be read.

"""

if not self.\_universal and limit < 0:

i = self.\_readbuffer.find('\n', self.\_offset) + 1

if i > 0:

line = self.\_readbuffer[self.\_offset: i]

self.\_offset = i

return line

if not self.\_universal:

return io.BufferedIOBase.readline(self, limit)

line = ''

while limit < 0 or len(line) < limit:

readahead = self.peek(2)

if readahead == '':

return line

match = self.PATTERN.search(readahead)

newline = match.group('newline')

if newline is not None:

if self.newlines is None:

self.newlines = []

if newline not in self.newlines:

self.newlines.append(newline)

self.\_offset += len(newline)

return line + '\n'

chunk = match.group('chunk')

if limit >= 0:

chunk = chunk[: limit - len(line)]

self.\_offset += len(chunk)

line += chunk

return line

def peek(self, n=1):

"""Returns buffered bytes without advancing the position."""

if n > len(self.\_readbuffer) - self.\_offset:

chunk = self.read(n)

if len(chunk) > self.\_offset:

self.\_readbuffer = chunk + self.\_readbuffer[self.\_offset:]

self.\_offset = 0

else:

self.\_offset -= len(chunk)

return self.\_readbuffer[self.\_offset: self.\_offset + 512]

def readable(self):

return True

def read(self, n=-1):

"""Read and return up to n bytes.

If the argument is omitted, None, or negative, data is read and returned until EOF is reached..

"""

buf = ''

if n is None:

n = -1

while True:

if n < 0:

data = self.read1(n)

elif n > len(buf):

data = self.read1(n - len(buf))

else:

return buf

if len(data) == 0:

return buf

buf += data

def \_update\_crc(self, newdata, eof):

if self.\_expected\_crc is None:

return

self.\_running\_crc = crc32(newdata, self.\_running\_crc) & 0xffffffff

if eof and self.\_running\_crc != self.\_expected\_crc:

raise BadZipfile("Bad CRC-32 for file %r" % self.name)

def read1(self, n):

"""Read up to n bytes with at most one read() system call."""

if n < 0 or n is None:

n = self.MAX\_N

len\_readbuffer = len(self.\_readbuffer) - self.\_offset

if self.\_compress\_left > 0 and n > len\_readbuffer + len(self.\_unconsumed):

nbytes = n - len\_readbuffer - len(self.\_unconsumed)

nbytes = max(nbytes, self.MIN\_READ\_SIZE)

nbytes = min(nbytes, self.\_compress\_left)

data = self.\_fileobj.read(nbytes)

self.\_compress\_left -= len(data)

if data and self.\_decrypter is not None:

data = ''.join(map(self.\_decrypter, data))

if self.\_compress\_type == ZIP\_STORED:

self.\_update\_crc(data, eof=(self.\_compress\_left==0))

self.\_readbuffer = self.\_readbuffer[self.\_offset:] + data

self.\_offset = 0

else:

self.\_unconsumed += data

if (len(self.\_unconsumed) > 0 and n > len\_readbuffer and

self.\_compress\_type == ZIP\_DEFLATED):

data = self.\_decompressor.decompress(

self.\_unconsumed,

max(n - len\_readbuffer, self.MIN\_READ\_SIZE)

)

self.\_unconsumed = self.\_decompressor.unconsumed\_tail

eof = len(self.\_unconsumed) == 0 and self.\_compress\_left == 0

if eof:

data += self.\_decompressor.flush()

self.\_update\_crc(data, eof=eof)

self.\_readbuffer = self.\_readbuffer[self.\_offset:] + data

self.\_offset = 0

data = self.\_readbuffer[self.\_offset: self.\_offset + n]

self.\_offset += len(data)

return data

def close(self):

try :

if self.\_close\_fileobj:

self.\_fileobj.close()

finally:

super(ZipExtFile, self).close()

def platform():

if xbmc.getCondVisibility('system.platform.android'): return 'android'

elif xbmc.getCondVisibility('system.platform.linux'): return 'linux'

elif xbmc.getCondVisibility('system.platform.linux.Raspberrypi'): return 'linux'

elif xbmc.getCondVisibility('system.platform.windows'): return 'windows'

elif xbmc.getCondVisibility('system.platform.osx'): return 'osx'

