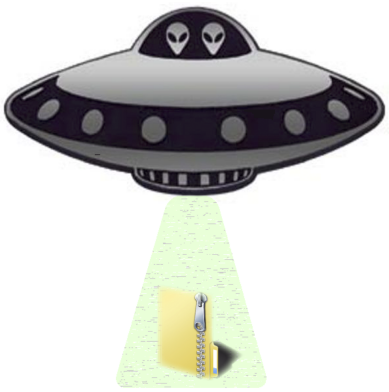
xphyle Documentation

Release 4.2.0+4.g1b6dd34.dirty

John P Didion

Dec 07, 2020

Contents



xphyle is a small python (3.4+) library that makes it easy to open compressed files and URLs for the highest possible performance available on your system.

- API
- Source code
- Report an issue

CHAPTER 1

Installation

xphyle is available from pypi:

```
pip install xphyle
```

xphyle tries to use the compression programs installed on your local machine (e.g. gzip, bzip2); if it can't, it will use the built-in python libraries (which are slower). Thus, xphyle has no required dependencies, but we recommend that if you install gzip, etc. if you don't already have them.

xphyle will use pigz for multi-threaded gzip compression if it is available. Multithreading support is disabled by default; to set the number of threads that xphyle should use:

```
xphyle.configure(threads=4)
```

or, to automatically set it to the number of cores available on your system:

```
xphyle.configure(threads=True)
```

If you have programs installed at a location that is not on your path, you can add those locations to xphyle's executable search:

```
xphyle.configure(executable_path=['/path', '/another/path', ...])
```

If you would like progress bars displayed for file operations, you need to configure one or both of the python-level and system-level progress bars.

For python-level operations, the pokrok API is used by default. Pokrok provides access to many popular progress bar libraries with a single, standard interface. Please see the documentation for more information about which libraries are currently supported and how to configure them. To enable this:

```
> pip install pokrok

xphyle.configure(progress=True)
```

You can also use you own preferred progress bar by passing a callable, which must take a single iterable argument and two optional keyword arguments and return an iterable:

```
def my_progress_wrapper(itr, desc='My progress bar', size=None):
    ...
```

(continues on next page)

(continued from previous page)

```
xphyle.configure(progress=my_progress_wrapper)
```

For system-level operations, an executable is required that reads from stdin and writes to stdout; pv is used by default. To enable this:

```
xphyle.configure(system_progress=True)
```

You can also use your own preferred program by passing a tuple with the command and arguments (<xphyle.progress.system_progress_command>() simplifies this):

```
xphyle.configure(system_progress=xphyle.progress.system_progress_command(
    'pv', '-pre', require=True))
```

CHAPTER 2

Working with files

The heart of xphyle is the simplicity of working with files. There is a single interface – `xopen` – for opening "file-like objects", regardless of whether they represent local files, remote files (referenced by URLs), or system streams (stdin, stdout, stderr); and regardless of whether they are compressed.

The following are functionally equivalent ways to open a gzip file:

```
import gzip
f = gzip.open('input.gz', 'rt')

from xphyle import xopen
f = xopen('input.gz', 'rt')
```

So then why use xphyle? Two reasons:

1. The `gzip.open` method of opening a gzip file above requires you to know that you are expecting a gzip file and only a gzip file. If your program optionally accepts either a compressed or a decompressed file, then you'll need several extra lines of code to either detect the file format or to make the user specify the format of the file they are providing. This becomes increasingly cumbersome with each additional format you want to support. On the other hand, `xopen` has the same interface regardless of the compression format. Furthermore, if xphyle doesn't currently support a file format that you would like to use, it enables you to add it via a simple API.
2. The `gzip.open` method of opening a gzip file uses python code to decompress the file. It's well written, highly optimized python code, but unfortunately it's still slower than your natively compiled system-level applications (e.g. pigz or gzip). The `xopen` method of opening a gzip file first tries to use pigz or gzip to decompress the file and provides access to the resulting stream of decompressed data (as a file-like object), and only falls back to `gzip.open` if neither program is available.

