pikepdf Documentation

Release 2.11.1

James R. Barlow

Apr 13, 2021

INTRODUCTION

pikepdf is a Python library allowing creation, manipulation and repair of PDFs. It provides a Pythonic wrapper around the C++ PDF content transformation library, QPDF.

Python + QPDF = "py" + "qpdf" = "pyqpdf", which looks like a dyslexia test and is no fun to type. But say "pyqpdf" out loud, and it sounds like "pikepdf".

Fig. 1: A northern pike, or *esox lucius*.

CHAPTER

ONE

AT A GLANCE

pikepdf is a library intended for developers who want to create, manipulate, parse, repair, and abuse the PDF format. It supports reading and write PDFs, including creating from scratch. Thanks to QPDF, it supports linearizing PDFs and access to encrypted PDFs.

```
# Rotate all pages in a file by 180 degrees
import pikepdf
my_pdf = pikepdf.Pdf.open('test.pdf')
for page in my_pdf.pages:
   page.Rotate = 180
my_pdf.save('test-rotated.pdf')
```

It is a low level library that requires knowledge of PDF internals and some familiarity with the PDF specification. It does not provide a user interface of its own.

pikepdf would help you build apps that do things like:

- *Copy pages* from one PDF into another
- *Split* and *merge* PDFs
- Extract content from a PDF such as text or *images*
- Replace content, such as *replacing an image* without altering the rest of the file
- Repair, reformat or `linearize` PDFs
- Change the size of pages and reposition content
- Optimize PDFs similar to Acrobat's features by downsampling images, deduplicating
- Calculate how much to charge for a scanning project based on the materials scanned
- Alter a PDF to meet a target specification such as PDF/A or PDF/X
- Add or modify PDF *metadata*
- Create well-formed but invalid PDFs for testing purposes

What it cannot do:

- Rasterize PDF pages for display (that is, produce an image that shows what a PDF page looks like at a particular resolution/zoom level) – use Ghostscript instead
- Convert from PDF to other similar paper capture formats like epub, XPS, DjVu, Postscript – use MuPDF or PyMuPDF
- Print to paper

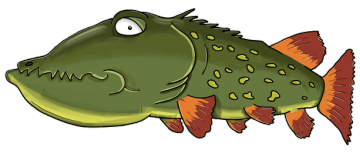


Fig. 1: Pike fish are tough, hard-fighting, aggressive predators.

If you only want to generate PDFs and not read or modify them, consider reportlab (a "write-only" PDF generator).

1.1 Requirements

pikepdf currently requires **Python 3.6+** or newer. pikepdf 1.x supports Python 3.5. Python 2.7 has never been supported.

1.2 Similar libraries

Unlike similar Python libraries such as PyPDF2 and pdfrw, pikepdf is not pure Python. These libraries were designed prior to Python wheels which has made Python extension libraries much easier to work with. By leveraging the existing mature code base of QPDF, despite being new, pikepdf is already more capable than both in many respects – for example, it can read compress object streams, repair damaged PDFs in many cases, and linearize PDFs. Unlike those libraries, it's not pure Python: it is impure and proud of it.

PyMuPDF is a PDF library with impressive capabilities. However, its AGPL license its much more restrictive than pikepdf, and its dependency on static libraries makes it difficult to include in open source Linux or BSD distributions.

1.3 In use

pikepdf is used by the same author's OCRmyPDF to inspect input PDFs, graft the generated OCR layers on to page content, and output PDFs. Its code contains several practical examples, particular in `pdfinfo.py`, `graft.py`, and `optimize.py`. pikepdf is also used in its test suite.

1.3.1 Installation

Basic installation

Fig. 3: A pike installation failure.

Most users on Linux, macOS or Windows with x64 systems should use `pip` to install pikepdf in their current Python environment (such as your project's virtual environment).

```
pip install pikepdf
```

Use `pip install --user pikepdf` to install the package for the current user only. Use `pip install pikepdf` to install to a virtual environment.

Linux users: If you have an older version of `pip`, such as the one that ships with Ubuntu 18.04, this command will attempt to compile the project instead of installing the wheel. If you want to get the binary wheel, upgrade `pip` with:

```
wget https://bootstrap.pypa.io/get-pip.py && python3 get-pip.py
pip --version  # should be 20.0 or newer
pip install pikepdf
```

32- and 64-bit wheels are available for Windows, Linux and macOS. Binary wheels should work on most systems, i.e. Linux distributions 2010 and newer, macOS 10.11 and newer (for Homebrew), Windows 7 and newer, **provided a recent version of pip is used to install them**. The Linux wheels currently include copies of libqpdf, libjpeg, and zlib. The Windows wheels include libqpdf. This is to ensure that up-to-date, compatible copies of dependent libraries are included.

Currently we do not build wheels for architectures other than x86 and x64.

Alpine Linux does not support Python wheels.

Platform support

Some platforms include versions of pikepdf that are distributed by the system package manager (such as `apt`). These versions may lag behind the version distributed with PyPI, but may be convenient for users that cannot use binary wheels.

Fig. 4: Packaged fish.

Debian, Ubuntu and other APT-based distributions

```
apt install pikepdf
```

Fedora

```
dnf install python-pikepdf
```

Alpine Linux

```
apk add py3-pikepdf
```

Installing on FreeBSD

```
pkg install py37-pikepdf
```

To attempt a manual install, try something like:

```
pkg install␣
↪python3 py37-lxml py37-pip py37-pybind11 qpdf
pip install --user pikepdf
```

This procedure is known to work on FreeBSD 11.3, 12.0, 12.1-RELEASE and 13.0-CURRENT. It has not been tested on other versions.

Building from source

Requirements

pikepdf requires:

- a C++14 compliant compiler - GCC (5 and up), clang (3.3 and up), MSVC (2015 or newer)
- pybind11
- libqpdf 10.0.3 or higher from the QPDF project.

On Linux the library and headers for libqpdf must be installed because pikepdf compiles code against it and links to it.

Check Repology for QPDF to see if a recent version of QPDF is available for your platform. Otherwise you must build QPDF from source. (Consider using the binary wheels, which bundle the required version of libqpdf.)

Compiling with GCC or Clang

- clone this repository
- install libjpeg, zlib and libqpdf on your platform, including headers
- `pip install .`

Note: pikepdf should be built with the same compiler and linker as libqpdf; to be precise both **must** use the same C++ ABI. On some platforms, setup.py may not pick the correct compiler so one may need to set environment variables `CC` and `CXX` to redirect it. If the wrong compiler is selected, `import pikepdf._qpdf` will throw an `ImportError` about a missing symbol.

On Windows (requires Visual Studio 2015)

pikepdf requires a C++14 compliant compiler (i.e. Visual Studio 2015 on Windows). See our continuous integration build script in `.appveyor.yml` for detailed and current instructions. Or use the wheels which save this pain.

These instructions require the precompiled binary `qpdf.dll`. See the QPDF documentation if you also need to build this DLL from source. Both should be built with the same compiler. You may not mix and match MinGW and Visual C++ for example.

Running a regular `pip install` command will detect the version of the compiler used to build Python and attempt to build the extension with it. We must force the use of Visual Studio 2015.

1. Clone this repository.
2. In a command prompt, run:

```
%VS140COMNTOOLS%\..\..\VC\vcvarsall.bat" x64
set DISTUTILS_USE_SDK=1
set MSSdk=1
```

3. Download qpdf-10.0.3-bin-msvc64.zip from the QPDF releases page.
4. Extract `bin\*.dll` (all the DLLs, both QPDF's and the Microsoft Visual C++ Runtime library) from the zip file above, and copy it to the `src/pikepdf` folder in the repository.
5. Run `pip install .` in the root directory of the repository.

Note: The user compiling `pikepdf` to must have registry editing rights on the machine to be able to run the `vcvarsall.bat` script.

Building against a QPDF source tree

Follow these steps to build pikepdf against a different version of QPDF, rather than the one provided with your operating system. This may be useful if you need a more recent version of QPDF than your operating system package manager provides, and you do not want to use Python wheels.

- Set the environment variable `QPDF_SOURCE_TREE` to the location of the QPDF source tree.
- Build QPDF, by running `make`. Refer to the QPDF installation instructions for further options and details.
- On Linux, modify `LD_LIBRARY_PATH`, prepending the path where the QPDF build produces `libqpdfXX.so`. This might be something like `$QPDF_SOURCE_TREE/.build/libs/libqpdfXX.so`. On macOS, locate the equivalent variable is `DYLD_LIBRARY_PATH`. On Windows, no action is needed. Generally, what you are doing here is telling the runtime dynamic linker to use the custom compiled version of QPDF instead of the system version.
- Build pikepdf. On Windows, locate the QPDF .dll files and copy them into the folder alongside the file named `_qpdf*.dll`.

Note that the Python wheels for pikepdf currently compile their own version of QPDF and several of its dependencies to ensure the wheels have the latest version. You can also refer to the GitHub Actions YAML files for build steps.

Building against a custom install of QPDF to /usr/local/lib

If you have previously installed a QPDF from source to `/usr/local/lib` on a POSIX platform, and you try to build pikepdf from source, it will prefer the operating system version of QPDF installed at `/usr/lib`. Since pikepdf strongly prefers recent versions of QPDF, you may want to use a more current version.

From a Git checkout of the pikepdf source tree, run:

```
env LDFLAGS='-L/usr/local/lib' CFLAGS='-I/usr/local/include/qpdf' pip install .
```

Building the documentation

Documentation is generated using Sphinx and you are currently reading it. To regenerate it:

```
pip install -r requirements/docs.txt
cd docs
make html
```

PyPy3 support

PyPy3 3.6 and 3.7 are currently supported. However, binary wheels for PyPy3 are not available for some platforms, since some dependencies of pikepdf (namely lxml) do not yet generate PyPy3 wheels of their own.

Platform	Source build supported	Wheel
Windows 64-bit		
Linux 64-bit		
macOS		

PyPy3 is not more performant than CPython for pikepdf, because the core of pikepdf is already written in C++. The benefit is for applications that want to use PyPy for improved performance of native Python and also want to use pikepdf.

1.3.2 Release notes

pikepdf releases use the semantic versioning policy.

The pikepdf API (as provided by `import pikepdf`) is stable and is in production use. Note that the C++ extension module `pikepdf._qpdf` is a private interface within pikepdf that applications should not access directly, along with any modules with a prefixed underscore.