elif xbmc.getCondVisibility('system.platform.atv2'): return 'atv2'

elif xbmc.getCondVisibility('system.platform.ios'): return 'ios'

elif xbmc.getCondVisibility('system.platform.darwin'): return 'ios'

class ZipFile(object):

fp = None

def \_\_init\_\_(self, file, mode="r", compression=ZIP\_STORED, allowZip64=False):

if mode not in ("r", "w", "a"):

raise RuntimeError('ZipFile() requires mode "r", "w", or "a"')

if compression == ZIP\_STORED:

pass

elif compression == ZIP\_DEFLATED:

if not zlib:

raise RuntimeError,\

"Compression requires the (missing) zlib module"

else:

raise RuntimeError, "That compression method is not supported"

self.\_allowZip64 = allowZip64

self.\_didModify = False

self.debug = 0

self.NameToInfo = {}

self.filelist = []

self.compression = compression

self.mode = key = mode.replace('b', '')[0]

self.pwd = None

self.\_comment = ''

#FTG mod for Kodi 18

if platform() == 'android':

file = io.FileIO(file,mode)

if isinstance(file, basestring):

self.\_filePassed = 0

self.filename = file

modeDict = {'r' : 'rb', 'w': 'wb', 'a' : 'r+b'}

try:

self.fp = open(file, modeDict[mode])

except IOError:

if mode == 'a':

mode = key = 'w'

self.fp = open(file, modeDict[mode])

else:

raise ################

else:

self.\_filePassed = 1

self.fp = file

self.filename = getattr(file, 'name', None)

try:

if key == 'r':

self.\_RealGetContents()

elif key == 'w':

self.\_didModify = True

elif key == 'a':

try:

self.\_RealGetContents()

self.fp.seek(self.start\_dir, 0)

except BadZipfile:

self.fp.seek(0, 2)

self.\_didModify = True

else:

raise RuntimeError('Mode must be "r", "w" or "a"')

except:

fp = self.fp

self.fp = None

if not self.\_filePassed:

fp.close()

raise

def \_\_enter\_\_(self):

return self

def \_\_exit\_\_(self, type, value, traceback):

self.close()

def \_RealGetContents(self):

"""Read in the table of contents for the ZIP file."""

fp = self.fp

try:

endrec = \_EndRecData(fp)

except IOError:

raise BadZipfile("File is not a zip file")

if not endrec:

raise BadZipfile, "File is not a zip file"

if self.debug > 1:

print endrec

size\_cd = endrec[\_ECD\_SIZE]

offset\_cd = endrec[\_ECD\_OFFSET]

self.\_comment = endrec[\_ECD\_COMMENT]

concat = endrec[\_ECD\_LOCATION] - size\_cd - offset\_cd

if endrec[\_ECD\_SIGNATURE] == stringEndArchive64:

concat -= (sizeEndCentDir64 + sizeEndCentDir64Locator)

if self.debug > 2:

inferred = concat + offset\_cd

print "given, inferred, offset", offset\_cd, inferred, concat

self.start\_dir = offset\_cd + concat

fp.seek(self.start\_dir, 0)

data = fp.read(size\_cd)

fp = cStringIO.StringIO(data)

total = 0

while total < size\_cd:

centdir = fp.read(sizeCentralDir)

if len(centdir) != sizeCentralDir:

raise BadZipfile("Truncated central directory")

centdir = struct.unpack(structCentralDir, centdir)

if centdir[\_CD\_SIGNATURE] != stringCentralDir:

raise BadZipfile("Bad magic number for central directory")

if self.debug > 2:

print centdir

filename = fp.read(centdir[\_CD\_FILENAME\_LENGTH])

x = ZipInfo(filename)

x.extra = fp.read(centdir[\_CD\_EXTRA\_FIELD\_LENGTH])

x.comment = fp.read(centdir[\_CD\_COMMENT\_LENGTH])

x.header\_offset = centdir[\_CD\_LOCAL\_HEADER\_OFFSET]