If you want to be explicit about whether to expect a compressed file, what type of compression to expect, or whether to try and use system programs, you can:

```
from xphyle import xopen
from xphyle.paths import STDIN

# Expect the file to not be compressed
f = xopen('input', 'rb', compression=False)

# Open a remote file. Expect the file to be compressed, and throw an error
# if it's not, or if the compression format cannot be determined.
```

(continues on next page)

(continued from previous page)

```
f = xopen('http://foo.com/input.gz', 'rt', compression=True)

# Open stdin. Expect the input to be gzip compressed, and throw an error if
# it's not
f = xopen(STDIN, 'rt', compression='gzip')

# Do not try to use the system-level gzip program for decompression
f = xopen('input.gz', 'rt', compression='gzip', use_system=False)
```

By default, `xopen` returns the file. If desired, `xopen` can also wrap the file such that it behaves just like a file with a few additional features:

- A file iterator is wrapped in a progress bar (if they have been enabled via the `configure` method described above).
- A simple event system that enables callbacks to be registered for various events. Currently, the only supported event is closing the file. The `xphyle.utils` package provides a few useful event listeners, e.g. to compress, move, or delete the file when it is closed.
- ContextManager functionality, such that the file is always compatible with `with`, e.g.:

```
def print_lines(path):
    # this works whether path refers to a local file, URL or STDIN
    with xopen(path, context_wrapper=True) as infile:
        for line in infile:
            print(line)
```

The wrapping behavior can be enabled by passing `context_wrapper=True` to `xopen`. You can configure `xopen` to wrap files by default:

```
xphyle.configure(default_xopen_context_wrapper=True)
```

Note that this represents a change from xphyle 1.x, in which wrapping occurred by default.

Another common pattern is to write functions that accept either a path or an open file object. Rather than having to test whether the user passed a path or a file and handle each differently, you can use the `open_` convenience method:

```
from xphyle import open_

def print_lines(path_or_file):
    with open_(path_or_file) as infile:
        for line in infile:
            print(line)
```

Note that `open_` wraps files by default, including already open file-like objects. To disable this, set `wrap_fileobj=False`.

2.1 Supported file formats

Currently, xphyle supports the most commonly used file formats: gzip, bzip2/7zip, and lzma/xz.

Also supported is block-based gzip (bgzip), a format commonly used in bioinformatics. Somewhat confusingly, '.gz' is an acceptable extension for bgzip files, and gzip will decompress bgzip files. Thus, to specifically use bgzip, either use a '.bgz' file extension or specify 'bgzip' as the compression format:

```
f = xopen('input.gz', 'rt', compression='bgzip', validate=False)
```

Additional compression formats may be added in the future. To get the most up-to-date list:

```
from xphyle.formats import FORMATS
print(', '.join(FORMATS.list_compression_formats())
```

2.2 Processes

As of xphyle 2.0.0, you can easily open subprocesses using the `xphyle.popen` method. This method is similar to python `subprocess.Popen`, except that it uses `xopen` to open files passed to stdin, stdout, and stderr, and/or to wrap subprocess PIPEs. `xphyle.popen` returns an `xphyle.Process` object, which is a subclass of `subprocess.Popen` but adds additional functionality, essentially making a Process behave like a regular file. Writing to a process writes to its stdin PIPE, and reading from a process reads from its stdout or stderr PIPE:

```
from xphyle import popen, PIPE
proc = popen('cat', stdin=PIPE, stdout='myfile.gz')
try:
    proc.write('foo')
finally:
    proc.close()

# equivalent to:
with popen('cat', stdin=PIPE, stdout='myfile.gz'):
    proc.write('foo')

# and also to:
popen('cat', stdin=PIPE, stdout='myfile.gz').communicate('foo')

# for the common case above, there's also a shortcut method
from xphyle.utils import exec_process
exec_process('cat', 'foo', stdout='myfile.gz')
```