Fig. 5: Releasing a pike.

v2.11.1

- Fixed an issue with `Object.emplace()` not retaining the original object's /Parent.
- Code coverage improvements.

v2.11.0

- Add new functions: `Pdf.generate_appearance_streams` and `Pdf.flatten_annotations`, to support common work with PDF forms.
- Fixed an issue with `pip install` on platforms that lack proper multiprocessing support.
- Additional documentation improvements from @m-holger - thanks again!

v2.10.0

- Fixed a XML External Entity (XXE) processing vulnerability in PDF XMP metadata parsing. (Reported by Eric Therond of Sonarsource.) All users should upgrade to get this security update. CVE-2021-29421 was assigned to this issue.
- Bind new functions to check, when a PDF is opened, whether the password used to open the PDF matched the owner password, user password, or both: `Pdf.user_password_matched` and `Pdf.owner_password_matched`.

v2.9.2

- Further expansion of test coverage of several functions, and minor bug fixes along the way.
- Improve parameter validation for some outline-related functions.
- Fixed overloaded `__repr__` functions in `_methods.py` not being applied.
- Some proofreading of the documentation by @m-holger - thanks!

v2.9.1

- Further expansion of test coverage.
- Fixed function signatures for `_repr_mimebundle_` functions to match IPython's spec.
- Fixed some error messages regarding attempts to do strange things with `pikepdf.Name`, like `pikepdf.Name.Foo = 3`.
- Eliminated code to handle an exception that provably does not occur.
- Test suite is now better at closing open file handles.
- Ensure that any demo code in README.md is valid and works.
- Embedded QPDF version in pikepdf Python wheels increased to 10.3.1.

v2.9.0

- We now issue a warning when attempting to use `pikepdf.open` on a `bytes` object where it could be either a PDF loaded into memory or a filename.
- `pikepdf.Page.label` will now return the "ordinary" page number if no special rules for pages are defined.
- Many improvements to tests and test coverage. Code coverage for both Python and C++ is now automatically published to codecov.io; previously coverage was only checked on the developer's machine.
- An obsolete private function `Object._roundtrip` was removed.

v2.8.0

- Fixed an issue with extracting data from images that had their DecodeParms structured as a list of dictionaries.
- Fixed an issue where a dangling stream object is created if we fail to create the requested stream dictionary.
- Calling `Dictionary()` and `Array()` on objects which are already of that type returns a shallow copy rather than throwing an exception, in keeping with Python semantics.
- **v2.8.0.post1**: The CI system was changed from Azure Pipelines to GitHub Actions, a transition we made to support generating binary wheels for more platforms. This post-release was the first release made with GitHub Actions. It ought to be functionally identical, but could different in some subtle way, for example because parts of it may have been built with different compiler versions.
- **v2.8.0.post2**: The previous .post1 release caused binary wheels for Linux to grow much larger, causing problems for AWS Lambda who require small file sizes. This change strips the binaries of debug symbols, also mitigates a rare PyPy test failure.
- Unfortunately, it appears that the transition from Azure Pipelines to GitHub Actions broke compatibility with macOS 10.13 and older. macOS 10.13 and older are considered end of life by Apple. No version of pikepdf v2.x ever promised support for macOS 10.13 – 10.14+ has always been an explicit requirement. It just so happens that for some time, pikepdf did actually work on 10.13.

v2.7.0

- Added an option to tell `Pdf.save` to recompress flate streams, and a global option to set the flate compression level. This option can be use to force the recompression of flate streams if they are not well compressed.
- Fixed "TypeError: only pages can be inserted" when attempting to an insert an unowned page using QPDF 10.2.0 or later.

v2.6.0

- Rebuild wheels against QPDF 10.2.0.

v2.5.2

- Fixed support for PyPy 3.7 on macOS.

v2.5.1

- Rebuild wheels against recently released pybind11 v2.6.2.
- Improved support for building against PyPy 3.6/7.3.1.

v2.5.0

- PyPy3 is now supported.
- Improved test coverage for some metadata issues.

v2.4.0

- The DocumentInfo dictionary can now be deleted with `del pdf.docinfo`.
- Fixed issues with updating the `dc:creator` XMP metadata entry.
- Improved error messages on attempting to encode strings containing Unicode surrogates.
- Fixed a rare random test failure related to strings containing Unicode surrogates.

v2.3.0

- Fixed two tests that failed with libqpdf 10.1.0.
- Add new function `pikepdf.Page.add_resource` which helps with adding a new object to the /Resources dictionary.
- Binary wheels now provide libqpdf 10.1.0.

v2.2.5

- Changed how one C++ function is called to support libqpdf 10.1.0.

v2.2.4

- Fixed another case where pikepdf should not be warning about metadata updates.

v2.2.3

- Fixed a warning that was incorrectly issued in v2.2.2 when pikepdf updates XMP metadata on the user's behalf.
- Fixed a rare test suite failure that occurred if two test files were generated with a different timestamp, due to timing of the tests.
- Hopefully fixed build on Cygwin (not tested, based on user report).

v2.2.2

- Fixed issue #150, adding author metadata breaks PDF/A conformance. We now log an error when this metadata is set incorrectly.
- Improve type checking in ocrmypdf.models.metadata module.
- Improve documentation for custom builds.

v2.2.1

- Fixed issue #143, PDF/A validation with veraPDF failing due to missing prefix on DocumentInfo dates.

v2.2.0

- Added features to look up the index of an page in the document and page labels
- Enable parallel compiling (again)
- Make it easier to create a `pikepdf.Stream` with a dictionary or from an existing dictionary.
- Converted most `.format()` strings to f-strings.
- Fixed incorrect behavior when assigning `Object.stream_dict`; this use to create a dictionary in the wrong place instead of overriding a stream's dictionary.

v2.1.2

- Fixed an issue the XMP metadata would not have a timezone set when updated. According to the XMP specification, the timezone should be included. Note that pikepdf will include the local machine timezone, unless explicitly directed otherwise.

v2.1.1

- The previous release inadvertently changed the type of exception in certain situations, notably throwing `ForeignObjectError` when this was not the correct error to throw. This release fixes that.

v2.1.0

- Improved error messages and documentation around `Pdf.copy_foreign`.
- Opt-in to mypy typing.

v2.0.0

This description includes changes in v2.0 beta releases.

Breaking changes

- We now require at least these versions or newer: - Python 3.6 - pybind11 2.6.0 - QPDF 10.0.3 - For macOS users, macOS 10.14 (Mojave)
- Attempting to modifying `Stream.Length` will raise an exception instead of a warning.
- `pikepdf.Stream()` can no longer parse content streams. That never made sense, since this class supports streams in general, and many streams are not content streams. Use `pikepdf.parse_content_stream` to a parse a content stream.
- `pikepdf.Permissions` is now represented as a `NamedTuple`. Probably not a concern unless some user made strong assumptions about this class and its superclass.
- Fixed the behavior of the `__eq__` on several classes to return `NotImplemented` for uncomparable objects, instead of `False`.
- The instance variable `PdfJpxImage.pil` is now a private variable.

New features

- Python 3.9 is supported.
- Significantly improved type hinting, including hints for functions written in C++.
- Documentation updates

Deprecations - `Pdf.root` is deprecated. Use `Pdf.Root`.

v2.0.0b2

- We now require QPDF 10.0.3.

v2.0.0b1

Breaking changes

- We now require at least these versions or newer: - Python 3.6 - pybind11 2.6.0 - QPDF 10.0.1 - For macOS users, macOS 10.14 (Mojave)
- Attempting to modifying `Stream.Length` will raise an exception instead of a warning.
- `pikepdf.Stream()` can no longer parse content streams. That never made sense, since this class supports streams in general, and many streams are not content streams. Use `pikepdf.parse_content_stream` to a parse a content stream.
- `pikepdf.Permissions` is now represented as a `NamedTuple`. Probably not a concern unless some user made strong assumptions about this class and its superclass.
- Fixed the behavior of the `__eq__` on several classes to return `NotImplemented` for uncomparable objects, instead of `False`.

New features

- Python 3.9 is supported.
- Significantly improved type hinting, including hints for functions written in C++.

v1.19.4

- Modify project settings to declare no support for Python 3.9 in pikepdf 1.x. pybind11 upstream has indicated there are stability problems when pybind11 2.5 (used by pikepdf 1.x) is used with Python 3.9. As such, we are marking Python 3.9 as unsupported by pikepdf 1.x. Python 3.9 users should switch to pikepdf 2.x.

v1.19.3

- Fixed an exception that occurred when building the documentation, introduced in the previous release.

v1.19.2

- Fixed an exception with setting metadata objects to unsupported RDF types. Instead we make a best effort to convert to an appropriate type.
- Prevent creating certain illegal dictionary key names.
- Document procedure to remove an image.

v1.19.1

- Fixed an issue with `unparse_content_stream`: we now assume the second item of each step in the content stream is an `Operator`.
- Fixed an issue with unparsing inline images.

v1.19.0

- Learned how to export CCITT images from PDFs that have ICC profiles attached.
- Cherry-picked a workaround to a possible use-after-free caused by pybind11 (pybind11 PR 2223).
- Improved test coverage of code that handles inline images.

v1.18.0

- You can now use `pikepdf.open(...allow_overwriting_input=True)` to allow overwriting the input file, which was previously forbidden because it can corrupt data. This is accomplished safely by loading the entire PDF into memory at the time it is opened rather than loading content as needed. The option is disabled by default, to avoid a performance hit.
- Prevent setup.py from creating junk temporary files (finally!)

v1.17.3

- Fixed crash when `pikepdf.Pdf` objects are used inside generators (#114) and not freed or closed before the generator exits.

v1.17.2

- Fixed issue, "seek of closed file" where JBIG2 image data could not be accessed (only metadata could be) when a JBIG2 was extracted from a PDF.

v1.17.1

- Fixed building against the oldest supported version of QPDF (8.4.2), and configure CI to test against the oldest version. (#109)

v1.17.0

- Fixed a failure to extract PDF images, where the image had both a palette and colorspace set to an ICC profile. The iamge is now extracted with the profile embedded. (#108)
- Added opt-in support for memory-mapped file access, using `pikepdf.open(... access_mode=pikepdf.AccessMode.mmap)`. Memory mapping file access performance considerably, but may make application exception handling more difficult.

v1.16.1

- Fixed an issue with JBIG2 extraction, where the version number of the jbig2dec software may be written to standard output as a side effect. This could interfere with test cases or software that expects pikepdf to be stdout-clean.
- Fixed an error that occurred when updating DocumentInfo to match XMP metadata, when XMP metadata had unexpected empty tags.
- Fixed setup.py to better support Python 3.8 and 3.9.
- Documentation updates.

v1.16.0

- Added support for extracting JBIG2 images with the image API. JBIG2 images are converted to `PIL.Image`. Requires a JBIG2 decoder such as jbig2dec.
- Python 3.5 support is deprecated and will end when Python 3.5 itself reaches end of life, in September 2020. At the moment, some tests are skipped on Python 3.5 because they depend on Python 3.6.
- Python 3.9beta is supported and is known to work on Fedora 33.