(x.create\_version, x.create\_system, x.extract\_version, x.reserved,

x.flag\_bits, x.compress\_type, t, d,

x.CRC, x.compress\_size, x.file\_size) = centdir[1:12]

x.volume, x.internal\_attr, x.external\_attr = centdir[15:18]

x.\_raw\_time = t

x.date\_time = ( (d>>9)+1980, (d>>5)&0xF, d&0x1F,

t>>11, (t>>5)&0x3F, (t&0x1F) \* 2 )

x.\_decodeExtra()

x.header\_offset = x.header\_offset + concat

x.filename = x.\_decodeFilename()

self.filelist.append(x)

self.NameToInfo[x.filename] = x

total = (total + sizeCentralDir + centdir[\_CD\_FILENAME\_LENGTH]

+ centdir[\_CD\_EXTRA\_FIELD\_LENGTH]

+ centdir[\_CD\_COMMENT\_LENGTH])

if self.debug > 2:

print "total", total

def namelist(self):

"""Return a list of file names in the archive."""

l = []

for data in self.filelist:

l.append(data.filename)

return l

def infolist(self):

"""Return a list of class ZipInfo instances for files in the

archive."""

return self.filelist

def printdir(self):

"""Print a table of contents for the zip file."""

print "%-46s %19s %12s" % ("File Name", "Modified ", "Size")

for zinfo in self.filelist:

date = "%d-%02d-%02d %02d:%02d:%02d" % zinfo.date\_time[:6]

print "%-46s %s %12d" % (zinfo.filename, date, zinfo.file\_size)

def testzip(self):

"""Read all the files and check the CRC."""

chunk\_size = 2 \*\* 20

for zinfo in self.filelist:

try:

with self.open(zinfo.filename, "r") as f:

while f.read(chunk\_size):

pass

except BadZipfile:

return zinfo.filename

def getinfo(self, name):

"""Return the instance of ZipInfo given 'name'."""

info = self.NameToInfo.get(name)

if info is None:

raise KeyError(

'There is no item named %r in the archive' % name)

return info

def setpassword(self, pwd):

"""Set default password for encrypted files."""

self.pwd = pwd

@property

def comment(self):

"""The comment text associated with the ZIP file."""

return self.\_comment

@comment.setter

def comment(self, comment):

if len(comment) > ZIP\_MAX\_COMMENT:

import warnings

warnings.warn('Archive comment is too long; truncating to %d bytes'

% ZIP\_MAX\_COMMENT, stacklevel=2)

comment = comment[:ZIP\_MAX\_COMMENT]

self.\_comment = comment

self.\_didModify = True

def read(self, name, pwd=None):

"""Return file bytes (as a string) for name."""

return self.open(name, "r", pwd).read()

def open(self, name, mode="r", pwd=None):

"""Return file-like object for 'name'."""

###FTG mod for Kodi 18

if mode not in ("r", "U", "rU"):

raise RuntimeError, 'open() requires mode "r", "U", or "rU"'

if not self.fp:

raise RuntimeError, \

"Attempt to read ZIP archive that was already closed"

if self.\_filePassed:

zef\_file = self.fp

should\_close = False

else:

zef\_file = open(self.filename, 'rb')

should\_close = True

try:

if isinstance(name, ZipInfo):

zinfo = name

else:

zinfo = self.getinfo(name)

zef\_file.seek(zinfo.header\_offset, 0)

fheader = zef\_file.read(sizeFileHeader)

if len(fheader) != sizeFileHeader:

raise BadZipfile("Truncated file header")

fheader = struct.unpack(structFileHeader, fheader)

if fheader[\_FH\_SIGNATURE] != stringFileHeader:

raise BadZipfile("Bad magic number for file header")

fname = zef\_file.read(fheader[\_FH\_FILENAME\_LENGTH])

if fheader[\_FH\_EXTRA\_FIELD\_LENGTH]:

zef\_file.read(fheader[\_FH\_EXTRA\_FIELD\_LENGTH])

if fname != zinfo.orig\_filename:

raise BadZipfile, \

'File name in directory "%s" and header "%s" differ.' % (

zinfo.orig\_filename, fname)

is\_encrypted = zinfo.flag\_bits & 0x1

zd = None

if is\_encrypted:

if not pwd:

pwd = self.pwd

if not pwd:

raise RuntimeError, "File %s is encrypted, " \

"password required for extraction" % name

zd = \_ZipDecrypter(pwd)

bytes = zef\_file.read(12)

h = map(zd, bytes[0:12])

if zinfo.flag\_bits & 0x8:

check\_byte = (zinfo.\_raw\_time >> 8) & 0xff

else:

check\_byte = (zinfo.CRC >> 24) & 0xff

if ord(h[11]) != check\_byte:

raise RuntimeError("Bad password for file", name)

return ZipExtFile(zef\_file, mode, zinfo, zd,

close\_fileobj=should\_close)