In addition, `open_` and `xopen` can open subprocesses. The primary difference is that `popen` enables customization of stdin, stdout, and stderr, whereas opening a process through `open_` or `xopen` uses default behavior of opening PIPEs for all of the streams, and wrapping the PIPE indicated by the file mode. For example:

```
# write to the process stdin
with open_('|cat', 'wt') as proc:
    proc.write('foo')

# this command wraps stdin with gzip compression
with open_('|zcat', 'wt', compression='gzip') as proc:
    proc.write('foo')

# this command wraps stdout with gzip decompression;
# furthermore, the compression format is determined
# automatically
with open_('|gzip -c foobar.txt', 'rt') as proc:
    text = proc.read()
```

Note that with `open_` and `xopen`, the system command must be specified as a string starting with '|'.

2.3 Buffers

As of xphyle 2.1.0, `open_` and `xopen` can also open buffer types. A buffer is an instance of `io.StringIO` or `io.BytesIO` (or similar) – basically an in memory read/write buffer. Passing open buffer objects worked before (they were treated as file-like), but now there is a special file type – `FileType.BUFFER` – that will cause them to be handled a bit differently. In addition, you can now pass `str` or `bytes` (the type objects) to automatically create the corresponding buffer type:

```
with open_(str) as buf:
    buf.write('foo')
string_foo = buf.getvalue()

# with compression, type must be 'bytes'
with open_(bytes, compression='gzip') as buf:
    buf.write('foo')
compressed_foo = buf.getvalue()
```

You can also create readable buffers by passing the string/bytes to read instead of a path, and explicitly specifying the file type:

```
with open_("This is a string I want to read", file_type=FileType.BUFFER) as buf:
    buf_str = buf.read()
```

2.4 Reading/writing data

The `xphyle.utils` module provides methods for many of the common operations that you'll want to perform on files. A few examples are shown below; you can read the API docs for a full list of methods and more detailed descriptions of each.

There are pairs of methods for reading/writing text and binary data using iterators:

```
# Copy from one file to another, changing the line separator from
# unix to windows
from xphyle.utils import read_lines, write_lines
write_lines(
    read_lines('linux_file.txt')
    'windows_file.txt',
    linesep='\r')

# Copy from one binary file to another, changing the encoding from
# ascii to utf-8
from xphyle.utils import read_bytes, write_bytes
def ascii2utf8(x):
    if isinstance(x, bytes):
        x = x.decode('ascii')
    return x.encode('utf-8')
write_bytes(
    read_bytes('ascii_file.txt', convert=ascii2utf8),
    'utf8-file.txt')
```

There's another pair of methods for reading/writing key=value files:

```
from collections import OrderedDict
from xphyle.utils import read_dict, write_dict
cats = OrderedDict((fluffy,'calico'), (droopy,'tabby'), (sneezy,'siamese'))
write_dict(cats, 'cats.txt.gz')
# change from '=' to '\t' delimited; preserve the order of the items
write_dict(
    read_dict(cats, 'cats.txt.gz', ordered=True),
    'cats.tsv', sep='\t')
```

You can also read from delimited files such as csv and tsv:

```
from xphyle.utils import read_delimited, read_delimited_as_dict

class Dog(object):
    def __init__(self, name, age, breed):
```

(continues on next page)

(continued from previous page)

```
        self.name = name
        self.age = age
        self.breed = breed
    def pet(self): ...
    def say(self, message): ...

for dog in read_delimited(
        'dogs.txt.gz', header=True,
        converters=(str,int,str),
        row_type=Dog):
    dog.pet()

dogs = read_delimited_as_dict(
        'dogs.txt.gz', header=True,
        key='name', converters=(str,int,str),
        row_type=Dog):
dogs['Barney'].say('Good Boy!')
```

There are convenience methods for compressing and decompressing files:

```
from xphyle.utils import compress_file, decompress_file, transcode_file

# Gzip compress recipes.txt, and delete the original
compress_file('recipes.txt', compression='gzip', keep=False)

# decompress a remote compressed file to a local file
decompress_file('http://recipes.com/allrecipes.txt.gz',
                'local_recipes.txt')

# Change from gzip to bz2 compression:
transcode_file('http://recipes.com/allrecipes.txt.gz',
               'local_recipes.txt.bz2')
```