v1.15.1

- Fixed a regression - `Pdf.save(filename)` may hold file handles open after the file is fully written.
- Documentation updates.

v1.15.0

- Fixed an issue where `Decimal` objects of precision exceeding the PDF specification could be written to output files, causing some PDF viewers, notably Acrobat, to parse the file incorrectly. We now limit precision to 15 digits, which ought to be enough to prevent rounding error and parsing errors.
- We now refuse to create pikepdf objects from `float` or `Decimal` that are `NaN` or `±Infinity`. These concepts have no equivalent in PDF.
- `pikepdf.Array` objects now implement `.append()` and `.extend()` with familiar Python `list` semantics, making them easier to edit.

v1.14.0

- Allowed use of `.keys()`, `.items()` on `pikepdf.Stream` objects.
- We now warn on attempts to modify `pikepdf.Stream.Length`, which pikepdf will manage on its own when the stream is serialized. In the future attempting to change it will become an error.
- Clarified documentation in some areas about behavior of `pikepdf.Stream`.

v1.13.0

- Added support for editing PDF Outlines (also known as bookmarks or the table of contents). Many thanks to Matthias Erll for this contribution.
- Added support for decoding run length encoded images.
- `Object.read_bytes()` and `Object.get_stream_buffer()` can now request decoding of uncommon PDF filters.
- Fixed test suite warnings related to pytest and hypothesis.
- Fixed build on Cygwin. Thanks to @jhgarrison for report and testing.

v1.12.0

- Microsoft Visual C++ Runtime libraries are now included in the pikepdf Windows wheel, to improve ease of use on Windows.
- Defensive code added to prevent using `.emplace()` on objects from a foreign PDF without first copying the object. Previously, this would raise an exception when the file was saved.

v1.11.2

- Fix "error caused by missing str function of Array" (#100, #101).
- Lots of delinting and minor fixes.

v1.11.1

- We now avoid creating an empty XMP metadata entry when files are saved.
- Updated documentation to describe how to delete the document information dictionary.

v1.11.0

- Prevent creation of dictionaries with invalid names (not beginning with `/`).
- Allow pikepdf's build to specify a qpdf source tree, allowing one to compile pikepdf against an unreleased/modified version of qpdf.
- Improved behavior of `pages.p()` and `pages.remove()` when invalid parameters were given.
- Fixed compatibility with libqpdf version 10.0.1, and build official wheels against this version.
- Fixed compatibility with pytest 5.x.
- Fixed the documentation build.
- Fixed an issue with running tests in a non-Unicode locale.
- Fixed a test that randomly failed due to a "deadline error".
- Removed a possibly nonfree test file.

v1.10.4

- Rebuild Python wheels with newer version of libqpdf. Fixes problems with opening certain password-protected files (#87).

v1.10.3

- Fixed `isinstance(obj, pikepdf.Operator)` not working. (#86)
- Documentation updates.

v1.10.2

- Fixed an issue where pages added from a foreign PDF were added as references rather than copies. (#80)
- Documentation updates.

v1.10.1

- Fixed build reproducibility (thanks to @lamby)
- Fixed a broken link in documentation (thanks to @maxwell-k)

v1.10.0

- Further attempts to recover malformed XMP packets.
- Added missing functionality to extract 1-bit palette images from PDFs.

v1.9.0

- Improved a few cases of malformed XMP recovery.
- Added an `unparse_content_stream` API to assist with converting the previously parsed content streams back to binary.

v1.8.3

- If the XMP metadata packet is not well-formed and we are confident that it is essentially empty apart from XML fluff, we fix the problem instead of raising an exception.

v1.8.2

- Fixed an issue where QPDF 8.4.2 would report different errors from QPDF 9.0.0, causing a test to fail. (#71)

v1.8.1

- Fixed an issue where files opened by name may not be closed correctly. Regression from v1.8.0.
- Fixed test for readable/seekable streams evaluated to always true.

v1.8.0

- Added API/property to iterate all objects in a PDF: `pikepdf.Pdf.objects`.
- Added `pikepdf.Pdf.check()`, to check for problems in the PDF and return a text description of these problems, similar to `qpdf --check`.
- Improved internal method for opening files so that the code is smaller and more portable.
- Added missing licenses to account for other binaries that may be included in Python wheels.
- Minor internal fixes and improvements to the continuous integration scripts.

v1.7.1

- This release was incorrectly marked as a patch-level release when it actually introduced one minor new feature. It includes the API change to support `pikepdf.Pdf.objects`.

v1.7.0

- Shallow object copy with `copy.copy(pikepdf.Object)` is now supported. (Deep copy is not yet supported.)
- Support for building on C++11 has been removed. A C++14 compiler is now required.
- pikepdf now generates manylinux2010 wheels on Linux.
- Build and deploy infrastructure migrated to Azure Pipelines.
- All wheels are now available for Python 3.5 through 3.8.

v1.6.5

- Fixed build settings to support Python 3.8 on macOS and Linux. Windows support for Python 3.8 is not currently tested since continuous integration providers have not updated to Python 3.8 yet.
- pybind11 2.4.3 is now required, to support Python 3.8.

v1.6.4

- When images were encoded with CCITTFaxDecode, type G4, with the /EncodedByteAlign set to true (not default), the image extracted by pikepdf would be a corrupted form of the original, usually appearing as a small speckling of black pixels at the top of the page. Saving an image with pikepdf was not affected; this problem only occurred when attempting to extract images. We now refuse to extract images with these parameters, as there is not sufficient documentation to determine how to extract them. This image format is relatively rare.

v1.6.3

- Fixed compatibility with libqpdf 9.0.0.
  - A new method introduced in libqpdf 9.0.0 overloaded an older method, making a reference to this method in pikepdf ambiguous.
  - A test relied on libqpdf raising an exception when a pikepdf user called `Pdf.save(..., min_version='invalid')`. libqpdf no longer raises an exception in this situation, but ignores the invalid version. In the interest of supporting both versions, we defer to libqpdf. The failing test is removed, and documentation updated.
- Several warnings, most specific to the Visual C++ compiler, were fixed.
- The Windows CI scripts were adjusted for the change in libqpdf ABI version.
- Wheels are now built against libqpdf 9.0.0.
- libqpdf 8.4.2 and 9.0.0 are both supported.

v1.6.2

- Fixed another build problem on Alpine Linux - musl-libc defines `struct FILE` as an incomplete type, which breaks pybind11 metaprogramming that attempts to reason about the type.
- Documentation improved to mention FreeBSD port.

v1.6.1

- Dropped our one usage of QPDF's C API so that we use only C++.
- Documentation improvements.

v1.6.0

- Added bindings for QPDF's page object helpers and token filters. These enable: filtering content streams, capturing pages as Form XObjects, more convenient manipulation of page boxes.
- Fixed a logic error on attempting to save a PDF created in memory in a way that overwrites an existing file.
- Fixed `Pdf.get_warnings()` failed with an exception when attempting to return a warning or exception.
- Improved manylinux1 binary wheels to compile all dependencies from source rather than using older versions.
- More tests and more coverage.
- libqpdf 8.4.2 is required.

v1.5.0

- Improved interpretation of images within PDFs that use an ICC colorspace. Where possible we embed the ICC profile when extracting the image, and profile access to the ICC profile.
- Fixed saving PDFs with their existing encryption.
- Fixed documentation to reflect the fact that saving a PDF without specifying encryption settings will remove encryption.
- Added a test to prevent overwriting the input PDF since overwriting corrupts lazy loading.
- `Object.write(filters=, decode_parms=)` now detects invalid parameters instead of writing invalid values to `Filters` and `DecodeParms`.
- We can now extract some images that had stacked compression, provided it is `/FlateDecode`.
- Add convenience function `Object.wrap_in_array()`.

v1.4.0

- Added support for saving encrypted PDFs. (Reading them has been supported for a long time.)
- Added support for setting the PDF extension level as well as version.
- Added support converting strings to and from PDFDocEncoding, by registering a `"pdfdoc"` codec.

v1.3.1

- Updated pybind11 to v2.3.0, fixing a possible GIL deadlock when pikepdf objects were shared across threads. (#27)
- Fixed an issue where PDFs with valid XMP metadata but missing an element that is usually present would be rejected as malformed XMP.

v1.3.0

- Remove dependency on `defusedxml.lxml`, because this library is deprecated. In the absence of other options for XML hardening we have reverted to standard `lxml`.
- Fixed an issue where `PdfImage.extract_to()` would write a file in the wrong directory.
- Eliminated an intermediate buffer that was used when saving to an IO stream (as opposed to a filename). We would previously write the entire output to a memory buffer and then write to the output buffer; we now write directly to the stream.
- Added `Object.emplace()` as a workaround for when one wants to update a page without generating a new page object so that links/table of contents entries to the original page are preserved.
- Improved documentation. Eliminated all `arg0` placeholder variable names, which appeared when the documentation generator could not read a C++ variable name.
- Added `PageList.remove(p=1)`, so that it is possible to remove pages using counting numbers.

v1.2.0

- Implemented `Pdf.close()` and `with`-block context manager, to allow Pdf objects to be closed without relying on `del`.
- `PdfImage.extract_to()` has a new keyword argument `fileprefix=`, which to specify a filepath where an image should be extracted with pikepdf setting the appropriate file suffix. This simplifies the API for the most common case of extracting images to files.
- Fixed an internal test that should have suppressed the extraction of JPEGs with a nonstandard ColorTransform parameter set. Without the proper color transform applied, the extracted JPEGs will typically look very pink. Now, these images should fail to extract as was intended.
- Fixed that `Pdf.save(object_stream_mode=...)` was ignored if the default `fix_metadata_version=True` was also set.
- Data from one `Pdf` is now copied to other `Pdf` objects immediately, instead of creating a reference that required source PDFs to remain available. `Pdf` objects no longer reference each other.
- libqpdf 8.4.0 is now required
- Various documentation improvements

v1.1.0

- Added workaround for macOS/clang build problem of the wrong exception type being thrown in some cases.
- Improved translation of certain system errors to their Python equivalents.
- Fixed issues resulting from platform differences in `datetime.strftime`. (#25)
- Added `Pdf.new`, `Pdf.add_blank_page` and `Pdf.make_stream` convenience methods for creating new PDFs from scratch.
- Added binding for new QPDF JSON feature: `Object.to_json`.
- We now automatically update the XMP PDFVersion metadata field to be consistent with the PDF's declared version, if the field is present.
- Made our Python-augmented C++ classes easier for Python code inspectors to understand.
- Eliminated use of the `imghdr` library.
- Autoformatted Python code with black.
- Fixed handling of XMP metadata that omits the standard `<x:xmpmeta>` wrapper.