################################################

except:

if should\_close:

zef\_file.close()

raise

def extract(self, member, path=None, pwd=None):

"""Extract a member from the archive to the current working directory,

using its full name. Its file information is extracted as accurately

as possible. `member' may be a filename or a ZipInfo object. You can

specify a different directory using `path'.

"""

if not isinstance(member, ZipInfo):

member = self.getinfo(member)

if path is None:

path = os.getcwd()

return self.\_extract\_member(member, path, pwd)

def extractall(self, path=None, members=None, pwd=None):

"""Extract all members from the archive to the current working

directory. `path' specifies a different directory to extract to.

`members' is optional and must be a subset of the list returned

by namelist().

"""

if members is None:

members = self.namelist()

for zipinfo in members:

self.extract(zipinfo, path, pwd)

def \_extract\_member(self, member, targetpath, pwd):

"""Extract the ZipInfo object 'member' to a physical

file on the path targetpath.

"""

arcname = member.filename.replace('/', os.path.sep)

if os.path.altsep:

arcname = arcname.replace(os.path.altsep, os.path.sep)

arcname = os.path.splitdrive(arcname)[1]

arcname = os.path.sep.join(x for x in arcname.split(os.path.sep)

if x not in ('', os.path.curdir, os.path.pardir))

if os.path.sep == '\\':

illegal = ':<>|"?\*'

if isinstance(arcname, unicode):

table = {ord(c): ord('\_') for c in illegal}

else:

table = string.maketrans(illegal, '\_' \* len(illegal))

arcname = arcname.translate(table)

arcname = (x.rstrip('.') for x in arcname.split(os.path.sep))

arcname = os.path.sep.join(x for x in arcname if x)

targetpath = os.path.join(targetpath, arcname)

targetpath = os.path.normpath(targetpath)

upperdirs = os.path.dirname(targetpath)

if upperdirs and not os.path.exists(upperdirs):

os.makedirs(upperdirs)

if member.filename[-1] == '/':

if not os.path.isdir(targetpath):

os.mkdir(targetpath)

return targetpath

with self.open(member, pwd=pwd) as source, \

file(targetpath, "wb") as target:

shutil.copyfileobj(source, target)

return targetpath

def \_writecheck(self, zinfo):

"""Check for errors before writing a file to the archive."""

if zinfo.filename in self.NameToInfo:

import warnings

warnings.warn('Duplicate name: %r' % zinfo.filename, stacklevel=3)

if self.mode not in ("w", "a"):

raise RuntimeError, 'write() requires mode "w" or "a"'

if not self.fp:

raise RuntimeError, \

"Attempt to write ZIP archive that was already closed"

if zinfo.compress\_type == ZIP\_DEFLATED and not zlib:

raise RuntimeError, \

"Compression requires the (missing) zlib module"

if zinfo.compress\_type not in (ZIP\_STORED, ZIP\_DEFLATED):

raise RuntimeError, \

"That compression method is not supported"

if not self.\_allowZip64:

requires\_zip64 = None

if len(self.filelist) >= ZIP\_FILECOUNT\_LIMIT:

requires\_zip64 = "Files count"

elif zinfo.file\_size > ZIP64\_LIMIT:

requires\_zip64 = "Filesize"

elif zinfo.header\_offset > ZIP64\_LIMIT:

requires\_zip64 = "Zipfile size"

if requires\_zip64:

raise LargeZipFile(requires\_zip64 +

" would require ZIP64 extensions")

def write(self, filename, arcname=None, compress\_type=None):

"""Put the bytes from filename into the archive under the name

arcname."""