There is a replacement for `fileinput`:

```
from xphyle.utils import fileinput

# By default, read from the files specified as command line arguments,
# or stdin if there are no command line arguments, and autodetect
# the compression format
for line in fileinput():
    print(line)

# Read from multiple files as if they were one
for line in fileinput(('myfile.txt', 'myotherfile.txt.gz')):
    print(line)
```

There's also a set of classes for writing to multiple files:

```
from xphyle.utils import fileoutput
from xphyle.utils import TeeFileOutput, CycleFileOutput, NCycleFileOutput

# write all lines in sourcefile.txt to both file1 and file2.gz
with fileoutput(
        ('file1', 'file2.gz'),
        file_output_type=TeeFileOutput) as out:
    out.writelines(read_lines('sourcefile.txt'))

# Alternate writing each line in sourcefile.txt to file1 and file2.gz
with fileoutput(
        ('file1', 'file2.gz'),
```

(continues on next page)

(continued from previous page)

```
        file_output_type=CycleFileOutput) as out:
    out.writelines(read_lines('sourcefile.txt'))

# Alternate writing four lines in sourcefile.txt to file1 and file2.gz
with fileoutput(
        ('file1', 'file2.gz'),
        file_output_type=NCycleFileOutput, n=4) as out:
    out.writelines(read_lines('sourcefile.txt'))

# Write up to 10,000 lines in each file before opening the next file
with RollingFileOutput('file{}.gz', n=10000) as out:
    out.writelines(read_lines('sourcefile.txt'))
```

And finally, there's some miscellaneous methods such as linecount:

```
from xphyle.utils import linecount
print("There are {} lines in file {}".format(
    linecount(path), path))
```

2.5 File paths

The `xphyle.paths` module provides methods for working with file paths. The API docs have a full list of methods and more detailed descriptions of each. Here are a few examples:

```
from xphyle.paths import *

# Get the absolute path, being smart about STDIN/STDOUT/STDERR and
# home directory shortcuts
abspath('/foo/bar/baz') # -> /foo/bar/baz
abspath('foo') # -> /path/to/current/dir/foo
abspath('~/foo') # -> /home/myname/foo
abspath(STDIN) # -> STDIN

# Splat a path into its component parts
dir, name, *extensions = split_path('/home/joe/foo.txt.gz') # ->
    # dir = '/home/joe'
    # name = 'foo'
    # extensions = ['txt', 'gz']

# Check that a path exists, is a file, and allows reading
# Raises IOError if any of the expectations are violated,
# otherwise returns the fully resolved path
path = check_path('file.txt.gz', 'f', 'r')

# Shortcuts to check whether a file is readable/writeable
path = check_readable_file('file.txt')
path = check_writeable_file('file.txt')

# There are also 'safe' versions of the methods that return
# None rather than raise IOError
path = safe_check_readable_file('nonexistant_file.txt') # path = None

# Find all files in a directory (recursively) that match a
# regular expression pattern
find('mydir', 'file.*\.txt\.gz')

# Lookup the path to an executable
gzip_path = get_executable_path('gzip')
```

TempDir is a particularly useful class, especially for unit testing. In fact, it us used extensively for unit testing xphyle itself. TempDir can be thought of as a virtual file system. It creates a temporary directory, and it provides methods to create subdirectories and files within that directory. When the `close()` method is called, the entire temporary directory is deleted. `TempDir` can also be used as a ContextManager:

```
with TempDir() as temp:
    # create three randomly named files under 'tempdir'
    paths = temp.make_empty_files(3)
    # create directory 'tempdir/foo'
    foo = temp.make_directory('foo')
    # create a randomly named file with the '.gz' suffix
    # within directory 'tempdir/foo'
    gzfile = temp[foo].make_file(suffix='.gz')
```