v1.0.5

- Fixed an issue where an invalid date in XMP metadata would cause an exception when updating DocumentInfo. For now, we warn that some DocumentInfo is not convertible. (In the future, we should also check if the XMP date is valid, because it probably is not.)
- Rebuilt the binary wheels with libqpdf 8.3.0. libqpdf 8.2.1 is still supported.

v1.0.4

- Updates to tests/resources (provenance of one test file, replaced another test file with a synthetic one)

v1.0.3

- Fixed regression on negative indexing of pages.

v1.0.2

- Fixed an issue where invalid values such as out of range years (e.g. 0) in DocumentInfo would raise exceptions when using DocumentInfo to populate XMP metadata with `.load_from_docinfo`.

v1.0.1

- Fixed an exception with handling metadata that contains the invalid XML entity `&#0;` (an escaped NUL)

v1.0.0

- Changed version to 1.0.

v0.10.2

Fixes

- Fixed segfault when overwriting the pikepdf file that is currently open on Linux.
- Fixed removal of an attribute metadata value when values were present on the same node.

v0.10.1

Fixes

- Avoid canonical XML since it is apparently too strict for XMP.

v0.10.0

Fixes

- Fixed several issues related to generating XMP metadata that passed veraPDF validation.
- Fixed a random test suite failure for very large negative integers.
- The lxml library is now required.

v0.9.2

Fixes

- Added all of the commonly used XML namespaces to XMP metadata handling, so we are less likely to name something 'ns1', etc.
- Skip a test that fails on Windows.
- Fixed build errors in documentation.

v0.9.1

Fixes

- Fix `Object.write()` accepting positional arguments it wouldn't use
- Fix handling of XMP data with timezones (or missing timezone information) in a few cases
- Fix generation of XMP with invalid XML characters if the invalid characters were inside a non-scalar object

v0.9.0

Updates

- New API to access and edit PDF metadata and make consistent edits to the new and old style of PDF metadata.
- 32-bit binary wheels are now available for Windows
- PDFs can now be saved in QPDF's "qdf" mode
- The Python package defusedxml is now required
- The Python package python-xmp-toolkit and its dependency libexempi are suggested for testing, but not required

Fixes

- Fixed handling of filenames that contain multibyte characters on non-UTF-8 systems

Breaking

- The `Pdf.metadata` property was removed, and replaced with the new metadata API
- `Pdf.attach()` has been removed, because the interface as implemented had no way to deal with existing attachments.

v0.3.7

- Add API for inline images to unparse themselves

v0.3.6

- Performance of reading files from memory improved to avoid unnecessary copies.
- It is finally possible to use `for key in pdfobj` to iterate contents of PDF Dictionary, Stream and Array objects. Generally these objects behave more like Python containers should now.
- Package API declared beta.

v0.3.5

Breaking

- `Pdf.save(...stream_data_mode=...)` has been dropped in favor of the newer `compress_streams=` and `stream_decode_level` parameters.

Fixes

- A use-after-free memory error that caused occasional segfaults and "QPDFFakeName" errors when opening from stream objects has been resolved.

v0.3.4

Updates

- pybind11 vendoring has ended now that v2.2.4 has been released

v0.3.3

Breaking

- libqpdf 8.2.1 is now required

Updates

- Improved support for working with JPEG2000 images in PDFs
- Added progress callback for saving files, `Pdf.save(..., progress=)`
- Updated pybind11 subtree

Fixes

- `del obj.AttributeName` was not implemented. The attribute interface is now consistent
- Deleting named attributes now defers to the attribute dictionary for Stream objects, as get/set do
- Fixed handling of JPEG2000 images where metadata must be retrieved from the file

v0.3.2

Updates

- Added support for direct image extraction of CMYK and grayscale JPEGs, where previously only RGB (internally YUV) was supported
- `Array()` now creates an empty array properly
- The syntax `Name.Foo in Dictionary()`, e.g. `Name.XObject in page.Resources`, now works

v0.3.1

Breaking

- `pikepdf.open` now validates its keyword arguments properly, potentially breaking code that passed invalid arguments
- libqpdf 8.1.0 is now required - libqpdf 8.1.0 API is now used for creating Unicode strings
- If a non-existent file is opened with `pikepdf.open`, a `FileNotFoundError` is raised instead of a generic error
- We are now *temporarily* vendoring a copy of pybind11 since its master branch contains unreleased and important fixes for Python 3.7.

Updates

- The syntax `Name.Thing` (e.g. `Name.DecodeParms`) is now supported as equivalent to `Name('/Thing')` and is the recommended way to refer names within a PDF
- New API `Pdf.remove_unneeded_resources()` which removes objects from each page's resource dictionary that are not used in the page. This can be used to create smaller files.

Fixes

- Fixed an error parsing inline images that have masks
- Fixed several instances of catching C++ exceptions by value instead of by reference

v0.3.0

Breaking

- Modified `Object.write` method signature to require `filter` and `decode_parms` as keyword arguments
- Implement automatic type conversion from the PDF Null type to `None`
- Removed `Object.unparse_resolved` in favor of `Object.unparse(resolved=True)`
- libqpdf 8.0.2 is now required at minimum

Updates

- Improved IPython/Jupyter interface to directly export temporary PDFs
- Updated to qpdf 8.1.0 in wheels
- Added Python 3.7 support for Windows
- Added a number of missing options from QPDF to `Pdf.open` and `Pdf.save`
- Added ability to delete a slice of pages
- Began using Jupyter notebooks for documentation

v0.2.2

- Added Python 3.7 support to build and test (not yet available for Windows, due to lack of availability on Appveyor)
- Removed setter API from `PdfImage` because it never worked anyway
- Improved handling of `PdfImage` with trivial palettes

v0.2.1

- `Object.check_owner` renamed to `Object.is_owned_by`
- `Object.objgen` and `Object.get_object_id` are now public functions
- Major internal reorganization with `pikepdf.models` becoming the submodule that holds support code to ease access to PDF objects as opposed to wrapping QPDF.

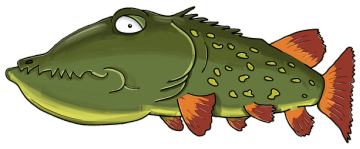
v0.2.0

- Implemented automatic type conversion for `int`, `bool` and `Decimal`, eliminating the `pikepdf.{Integer,Boolean,Real}` types. Removed a lot of associated numerical code.

Everything before v0.2.0 can be considered too old to document.

1.3.3 Tutorial

This brief tutorial should give you an introduction and orientation to pikepdf's paradigm and syntax. From there, we refer to you various topics.



Opening and saving PDFs

In contrast to better known PDF libraries, pikepdf uses a single object to represent a PDF, whether reading, writing or merging. We have cleverly named this *pikepdf.Pdf*. In this documentation, a `Pdf` is a class that allows manipulate the PDF, meaning the file.

```
from pikepdf import Pdf
new_pdf = Pdf.new()
with Pdf.open('sample.pdf') as pdf:
    pdf.save('output.pdf')
```

You may of course use `from pikepdf import Pdf as ...` if the short class name conflicts or `from pikepdf import Pdf as PDF` if you prefer uppercase.

pikepdf.open() is a shorthand for *pikepdf.Pdf.open()*.

The PDF class API follows the example of the widely-used Pillow image library. For clarity there is no default constructor since the arguments used for creation and opening are different. To make a new empty PDF, use `Pdf.new()` not `Pdf()`.

`Pdf.open()` also accepts seekable streams as input, and `Pdf.save()` accepts streams as output. `pathlib.Path` objects are fully supported wherever pikepdf accepts a filename.

Inspecting pages

Manipulating pages is fundamental to PDFs. pikepdf presents the pages in a PDF through the *pikepdf.Pdf.pages* property, which follows the `list` protocol. As such page numbers begin at 0.

Let's open a simple PDF that contains four pages.

```
In [1]: from pikepdf import Pdf

In [2]: pdf = Pdf.open('../tests/resources/fourpages.pdf')
```

How many pages?

```
In [3]: len(pdf.pages)
Out[3]: 4
```

pikepdf integrates with IPython and Jupyter's rich object APIs so that you can view PDFs, PDF pages, or images within PDF in a IPython window or Jupyter notebook. This makes it easier to test visual changes.

```
In [4]: pdf
Out[4]: « In Jupyter you would see the PDF here »

In [5]: pdf.pages[0]
Out[5]: « In Jupyter you would see an image of the PDF page here »
```

You can also examine individual pages, which we'll explore in the next section. Suffice to say that you can access pages by indexing them and slicing them.

```
In [6]: pdf.pages[0]
Out[6]: « In Jupyter you would see an image of the PDF page here »
```

Note: *pikepdf.Pdf.open()* can open almost all types of encrypted PDF! Just provide the `password=` keyword argument.

For more details on document assembly, see *PDF split, merge and document assembly*.

Pages are dictionaries

In PDFs, the main data structure is the **dictionary**, a key-value data structure much like a Python `dict` or `attrdict`. The major difference is that the keys can only be **names**, and the values can only be PDF types, including other dictionaries.

PDF dictionaries are represented as *pikepdf.Dictionary* objects, and names are of type *pikepdf.Name*. A page is just a dictionary with certain required keys that is referenced by the PDF's "page tree". (pikepdf manages the page tree for you.)

```
In [7]: from pikepdf import Pdf

In [8]: example = Pdf.open('../tests/resources/congress.pdf')

In [9]: page1 = example.pages[0]
```

repr() output

Let's example the page's `repr()` output:

```
In [10]: repr(page1)
Out[10]: '<pikepdf.Dictionary(Type="/Page")({\n  "/Contents": pikepdf.Stream(stream_
↪dict={\n      "/Length": 50\n    }, data=<...>),\n  "/MediaBox": [ 0, 0, 200, 304 ],
↪\n  "/Parent": <reference to /Pages>,\n  "/Resources": {\n    "/XObject": {\n
↪"/Im0": pikepdf.Stream(stream_dict={\n          "/BitsPerComponent": 8,\n
↪"/ColorSpace": "/DeviceRGB",\n          "/Filter": [ "/DCTDecode" ],\n          "/
↪Height": 1520,\n          "/Length": 192956,\n          "/Subtype": "/Image",\n
↪      "/Type": "/XObject",\n          "/Width": 1000\n        }, data=<...>)\n    }\
↪n  },\n  "/Type": "/Page"\n})>'
```

The angle brackets in the output indicate that this object cannot be constructed with a Python expression because it contains a reference. When angle brackets are omitted from the `repr()` of a pikepdf object, then the object can be replicated with a Python expression, such as `eval(repr(x)) == x`. Pages typically have indirect references to themselves and other pages, so they cannot be represented as an expression.