if not self.fp:

raise RuntimeError(

"Attempt to write to ZIP archive that was already closed")

st = os.stat(filename)

isdir = stat.S\_ISDIR(st.st\_mode)

mtime = time.localtime(st.st\_mtime)

date\_time = mtime[0:6]

if arcname is None:

arcname = filename

arcname = os.path.normpath(os.path.splitdrive(arcname)[1])

while arcname[0] in (os.sep, os.altsep):

arcname = arcname[1:]

if isdir:

arcname += '/'

zinfo = ZipInfo(arcname, date\_time)

zinfo.external\_attr = (st[0] & 0xFFFF) << 16L

if isdir:

zinfo.compress\_type = ZIP\_STORED

elif compress\_type is None:

zinfo.compress\_type = self.compression

else:

zinfo.compress\_type = compress\_type

zinfo.file\_size = st.st\_size

zinfo.flag\_bits = 0x00

zinfo.header\_offset = self.fp.tell()

self.\_writecheck(zinfo)

self.\_didModify = True

if isdir:

zinfo.file\_size = 0

zinfo.compress\_size = 0

zinfo.CRC = 0

zinfo.external\_attr |= 0x10

self.filelist.append(zinfo)

self.NameToInfo[zinfo.filename] = zinfo

self.fp.write(zinfo.FileHeader(False))

return

with open(filename, "rb") as fp:

# Must overwrite original zip for Kodi 18

#FTG mod for Kodi 18

zinfo.CRC = CRC = 0

zinfo.compress\_size = compress\_size = 0

zip64 = self.\_allowZip64 and \

zinfo.file\_size \* 1.05 > ZIP64\_LIMIT

self.fp.write(zinfo.FileHeader(zip64))

if zinfo.compress\_type == ZIP\_DEFLATED:

cmpr = zlib.compressobj(zlib.Z\_DEFAULT\_COMPRESSION,

zlib.DEFLATED, -15)

else:

cmpr = None

file\_size = 0

while 1:

buf = fp.read(1024 \* 8)

if not buf:

break

file\_size = file\_size + len(buf)

CRC = crc32(buf, CRC) & 0xffffffff

if cmpr:

buf = cmpr.compress(buf)

compress\_size = compress\_size + len(buf)

self.fp.write(buf)

if cmpr:

buf = cmpr.flush()

compress\_size = compress\_size + len(buf)

self.fp.write(buf)

zinfo.compress\_size = compress\_size

else:

zinfo.compress\_size = file\_size

zinfo.CRC = CRC

zinfo.file\_size = file\_size

if not zip64 and self.\_allowZip64:

if file\_size > ZIP64\_LIMIT:

raise RuntimeError('File size has increased during compressing')

if compress\_size > ZIP64\_LIMIT:

raise RuntimeError('Compressed size larger than uncompressed size')

#FTG mod for Kodi 18 Android

position = self.fp.tell()

self.fp.seek(zinfo.header\_offset, 0)

self.fp.write(zinfo.FileHeader(zip64))

self.fp.seek(position, 0)

self.filelist.append(zinfo)

self.NameToInfo[zinfo.filename] = zinfo

#######################################

def writestr(self, zinfo\_or\_arcname, bytes, compress\_type=None):

"""Write a file into the archive. The contents is the string

'bytes'. 'zinfo\_or\_arcname' is either a ZipInfo instance or

the name of the file in the archive."""

if not isinstance(zinfo\_or\_arcname, ZipInfo):

zinfo = ZipInfo(filename=zinfo\_or\_arcname,

date\_time=time.localtime(time.time())[:6])

zinfo.compress\_type = self.compression

if zinfo.filename[-1] == '/':

zinfo.external\_attr = 0o40775 << 16

zinfo.external\_attr |= 0x10

else:

zinfo.external\_attr = 0o600 << 16

else:

zinfo = zinfo\_or\_arcname

if not self.fp:

raise RuntimeError(

"Attempt to write to ZIP archive that was already closed")

if compress\_type is not None:

zinfo.compress\_type = compress\_type

zinfo.file\_size = len(bytes)

zinfo.header\_offset = self.fp.tell()

self.\_writecheck(zinfo)

self.\_didModify = True

zinfo.CRC = crc32(bytes) & 0xffffffff

if zinfo.compress\_type == ZIP\_DEFLATED:

co = zlib.compressobj(zlib.Z\_DEFAULT\_COMPRESSION,

zlib.DEFLATED, -15)

bytes = co.compress(bytes) + co.flush()

zinfo.compress\_size = len(bytes)

else:

zinfo.compress\_size = zinfo.file\_size

zip64 = zinfo.file\_size > ZIP64\_LIMIT or \

zinfo.compress\_size > ZIP64\_LIMIT

if zip64 and not self.\_allowZip64:

raise LargeZipFile("Filesize would require ZIP64 extensions")

self.fp.write(zinfo.FileHeader(zip64))

self.fp.write(bytes)