Another useful set of classes is FileSpec, DirSpec, and PathSpec. These classes help with the common problem of working files that match a specific pattern, especially when you need to then extract some pieces of information from the file names. For example, you may need to find all the files starting with 'foo' within any subdirectory of '/bar', and then performing different operations depending on the extension. You could use a PathSpec for this:

```
spec = PathSpec(
    DirSpec(PathVar('subdir'), template=os.path.join('/bar', '{subdir}')),
    FileSpec(
        PathVar('name', pattern='foo.*'),
        PathVar('ext'),
        template='{name}.{ext}'))
files = spec.find(recursive=True)
for f in files:
    if f['ext'] == 'txt':
        process_text_file(f)
    else:
        process_binary_file(f)
```

A FileSpec or DirSpec has two related fields: a template, which is a python fstring and is used for constructing filenames from component pieces; and a pattern, which is a regular expression and is used for matching to path strings. The named components of the template correspond to path variables (instances of the PathVar class). Each PathVar can provide its own pattern, as well as lists of valid or invalid values. If a pattern is not specified during FileSpec/DirSpec creation, the pattern is automatically created by simply substituting the PathVar patterns for the corresponding components in the template string ('.*' by default).

Note that a DirSpec is only able to construct/match directory paths, and a FileSpec is only able to construct/match file names. A PathSpec is simply a composite type of a DirSpec and a FileSpec that can be used to construct/match full paths.

Each of the *Spec classes has three methods:

- construct: Given values for all of the path vars, construct a new path. Note that __call__ is an alias for construct.
- parse: Match a path against the *Spec's pattern. If the path matches, the component's are extracted (through the use of named capture groups), otherwise an exception is raised.
- find: Find all directories/files/paths that match the *Spec's pattern.

All of these methods return a PathInst, which is a subclass of pathlib.Path (specifically, a subclass of pathlib.WindowsPath when code is run on Windows, otherwise a PosixPath) that has an additional slot, 'values', that is a dictionary of the component name, value pairs, and overrides a few methods.

CHAPTER 3

Extending xphyle

You can add support for another compression format by extending one of the base classes in <xphyle.format>:

```
import xphyle.formats

class FooFormat(xphyle.formats.SingleExeCompressionFormat):
    """Implementation of CompressionFormat for foo files.
    """
    @property
    def name(self) -> str:
        return 'foo'

    @property
    def exts(self) -> Tuple[str, ...]:
        return ('foo',)

    @property
    def system_commands(self) -> Tuple[str, ...]:
        return ('foo',)

    @property
    def compresslevel_range(self) -> Tuple[int, int]:
        return (1, 11)

    @property
    def default_compresslevel(self) -> int:
        return 6

    @property
    def magic_bytes(self) -> Tuple[Tuple[int, ...], ...]:
        return ((0x0F, 0x00),)

    @property
    def mime_types(self) -> Tuple[str, ...]:
        return ('application/foo',)

    # build the system command
    # op = 'c' for compress, 'd' for decompress
    # src = the source file, or STDIN if input should be read from stdin
```

(continues on next page)

(continued from previous page)

```
    # stdout = True if output should be written to stdout
    # compresslevel = the compression level
    def get_command(self, op, src=STDIN, stdout=True, compresslevel=6):
        cmd = [self.executable_path]
        if op == 'c':
            # adjust the compresslevel to be within the range allowed
            # by the program
            compresslevel = self._get_compresslevel(compresslevel)
            cmd.append('-{}'.format(compresslevel))
            cmd.append('-z')
        elif op == 'd':
            cmd.append('-d')
        if stdout:
            cmd.append('-c')
        if src != STDIN:
            cmd.append(src)
        return cmd

    def open_file_python(self, filename, mode, **kwargs):
        # self.lib is a property that lazily imports and returns the
        # python library named in the ``name`` member above
        return self.lib.open_foo(filename, mode, **kwargs)
```

Then, register your format:

```
xphyle.formats.register_compression_format(FooFormat)
```

Also, note that you can support custom URL schemes by the standard method of adding urllib handlers:

```
import urllib.request
urllib.request.OpenerDirector.add_handler(my_handler)
```