Item and attribute notation

Dictionary keys may be looked up using attributes (`page1.MediaBox`) or keys (`page1['/MediaBox']`).

```
In [11]: page1.MediaBox      # preferred notation for standard PDF names
Out[11]: pikepdf.Array([ 0, 0, 200, 304 ])

In [12]: page1['/MediaBox']  # also works
Out[12]: pikepdf.Array([ 0, 0, 200, 304 ])
```

By convention, pikepdf uses attribute notation for standard names (the names that are normally part of a dictionary, according to the PDF Reference Manual), and item notation for names that may not always appear. For example, the images belong to a page always appear at `page.Resources.XObject` but the names of images are arbitrarily chosen by whatever software generates the PDF (`/Im0`, in this case). (Whenever expressed as strings, names begin with `/`.)

```
In [13]: page1.Resources.XObject['/Im0']
```

Item notation here would be quite cumbersome: `['/Resources']['/XObject]['/Im0']` (not recommended).

Attribute notation is convenient, but not robust if elements are missing. For elements that are not always present, you can use `.get()`, which behaves like `dict.get()` in core Python. A library such as glom might help when working with complex structured data that is not always present.

(For now, we'll set aside what a page's `MediaBox` and `Resources.XObject` are for. See *Working with pages* for details.)

Deleting pages

Removing pages is easy too.

```
In [14]: del pdf.pages[1:3]  # Remove pages 2-3 labeled "second page" and "third page"
```

```
In [15]: len(pdf.pages)
Out[15]: 2
```

Saving changes

Naturally, you can save your changes with *pikepdf.Pdf.save()*. `filename` can be a `pathlib.Path`, which we accept everywhere.

```
In [16]: pdf.save('output.pdf')
```

You may save a file multiple times, and you may continue modifying it after saving. For example, you could create an unencrypted version of document, then apply a watermark, and create an encrypted version.

Fig. 6: Saving pike.

Note: You may not overwrite the input file (or whatever Python object provides the data) when saving or at any other time. pikepdf assumes it will have exclusive access to the input file or input data you give it to, until `pdf.close()` is called.

Saving secure PDFs

To save an encrypted (password protected) PDF, use a *pikepdf.Encryption* object to specify the encryption settings. By default, pikepdf selects the strongest security handler and algorithm (AES-256), but allows full access to modify file contents. A *pikepdf.Permissions* object can be used to specify restrictions.

```
In [17]: no_extracting = pikepdf.Permissions(extract=False)

In [18]: pdf.save('encrypted.pdf', encryption=pikepdf.Encryption(
   ....:       user="user password", owner="owner password", allow=no_extracting
   ....: ))
   ....:
```

Refer to our *security documentation* for more information on user/owner passwords and PDF permissions.

Next steps

Have a look at pikepdf topics that interest you, or jump to our detailed API reference...

1.3.4 PDF split, merge, and document assembly

This section discusses working with PDF pages: splitting, merging, copying, deleting. We're treating pages as a unit, rather than working with the content of individual pages.

Let's continue with `fourpages.pdf` from the *Tutorial*.

```
In [1]: from pikepdf import Pdf

In [2]: pdf = Pdf.open('../tests/resources/fourpages.pdf')
```

Note: In some parts of the documentation we skip closing `Pdf` objects for brevity. In production code, you should open them in a `with` block or explicitly close them.

Split a PDF into single page PDFs

All we need are new PDFs to hold the destination pages.

```
In [3]: pdf = Pdf.open('../tests/resources/fourpages.pdf')

In [4]: for n, page in enumerate(pdf.pages):
   ...:     dst = Pdf.new()
   ...:     dst.pages.append(page)
   ...:     dst.save(f'{n:02d}.pdf')
   ...:
```

Note: This example will transfer data associated with each page, so that every page stands on its own. It will *not* transfer some metadata associated with the PDF as a whole, such as the list of bookmarks.

Merge (concatenate) PDF from several PDFs

We create an empty `Pdf` which will be the container for all the others.

```
In [5]: from glob import glob

In [6]: pdf = Pdf.new()

In [7]: for file in glob('*.pdf'):
   ...:     src = Pdf.open(file)
   ...:     pdf.pages.extend(src.pages)
   ...:

In [8]: pdf.save('merged.pdf')
```

This code sample is enough to merge most PDFs, but there are some things it does not do that a more sophisticated function might do. One could call *pikepdf.Pdf.remove_unreferenced_resources()* to remove unreferenced objects from the pages' /Resources dictionaries. It may also be necessary to chose the most recent version of all source PDFs. Here is a more sophisticated example:

```
In [9]: from glob import glob

In [10]: pdf = Pdf.new()

In [11]: version = pdf.pdf_version

In [12]: for file in glob('*.pdf'):
   ....:     src = Pdf.open(file)
   ....:     version = max(version, src.pdf_version)
   ....:     pdf.pages.extend(src.pages)
   ....:

In [13]: pdf.remove_unreferenced_resources()

In [14]: pdf.save('merged.pdf', min_version=version)
```

This improved example would still leave metadata blank. It's up to you to decide how to combine metadata from multiple PDFs.

Reversing the order of pages

Suppose the file was scanned backwards. We can easily reverse it in place - maybe it was scanned backwards, a common problem with automatic document scanners.

```
In [15]: pdf.pages.reverse()
```

```
In [16]: pdf
Out[16]: <pikepdf.Pdf description='../tests/resources/fourpages.pdf'>
```

Pretty nice, isn't it? But the pages in this file already were in correct order, so let's put them back.

```
In [17]: pdf.pages.reverse()
```

Copying pages from other PDFs

Now, let's add some content from another file. Because pdf.pages behaves like a list, we can use pages.extend() on another file's pages.

```
In [18]: pdf = Pdf.open('../tests/resources/fourpages.pdf')

In [19]: appendix = Pdf.open('../tests/resources/sandwich.pdf')

In [20]: pdf.pages.extend(appendix.pages)
```

We can use pages.insert() to insert into one of more pages into a specific position, bumping everything else ahead.

Copying pages between Pdf objects will create a shallow copy of the source page within the target Pdf, rather than the typical Python behavior of creating a reference. Therefore modifying pdf.pages[-1] will not affect

`appendix.pages[0]`. (Normally, assigning objects between Python lists creates a reference, so that the two objects are identical, `list[0] is list[1]`.)

```
In [21]: graph = Pdf.open('../tests/resources/graph.pdf')

In [22]: pdf.pages.insert(1, graph.pages[0])

In [23]: len(pdf.pages)
Out[23]: 6
```

We can also replace specific pages with assignment (or slicing).

```
In [24]: congress = Pdf.open('../tests/resources/congress.pdf')

In [25]: pdf.pages[2].objgen
Out[25]: (4, 0)

In [26]: pdf.pages[2] = congress.pages[0]

In [27]: pdf.pages[2].objgen
Out[27]: (33, 0)
```

The method above will break any indirect references (such as table of contents entries and hyperlinks) within `pdf` to `pdf.pages[2]`. Perhaps that is the behavior you want, if the replacement means those references are no longer valid. This is shown by the change in *pikepdf.Object.objgen*.

Emplacing pages

To preserve indirect references, use *pikepdf.Object.emplace()*, which will (conceptually) delete all of the content of target and replace it with the content of source, thus preserving indirect references to the page. (Think of this as demolishing the interior of a house, but keeping it at the same address.)

```
In [28]: pdf = Pdf.open('../tests/resources/fourpages.pdf')

In [29]: congress = Pdf.open('../tests/resources/congress.pdf')

In [30]: pdf.pages[2].objgen
Out[30]: (5, 0)

In [31]: pdf.pages.append(congress.pages[0])  # Transfer page to new pdf

In [32]: pdf.pages[2].emplace(pdf.pages[-1])

In [33]: del pdf.pages[-1]  # Remove donor page

In [34]: pdf.pages[2].objgen
Out[34]: (5, 0)
```

Copying pages within a PDF

As you may have guessed, we can assign pages to copy them within a `Pdf`:

```
In [35]: pdf = Pdf.open('../tests/resources/fourpages.pdf')

In [36]: pdf.pages[3] = pdf.pages[0]  # The last shall be made first
```

As above, copying a page creates a shallow copy rather than a Python object reference.

Also as above *`pikepdf.Object.emplace()`* can be used to create a copy that preserves the functionality of indirect references within the PDF.

Using counting numbers

Because PDF pages are usually numbered in counting numbers (1, 2, 3...), pikepdf provides a convenience accessor `.p()` that uses counting numbers:

```
In [37]: pdf.pages.p(1)         # The first page in the document

In [38]: pdf.pages[0]           # Also the first page in the document

In [39]: pdf.pages.remove(p=1)    # Remove first page in the document
```

To avoid confusion, the `.p()` accessor does not accept Python slices, and `.p(0)` raises an exception. It is also not possible to delete using it.

PDFs may define their own numbering scheme or different numberings for different sections, such as using Roman numerals for an introductory section. `.pages` does not look up this information.

Accessing page labels

If a PDF defines custom page labels, such as a typical report with preface material beginning with Roman numerals (i, ii, iii...), body using Arabic numerals (1, 2, 3...), and an appendix using some other convention (A-1, A-2, ...), you can look up the page label using the *`pikepdf.Page`* helper class, as follows:

```
In [40]: Page(pdf.pages[1]).label
Out[40]: 'i'
```

There is currently no API to help with modifying the `pdf.Root.PageLabels` data structure, which contains the label definitions.

Pages information from Root

> **Warning:** It's possible to obtain page information through *`pikepdf.Pdf.Root`* object but **not recommended**. (In PDF parlance, this is the `/Root` object).
>
> The internal consistency of the various `/Page` and `/Pages` is not guaranteed when accessed in this manner, and in some PDFs the data structure for these is fairly complex. Use the `.pages` interface instead.

1.3.5 Working with pages

This section details with how to view and edit the contents of a page.

pikepdf is not an ideal tool for producing new PDFs from scratch – and there are many good tools for that, as mentioned elsewhere. pikepdf is better at inspecting, editing and transforming existing PDFs.