#FTG Mod for kodi 18

if zinfo.flag\_bits & 0x08:

fmt = '<LQQ' if zip64 else '<LLL'

self.fp.write(struct.pack(fmt, zinfo.CRC, zinfo.compress\_size,

zinfo.file\_size))

######################################

self.fp.flush()

self.filelist.append(zinfo)

self.NameToInfo[zinfo.filename] = zinfo

def \_\_del\_\_(self):

"""Call the "close()" method in case the user forgot."""

self.close()

def close(self):

"""Close the file, and for mode "w" and "a" write the ending

records."""

if self.fp is None:

return

try:

if self.mode in ("w", "a") and self.\_didModify:

pos1 = self.fp.tell()

for zinfo in self.filelist:

dt = zinfo.date\_time

dosdate = (dt[0] - 1980) << 9 | dt[1] << 5 | dt[2]

dostime = dt[3] << 11 | dt[4] << 5 | (dt[5] // 2)

extra = []

if zinfo.file\_size > ZIP64\_LIMIT \

or zinfo.compress\_size > ZIP64\_LIMIT:

extra.append(zinfo.file\_size)

extra.append(zinfo.compress\_size)

file\_size = 0xffffffff

compress\_size = 0xffffffff

else:

file\_size = zinfo.file\_size

compress\_size = zinfo.compress\_size

if zinfo.header\_offset > ZIP64\_LIMIT:

extra.append(zinfo.header\_offset)

header\_offset = 0xffffffffL

else:

header\_offset = zinfo.header\_offset

extra\_data = zinfo.extra

if extra:

#FTG mod for Kodi 18

extra\_data = struct.pack(

'<HH' + 'Q'\*len(extra),

1, 8\*len(extra), \*extra) + extra\_data

#############################

extract\_version = max(45, zinfo.extract\_version)

create\_version = max(45, zinfo.create\_version)

else:

extract\_version = zinfo.extract\_version

create\_version = zinfo.create\_version

try:

filename, flag\_bits = zinfo.\_encodeFilenameFlags()

centdir = struct.pack(structCentralDir,

stringCentralDir, create\_version,

zinfo.create\_system, extract\_version, zinfo.reserved,

flag\_bits, zinfo.compress\_type, dostime, dosdate,

zinfo.CRC, compress\_size, file\_size,

len(filename), len(extra\_data), len(zinfo.comment),

0, zinfo.internal\_attr, zinfo.external\_attr,

header\_offset)

except DeprecationWarning:

print >>sys.stderr, (structCentralDir,

stringCentralDir, create\_version,

zinfo.create\_system, extract\_version, zinfo.reserved,

zinfo.flag\_bits, zinfo.compress\_type, dostime, dosdate,

zinfo.CRC, compress\_size, file\_size,

len(zinfo.filename), len(extra\_data), len(zinfo.comment),

0, zinfo.internal\_attr, zinfo.external\_attr,

header\_offset)

raise

self.fp.write(centdir)

self.fp.write(filename)

self.fp.write(extra\_data)

self.fp.write(zinfo.comment)

#FTG mod for Kodi 18

pos2 = self.fp.tell()

centDirCount = len(self.filelist)

centDirSize = pos2 - pos1

centDirOffset = pos1

requires\_zip64 = None

if centDirCount > ZIP\_FILECOUNT\_LIMIT:

requires\_zip64 = "Files count"

elif centDirOffset > ZIP64\_LIMIT:

requires\_zip64 = "Central directory offset"

elif centDirSize > ZIP64\_LIMIT:

requires\_zip64 = "Central directory size"

if requires\_zip64:

##############

if not self.\_allowZip64:

raise LargeZipFile(requires\_zip64 +

" would require ZIP64 extensions")

zip64endrec = struct.pack(

structEndArchive64, stringEndArchive64,

44, 45, 45, 0, 0, centDirCount, centDirCount,

centDirSize, centDirOffset)

self.fp.write(zip64endrec)

zip64locrec = struct.pack(

structEndArchive64Locator,

stringEndArchive64Locator, 0, pos2, 1)

self.fp.write(zip64locrec)

centDirCount = min(centDirCount, 0xFFFF)

centDirSize = min(centDirSize, 0xFFFFFFFF)

centDirOffset = min(centDirOffset, 0xFFFFFFFF)

endrec = struct.pack(structEndArchive, stringEndArchive,

0, 0, centDirCount, centDirCount,

centDirSize, centDirOffset, len(self.\_comment))

self.fp.write(endrec)

self.fp.write(self.\_comment)

self.fp.flush()

finally:

fp = self.fp

self.fp = None

if not self.\_filePassed:

fp.close()

class PyZipFile(ZipFile):

"""Class to create ZIP archives with Python library files and packages."""

def writepy(self, pathname, basename = ""):

"""Add all files from "pathname" to the ZIP archive.

If pathname is a package directory, search the directory and

all package subdirectories recursively for all \*.py and enter

the modules into the archive. If pathname is a plain

directory, listdir \*.py and enter all modules. Else, pathname

must be a Python \*.py file and the module will be put into the

archive. Added modules are always module.pyo or module.pyc.

This method will compile the module.py into module.pyc if

necessary.

"""

dir, name = os.path.split(pathname)

if os.path.isdir(pathname):

initname = os.path.join(pathname, "\_\_init\_\_.py")

if os.path.isfile(initname):

# This is a package directory, add it

if basename:

basename = "%s/%s" % (basename, name)

else:

basename = name

if self.debug:

print "Adding package in", pathname, "as", basename

fname, arcname = self.\_get\_codename(initname[0:-3], basename)

if self.debug:

print "Adding", arcname

self.write(fname, arcname)

dirlist = os.listdir(pathname)

dirlist.remove("\_\_init\_\_.py")

for filename in dirlist:

path = os.path.join(pathname, filename)

root, ext = os.path.splitext(filename)

if os.path.isdir(path):

if os.path.isfile(os.path.join(path, "\_\_init\_\_.py")):

self.writepy(path, basename) # Recursive call

elif ext == ".py":

fname, arcname = self.\_get\_codename(path[0:-3],

basename)

if self.debug:

print "Adding", arcname

self.write(fname, arcname)

else:

if self.debug:

print "Adding files from directory", pathname

for filename in os.listdir(pathname):

path = os.path.join(pathname, filename)

root, ext = os.path.splitext(filename)

if ext == ".py":

fname, arcname = self.\_get\_codename(path[0:-3],

basename)

if self.debug:

print "Adding", arcname

self.write(fname, arcname)

else:

if pathname[-3:] != ".py":

raise RuntimeError, \

'Files added with writepy() must end with ".py"'

fname, arcname = self.\_get\_codename(pathname[0:-3], basename)

if self.debug:

print "Adding file", arcname

self.write(fname, arcname)

def \_get\_codename(self, pathname, basename):

"""Return (filename, archivename) for the path.

Given a module name path, return the correct file path and

archive name, compiling if necessary. For example, given

/python/lib/string, return (/python/lib/string.pyc, string).

"""

file\_py = pathname + ".py"

file\_pyc = pathname + ".pyc"

file\_pyo = pathname + ".pyo"

if os.path.isfile(file\_pyo) and \

os.stat(file\_pyo).st\_mtime >= os.stat(file\_py).st\_mtime:

fname = file\_pyo #FTG mod for Kodi 18

elif not os.path.isfile(file\_pyc) or \

os.stat(file\_pyc).st\_mtime < os.stat(file\_py).st\_mtime:

import py\_compile

if self.debug:

print "Compiling", file\_py

try:

py\_compile.compile(file\_py, file\_pyc, None, True)

except py\_compile.PyCompileError,err:

print err.msg

fname = file\_pyc

else:

fname = file\_pyc

archivename = os.path.split(fname)[1]

if basename:

archivename = "%s/%s" % (basename, archivename)

return (fname, archivename)