Page objects in PDFs are dictionaries.

```
In [1]: from pikepdf import Pdf, Page

In [2]: example = Pdf.open('../tests/resources/congress.pdf')

In [3]: pageobj1 = example.pages[0]

In [4]: pageobj1
Out[4]:
<pikepdf.Dictionary(Type="/Page")({
  "/Contents": pikepdf.Stream(stream_dict={
      "/Length": 50
    }, data=<...>),
  "/MediaBox": [ 0, 0, 200, 304 ],
  "/Parent": <reference to /Pages>,
  "/Resources": {
    "/XObject": {
      "/Im0": pikepdf.Stream(stream_dict={
          "/BitsPerComponent": 8,
          "/ColorSpace": "/DeviceRGB",
          "/Filter": [ "/DCTDecode" ],
          "/Height": 1520,
          "/Length": 192956,
          "/Subtype": "/Image",
          "/Type": "/XObject",
          "/Width": 1000
        }, data=<...>)
    }
  },
  "/Type": "/Page"
})>
```

The page's `/Contents` key contains instructions for drawing the page content. This is a *content stream*, which is a stream object that follows special rules.

Also attached to this page is a `/Resources` dictionary, which contains a single XObject image. The image is compressed with the `/DCTDecode` filter, meaning it is encoded with the DCT (discrete cosine transform), so it is a JPEG. pikepdf has special APIs for *working with images*.

The `/MediaBox` describes the bounding box of the page in PDF pt units (1/72" or 0.35 mm).

You *can* access the page dictionary data structure directly, but it's fairly complicated. There are a number of rules, optional values and implied values. It's easier to use page helpers, which ensure that the page is modified in a semantically correct manner.

Page helpers

pikepdf provides a helper class, *pikepdf.Page*, which provides higher-level functions to manipulate pages than the standard page dictionary used in the previous examples.

Currently pikepdf does not automatically return helper classes. You must initialize them. In a future release, it will return them automatically.

```
In [5]: from pikepdf import Pdf, Page

In [6]: page = Page(pageobj1)

In [7]: page.trimbox
Out[7]: pikepdf.Array([ 0, 0, 200, 304 ])
```

One advantage of page helpers is that they resolve implicit information. For example, `page.trimbox` will return an appropriate trim box for this page, which in this case is equal to the media box.

1.3.6 Object model

This section covers the object model pikepdf uses in more detail.

A *pikepdf.Object* is a Python wrapper around a C++ `QPDFObjectHandle` which, as the name suggests, is a handle (or pointer) to a data structure in memory, or possibly a reference to data that exists in a file. Importantly, an object can be a scalar quantity (like a string) or a compound quantity (like a list or dict, that contains other objects). The fact that the C++ class involved here is an object *handle* is an implementation detail; it shouldn't matter for a pikepdf user.

The simplest types in PDFs are directly represented as Python types: `int`, `bool`, and `None` stand for PDF integers, booleans and the "null". `Decimal` is used for floating point numbers in PDFs. If a value in a PDF is assigned a Python `float`, pikepdf will convert it to `Decimal`.

Types that are not directly convertible to Python are represented as *pikepdf.Object*, a compound object that offers a superset of possible methods, some of which only if the underlying type is suitable. Use the EAFP (easier to ask forgiveness than permission) idiom, or `isinstance` to determine the type more precisely. This partly reflects the fact that the PDF specification allows many data fields to be one of several types.

For convenience, the `repr()` of a `pikepdf.Object` will display a Python expression that replicates the existing object (when possible), so it will say:

```
>>> catalog_name = pdf.Root.Type
pikepdf.Name("/Catalog")
>>> isinstance(catalog_name, pikepdf.Name)
True
>>> isinstance(catalog_name, pikepdf.Object)
True
```

Making PDF objects

You may construct a new object with one of the classes:

- *pikepdf.Array*
- *pikepdf.Dictionary*
- *pikepdf.Name* - the type used for keys in PDF Dictionary objects
- *pikepdf.String* - a text string (treated as `bytes` and `str` depending on context)

These may be thought of as subclasses of `pikepdf.Object`. (Internally they **are** `pikepdf.Object`.)

There are a few other classes for special PDF objects that don't map to Python as neatly.

- `pikepdf.Operator` - a special object involved in processing content streams
- `pikepdf.Stream` - a special object similar to a `Dictionary` with binary data attached
- `pikepdf.InlineImage` - an image that is embedded in content streams

The great news is that it's often unnecessary to construct `pikepdf.Object` objects when working with pikepdf. Python types are transparently *converted* to the appropriate pikepdf object when passed to pikepdf APIs – when possible. However, pikepdf sends `pikepdf.Object` types back to Python on return calls, in most cases, because pikepdf needs to keep track of objects that came from PDFs originally.

Object lifecycle and memory management

As mentioned above, a *pikepdf.Object* may reference data that is lazily loaded from its source *pikepdf.Pdf*. Closing the *Pdf* with *pikepdf.Pdf.close()* will invalidate some objects, depending on whether or not the data was loaded, and other implementation details that may change. Generally speaking, a *pikepdf.Pdf* should be held open until it is no longer needed, and objects that were derived from it may or may not be usable after it is closed.

Simple objects (booleans, integers, decimals, `None`) are copied directly to Python as pure Python objects.

For PDF stream objects, use *pikepdf.Object.read_bytes()* to obtain a copy of the object as pure bytes data, if this information is required after closing a PDF.

When objects are copied from one *pikepdf.Pdf* to another, the underlying data is copied immediately into the target. As such it is possible to merge hundreds of *Pdf* into one, keeping only a single source at a time and the target file open.

1.3.7 Stream objects

A *pikepdf.Stream* object works like a PDF dictionary with some encoded bytes attached. The dictionary is metadata that describes how the stream is encoded. PDF can, and regularly does, use a variety of encoding filters. A stream can be encoded with one or more filters. Images are a type of stream object.

This is not the same type of object as Python's file-like I/O objects, which are sometimes called streams.

Most of the interesting content in a PDF (images and content streams) are inside stream objects.

Because the PDF specification unfortunately defines several terms that involve the word *stream*, let's attempt to clarify:

stream object A PDF object that contains binary data and a metadata dictionary to describes it, represented as *pikepdf.Stream*. In HTML this is equivalent to a `<object>` tag with attributes and data.

object stream A stream object (not a typo, an object stream really is a type of stream object) in a PDF that contains a number of other objects in a PDF, grouped together for better compression. In pikepdf there is an option to save

PDFs with this feature enabled to improve compression. Otherwise, this is just a detail about how PDF files are encoded.

content stream A stream object that contains some instructions to draw graphics and text on a page, or inside a Form XObject. In HTML this is equivalent to the HTML file itself. Content streams do not cross pages.

Form XObject A group of images, text and drawing commands that can be rendered elsewhere in a PDF as a group. This is often used when a group of objects are needed at different scales or on multiple pages. In HTML this is like an `<svg>`. It is not a fillable PDF form (although a fillable PDF form could involve Form XObjects).

(Python) stream A stream is another name for a file object or file-like object, as described in the Python `io` module.

Reading stream objects

Fortunately, `pikepdf.Stream.read_bytes()` will apply all filters and decode the uncompressed bytes, or throw an error if this is not possible. `pikepdf.Stream.read_raw_bytes()` provides access to the compressed bytes.

Three types of stream object are particularly noteworthy: content streams, which describe the order of drawing operators; images; and XMP metadata. pikepdf provides helper functions for working with these types of streams.

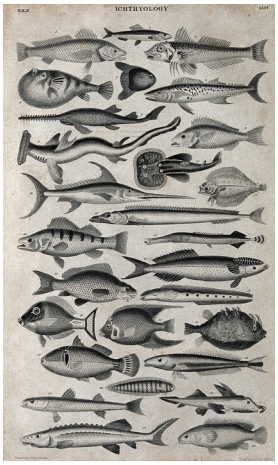


Fig. 7: When it comes to taxonomy, software developers have it easy.

Reading stream objects as a Python I/O streams

You were warned about terminology.

In the interesting of preversing our remaining sanity, you cannot access a stream object as a file-like object directly.

To efficiently access a `pikepdf.Stream` as a Python file object, you may do:

```
pdf.pages[0].Contents.page_contents_coalesce()
filelike_object␣
↪= BytesIO(pdf.pages[0].Contents.get_stream_buffer())
```

1.3.8 Working with content streams

A content stream is a stream object associated with either a page or a Form XObject that describes where and how to draw images, vectors, and text. (These PDF streams have nothing to do with Python I/O streams.)

Content streams are binary data that can be thought of as a list of operators and zero or more operands. Operands are given first, followed by the operator. It is a stack-based language, loosely based on PostScript. (It's not actually PostScript, but sometimes well-meaning people mistakenly say that it is!) Like HTML, it has a precise grammar, and also like (pure) HTML, it has no loops, conditionals or variables.

A typical example is as follows (with additional whitespace and PostScript-style `%`-comments):

```
q                   % 1. Push graphics stack.
100 0 0 100 0 0 cm  % 2.
↪ The 6 numbers are the operands, followed by cm operator.
```

(continues on next page)

(continued from previous page)

```
↪  %    This configures the current transformation matrix.
/Image1 Do
↪   % 3. Draw the object named /Image1 from the /Resources
                    %    dictionary.
Q                   % 4. Pop graphics stack.
```

The pattern `q, cm, <drawing commands>, Q` is extremely common. The drawing commands may recurse with another `q, cm, ..., Q`.

pikepdf provides a C++ optimized content stream parser and a filter. The parser is best used for reading and interpreting content streams; the filter is better for low level editing.

How content streams draw images

This example prints a typical content stream from a real file, which like the contrived example above, displays an actual image.

```
In [1]: with
↪pikepdf.open("../tests/resources/congress.pdf") as pdf:
   ...:     page = pdf.pages[0]
   ...:     commands = []
   ...:     for
↪operands, operator in pikepdf.parse_content_stream(page):
   ...:
↪       print(f"Operands {operands}, operator {operator}")
   ...:         commands.append([operands, operator])
   ...:
Operands [], operator q
Operands [Decimal('200.0000'), 0, 0, Decimal('304.0000
↪'), Decimal('0.0000'), Decimal('0.0000')], operator cm
Operands [pikepdf.Name("/Im0")], operator Do
Operands [], operator Q
```

PDF content streams are stateful. The commands `q`, `cm` and `Q` manipulate the current transform matrix (CTM) which describes where we will draw next. *In most cases* you have to track every manipulation of the CTM to figure out what will happen, even to answer a question like, "where will this image be drawn, and how big will it be?"

But *in this simple case*, we can read the matrix directly. The decimal numbers 200.0 and 304.0 establish the width and height at which the image should be drawn, in PDF points (1/72" or about 0.35 mm). The pixel dimensions of the image have no effect. If we substituted that image for another, the new image would be drawn in the same location on the page, painted into the 200 × 304 rectangle regardless of its pixel dimensions.

Editing a content stream

Let's continue with the file above and center the image on the page, and reduce its size by 50%. Because we can! For that, we need to rewrite the second command in the content stream.

We take the original matrix (`original`) and then translated it to the center of this page. We know that the full page image is 200 × 304 PDF points, so we translate by one half on each axis: `.translated(200/2, 304/2)`. Then we scale by 0.5: `.scaled(0.5, 0.5)`.

```
In [2]: original = pikepdf.PdfMatrix(commands[1][0])  # command cm, operands

In [3]: new_matrix = original.translated(200/2, 304/2).scaled(0.5, 0.5)

In [4]: new_matrix
Out[4]: pikepdf.Matrix(((100.0, 0.0, 0.0), (0.0, 152.0, 0.0), (50.0, 76.0, 1.0)))
```

On an important note, the PDF coordinate system is nailed to the **bottom left** corner of the page, and on y-axis, **up is positive**. That is, the coordinate system is more like the first quadrant of a Cartesian graph than the **down is positive** convention normally used in pixel graphics:

Thus the command `.translated(200/2, 304/2)` is translated from the origin at the bottom left, (0, 0), to the right by 100 units, and up 152 units. (Some PDF programs insert a command to "flip" the coordinate system, by translating to the top left corner and scaling by (1, -1).)

After calculating our new matrix, we need to insert it back into the parsed content stream, "unparse" it to binary data, and replace the old content stream.

```
In [5]: commands[1][0] = pikepdf.Array([*new_matrix.shorthand])

In [6]: new_content_stream = pikepdf.unparse_content_stream(commands)

In [7]: new_content_stream
Out[7]: b' q\n100 0 0 152 50 76 cm\n/Im0 Do\n Q'

In [8]: page.Contents = pdf.make_stream(new_content_stream)

# You could save the file here to see it
# pdf.save(...)
```

Note: To rotate an image, first translate it so that the image is centered at (0, 0), rotate then apply the rotate, then translate it to its new center position. This is because rotations occur around (0, 0).

Note: In this illustration, the page's MediaBox is located at (0, 0) for simplicity. The MediaBox can be offset from the origin, and code that edits content streams may need to account for this relatively condition.

Editing content streams robustly

The stateful nature of PDF content streams makes editing them complicated. Edits like the example above will work when the input file is known to have a fixed structure (that is, the state at the time of editing is known). You can always prepend content to the top of the content stream, since the initial state is known. And you can often append content to the end the stream, since the final state is predictable if every `q` (push state) has a matching `Q` (pop state).

Otherwise, you must track the graphics state and maintain a stack of states.

Most applications will end up parsing the content stream into a higher level representation that is easier edit and then serializing it back, totally rewriting the content stream. Content streams should be thought of as an output format.

Extracting text from PDFs

If you guessed that the content streams were the place to look for text inside a PDF – you'd be correct. Unfortunately, extracting the text is fairly difficult because content stream actually specifies as a font and glyph numbers to use. Sometimes, there is a 1:1 transparent mapping between Unicode numbers and glyph numbers, and dump of the content stream will show the text. In general, you cannot rely on there being a transparent mapping; in fact, it is perfectly legal for a font to specify no Unicode mapping at all, or to use an unconventional mapping (when a PDF contains a subsetted font for example).

We strongly recommend against trying to scrape text from the content stream.

pikepdf does not currently implement text extraction. We recommend pdfminer.six, a read-only text extraction tool. If you wish to write PDFs containing text, consider reportlab.

1.3.9 Working with images

PDFs embed images as binary stream objects within the PDF's data stream. The stream object's dictionary describes properties of the image such as its dimensions and color space. The same image may be drawn multiple times on multiple pages, at different scales and positions.

In some cases such as JPEG2000, the standard file format of the image is used verbatim, even when the file format contains headers and information that is repeated in the stream dictionary. In other cases such as for PNG-style encoding, the image file format is not used directly.

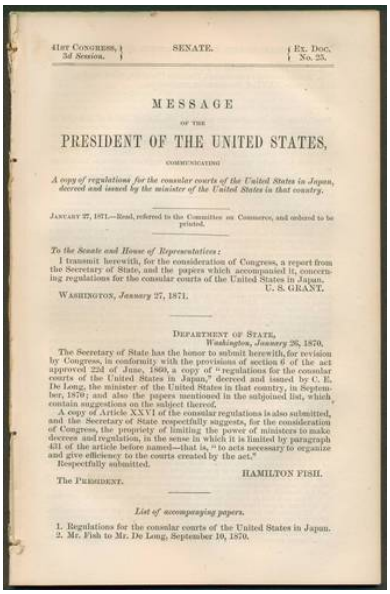
pikepdf currently has no facility to embed new images into PDFs. We recommend img2pdf instead, because it does the job so well. pikepdf instead allows for image inspection and lossless/transcode free (where possible) "pdf2img".

Playing with images

pikepdf provides a helper class *PdfImage* for manipulating images in a PDF.
The helper class helps manage the complexity of the image dictionaries.

```
In [1]: from pikepdf import Pdf, PdfImage, Name

In [2]:␣
↪example = Pdf.open('../tests/resources/congress.pdf')

In [3]: page1 = example.pages[0]

In [4]: list(page1.images.keys())
Out[4]: ['/Im0']

In [5]: rawimage␣
↪= page1.images['/Im0']  # The raw object/dictionary

In [6]: pdfimage = PdfImage(rawimage)

In [7]: type(pdfimage)
Out[7]: pikepdf.models.image.PdfImage
```

In Jupyter (or IPython with a suitable backend) the image will be displayed.



41st CONGRESS, 3d Session. SENATE. Ex. Doc. No. 25.

MESSAGE
OF THE
PRESIDENT OF THE UNITED STATES,
COMMUNICATING
A copy of regulations for the consular courts of the United States in Japan, decreed and issued by the minister of the United States in that country.

JANUARY 27, 1871.—Read, referred to the Committee on Commerce, and ordered to be printed.

To the Senate and House of Representatives:
I transmit herewith, for the consideration of Congress, a report from the Secretary of State, and the papers which accompanied it, concerning regulations for the consular courts of the United States in Japan.
U. S. GRANT.
WASHINGTON, *January* 27, 1871.

DEPARTMENT OF STATE,
Washington, January 26, 1870.
The Secretary of State has the honor to submit herewith, for revision by Congress, in conformity with the provisions of section 6 of the act approved 22d of June, 1860, a copy of "regulations for the consular courts of the United States in Japan," decreed and issued by C. E. De Long, the minister of the United States in that country, in September, 1870; and also the papers mentioned in the subjoined list, which contain suggestions on the subject thereof.
A copy of Article XXVI of the consular regulations is also submitted, and the Secretary of State respectfully suggests, for the consideration of Congress, the propriety of limiting the power of ministers to make decrees and regulation, in the sense in which it is limited by paragraph 431 of the article before named—that is, "to acts necessary to organize and give efficiency to the courts created by the act."
Respectfully submitted.
HAMILTON FISH.
The PRESIDENT.

List of accompanying papers.
1. Regulations for the consular courts of the United States in Japan.
2. Mr. Fish to Mr. De Long, September 10, 1870.

You can also inspect the properties of the image. The parameters are similar to Pillow's.

```
In [8]: pdfimage.colorspace
Out[8]: '/DeviceRGB'

In [9]: pdfimage.width, pdfimage.height
Out[9]: (1000, 1520)
```

Note: `.width` and `.height` are the resolution of the image in pixels, not the size of the image in page coordinates. The size of the image in page coordinates is determined by the content stream.

Extracting images

Extracting images is straightforward. *extract_to()* will extract images to a specified file prefix. The extension is determined while extracting and appended to the filename. Where possible, `extract_to` writes compressed data directly to the stream without transcoding. (Transcoding lossy formats like JPEG can reduce their quality.)

```
In [10]: pdfimage.extract_to(fileprefix='image')
Out[10]: 'image.jpg'
```

It also possible to extract to a writable Python stream using `.extract_to(stream=...`).

You can also retrieve the image as a Pillow image (this will transcode):

```
In [11]: type(pdfimage.as_pil_image())
Out[11]: PIL.JpegImagePlugin.JpegImageFile
```

Another way to view the image is using Pillow's `Image.show()` method.

Not all image types can be extracted. Also, some PDFs describe an image with a mask, with transparency effects. pikepdf can only extract the images themselves, not rasterize them exactly as they would appear in a PDF viewer. In the vast majority of cases, however, the image can be extracted as it appears.

Note: This simple example PDF displays a single full page image. Some PDF creators will paint a page using multiple images, and features such as layers, transparency and image masks. Accessing the first image on a page is like an HTML parser that scans for the first `<img src="">` tag it finds. A lot more could be happening. There can be multiple images drawn multiple times on a page, vector art, overdrawing, masking, and transparency. A set of resources can be grouped together in a "Form XObject" (not to be confused with a PDF Form), and drawn at all once. Images can be referenced by multiple pages.

Replacing an image

In this example we extract an image and replace it with a grayscale equivalent.

```
In [12]: import zlib

In [13]: rawimage = pdfimage.obj

In [14]: pillowimage = pdfimage.as_pil_image()

In [15]: grayscale = pillowimage.convert('L')

In [16]: grayscale = grayscale.resize((32, 32))

In [17]: rawimage.write(zlib.compress(grayscale.
 ↪tobytes()), filter=Name("/FlateDecode"))

In [18]: rawimage.ColorSpace = Name("/DeviceGray")

In [19]: rawimage.Width, rawimage.Height = 32, 32
```

Notes on this example:

- It is generally possible to use `zlib.compress()` to generate compressed image data, although this is not as efficient as using a program that knows it is preparing a PDF.
- In general we can resize an image to any scale. The PDF content stream specifies where to draw an image and at what scale.
- This example would replace all occurrences of the image if it were used multiple times in a PDF.

Removing an image

The easy way to remove an image is to replace it with a 1x1 pixel transparent image. A transparent image can be created by setting the `/ImageMask` to true.

Note that, if an image is referenced on multiple pages, this procedure only updates the occurrence on one page. If all references to the image are deleted, it should not be included in the output file.

```
In [20]:␣
↪pdf = pikepdf.open('../tests/resources/sandwich.pdf')

In [21]: page = pdf.pages[0]

In [22]: image_name, image = next(page.images.items())

In [23]: new_image = pdf.make_stream(b'\xff')

In [24]: new_image.Width, new_image.Height = 1, 1

In [25]: new_image.BitsPerComponent = 1

In [26]: new_image.ImageMask = True

In [27]: new_image.Decode = [0, 1]

In [28]: page.Resources.XObject[image_name] = new_image
```

1.3.10 Character encoding

There are three hard problems in computer science:
1) Converting from PDF,
2) Converting to PDF, and
3) OO

—Marseille Folog

In most circumstances, pikepdf performs appropriate encodings and decodings on its own, or returns *pikepdf.String* if it is not clear whether to present data as a string or binary data.

`str(pikepdf.String)` is performed by inspecting the binary data. If the binary data begins with a UTF-16 byte order mark, then the data is interpreted as UTF-16 and returned as a Python `str`. Otherwise, the data is returned as a Python `str`, if the binary data will be interpreted as PDFDocEncoding and

decoded to `str`. Again, in most cases this is correct behavior and will operate transparently.

Some functions are available in circumstances where it is necessary to force a particular conversion.

PDFDocEncoding

The PDF specification defines PDFDocEncoding, a character encoding used only in PDFs. This encoding matches ASCII for code points 32 through 126 (0x20 to 0x7e). At all other code points, it is not ASCII and cannot be treated as equivalent. If you look at a PDF in a binary file viewer (hex editor), a string surrounded by parentheses such as `(Hello World)` is usually using PDFDocEncoding.

When pikepdf is imported, it automatically registers `"pdfdoc"` as a codec with the standard library, so that it may be used in string and byte conversions.

```
"•".encode('pdfdoc') == b'\x81'
```

Other codecs

Two other codecs are commonly used in PDFs, but they are already part of the standard library.

WinAnsiEncoding is identical Windows Code Page 1252, and may be converted using the `"cp1252"` codec.

MacRomanEncoding may be converted using the `"macroman"` codec.

1.3.11 PDF Metadata

PDF has two different types of metadata: XMP metadata, and DocumentInfo, which is deprecated but still relevant. For backward compatibility, both should contain the same content. pikepdf provides a convenient interface that coordinates edits to both, but is limited to the most common metadata features.

XMP (Extensible Metadata Platform) Metadata is a metadata specification in XML format that is used many formats other than PDF. For full information on XMP, see Adobe's XMP Developer Center. The XMP Specification also provides useful information.

pikepdf can read compound metadata quantities, but can only modify scalars. For more complex changes consider using the `python-xmp-toolkit` library and its libexempi dependency; but note that it is not capable of synchronizing changes to the older DocumentInfo metadata.

Automatic metadata updates

By default pikepdf will create a XMP metadata block and set `pdf:PDFVersion` to a value that matches the PDF version declared elsewhere in the PDF, whenever a PDF is saved. To suppress this behavior, save with `pdf.save(..., fix_metadata_version=False)`.

Also by default, `Pdf.open_metadata()` will synchronize the XMP metadata with the older document information dictionary. This behavior can also be adjusted using keyword arguments.

Accessing metadata

The XMP metadata stream is attached the PDF's root object, but to simplify management of this, use *pikepdf.Pdf.open_metadata()*. The returned *pikepdf.models.PdfMetadata* object may be used for reading, or entered with a `with` block to modify and commit changes. If you use this interface, pikepdf will synchronize changes to new and old metadata.

A PDF must still be saved after metadata is changed.

```
In [1]:␣
↪pdf = pikepdf.open('../tests/resources/sandwich.pdf')

In [2]: meta = pdf.open_metadata()

In [3]: meta['xmp:CreatorTool']
Out[3]: 'ocrmypdf 5.3.3 / Tesseract OCR-PDF 3.05.01'
```

If no XMP metadata exists, an empty XMP metadata container will be created.

Open metadata in a `with` block to open it for editing. When the block is exited, changes are committed (updating XMP and the Document Info dictionary) and attached to the PDF object. The PDF must still be saved. If an exception occurs in the block, changes are discarded.

```
In [4]: with pdf.open_metadata() as meta:
   ...:     meta['dc:title'] = "Let's change the title"
   ...:
```

The list of available metadata fields may be found in the XMP Specification.

Removing metadata items

After opening metadata, use `del meta['dc:title']` to delete a metadata entry.

To remove all of a PDF's metadata records, don't use `pdf.open_metadata`. Instead, use `del pdf.Root.Metadata` and `del pdf.docinfo` to remove the XMP and document info metadata, respectively.

Checking PDF/A conformance

The metadata interface can also test if a file **claims** to be conformant to the PDF/A specification.

```
In [5]: pdf = pikepdf.open('../tests/
↪resources/veraPDF test suite 6-2-10-t02-pass-a.pdf')

In [6]: meta = pdf.open_metadata()

In [7]: meta.pdfa_status
Out[7]: '1B'
```

Note: Note that this property merely *tests* if the file claims to be conformant to the PDF/A standard. Use a tool such as veraPDF (official tool), or third party web services such as PDFEN or 3-HEIGHTS™ PDF VALIDATOR to verify conformance.

Notice for application developers

If you are using pikepdf to create some kind of PDF application, you should update the fields `xmp:CreatorTool` and `pdf:Producer`. You could, for example, set `xmp:CreatorTool` to your application's name and version, and `pdf:Producer` to pikepdf. Refer to Adobe's documentation to decide what describes the circumstances.

This will help PDF developers identify the application that generated a particular PDF and is valuable debugging information.

Low-level XMP metadata access

You can read the raw XMP metadata if desired. For example, one could extract it and edit it using the full featured `python-xmp-toolkit` library.

```
In [8]: xmp = pdf.Root.Metadata.read_bytes()

In [9]: type(xmp)
Out[9]: bytes

In [10]: print(xmp.decode())
<?xpacket begin='' id='W5M0MpCehiHzreSzNTczkc9d'?>
<x:xmpmeta xmlns:x="adobe:ns:meta/">
  <rdf:RDF␣
↪xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description␣
↪xmlns:dc="http://purl.org/dc/elements/1.1/" rdf:about="">
      <dc:creator>
        <rdf:Seq>
          <rdf:li>veraPDF Consortium</rdf:li>
        </rdf:Seq>
      </dc:creator>
    </rdf:Description>
    <rdf:Description xmlns:xmp="http://ns.adobe.
↪com/xap/1.0/" rdf:about="" xmp:CreatorTool="veraPDF␣
↪Test Builder" xmp:CreateDate="2015-03-10T17:19:21+01:00
↪" xmp:ModifyDate="2015-03-10T17:19:21+01:00"/>
```

(continues on next page)

(continued from previous page)

```
 ↪ <rdf:Description xmlns:pdf="http://ns.adobe.com/pdf/1.3/
↪" rdf:about="" pdf:Producer="veraPDF Test Builder 1.0 "/>
    <rdf:Description␣
↪xmlns:pdfaid="http://www.aiim.org/pdfa/ns/id/
↪" rdf:about="" pdfaid:part="1" pdfaid:conformance="B"/>
  </rdf:RDF>
</x:xmpmeta>
<?xpacket end='w'?>
```

Editing XMP with a generic XML library is probably not worth the trouble; the semantics are fairly complex.

Warning: Manually changes to XMP stream object will not be synchronized with live PdfMetadata object or the DocumentInfo block.

The Document Info dictionary

The Document Info block is an older, now deprecated object in which metadata may be stored. The Document Info is not attached to the /Root object. It may be accessed using the `.docinfo` property. If no Document Info exists, touching the `.docinfo` will properly initialize an empty one.

Here is an example of a Document Info block.

```
In [11]:␣
↪pdf = pikepdf.open('../tests/resources/sandwich.pdf')

In [12]: pdf.docinfo
Out[12]:
pikepdf.Dictionary({
  "/CreationDate": "D:20170911132748-07'00'",
  "/Creator": "ocrmypdf 5.3.3 / Tesseract OCR-PDF 3.05.01",
  "/ModDate": "D:20170911132748-07'00'",
  "/Producer": "GPL Ghostscript 9.21"
})
```

It is permitted in pikepdf to directly interact with Document Info as with other PDF dictionaries. However, it is better to use `.open_metadata()` because that interface will apply changes to both XMP and Document Info in a consistent manner.

You may copy from data from a Document Info object in the current PDF or another PDF into XMP metadata using *load_from_docinfo()*.

1.3.12 Outlines

Outlines (sometimes also called *bookmarks*) are shown in a the PDF viewer aside of the page, allowing for navigation within the document.

Creating outlines

Outlines can be created from scratch, e.g. when assembling a set of PDF files into a single document.

The following example adds outline entries referring to the 1st, 3rd and 9th page of an existing PDF.

```
In [1]: from pikepdf import Pdf, OutlineItem

In [2]: pdf = Pdf.open('document.pdf')

In [3]: with pdf.open_outline() as outline:
   ...:     outline.root.extend([
   ...:         # Page counts are zero-based
   ...:         OutlineItem('Section One', 0),
   ...:         OutlineItem('Section Two', 2),
   ...:         OutlineItem('Section Three', 8)
   ...:     ])
   ...:

In [4]: pdf.save('document_with_outline.pdf')
```

Another example, for automatically adding an entry for each file in a merged document:

```
In [5]: from glob import glob

In [6]: pdf = Pdf.new()

In [7]: page_count = 0

In [8]: with pdf.open_outline() as outline:
   ...:     for file in glob('*.pdf'):
   ...:         src = Pdf.open(file)
   ...:         oi = OutlineItem(file, page_count)
   ...:         outline.root.append(oi)
   ...:         page_count += len(src.pages)
   ...:         pdf.pages.extend(src.pages)
   ...:

In [9]: pdf.save('merged.pdf')
```

Editing outlines

Existing outlines can be edited. Entries can be moved and renamed without affecting the targets they refer to.

Destinations

Destinations tell the PDF viewer where to go when navigating through outline items. The simplest case is a reference to a page, together with the page location, e.g. `Fit` (default). However, named destinations can also be assigned.

The PDF specification allows for either use of a destination (`Dest` attribute) or an action (`A` attribute), but not both on the same element. `OutlineItem` elements handle this as follows:

- When creating new outline entries passing in a page number or reference name, the `Dest` attribute is used.
- When editing an existing entry with an assigned action, it is left as-is, unless a `destination` is set. The latter is preferred if both are present.

Creating a more detailed destination with page location:

```
In [10]: oi = OutlineItem('First', 0, 'FitB', top=1000)
```

The above will call `make_page_destination` when saving to a `Pdf` document, roughly equivalent to the following:

```
In [11]: oi.destination␣
↪= make_page_destination(pdf, 0, 'FitB', top=1000)
```

Outline structure

For nesting outlines, add items to the `children` list of another `OutlineItem`.

```
In [12]: with pdf.open_outline() as outline:
   ....:     main_item = OutlineItem('Main', 0)
   ....:     outline.root.append(main_item)
   ....:     main_item.children.append(OutlineItem('A', 1))
   ....:
```

1.3.13 PDF security

Password security

Password security in PDFs is widely supported, including by pikepdf. Unfortunately, its security has limitations and may offer more security theatre than real security, depending on your needs.

Note the following limitations of password security in PDFs:

- anyone with the user password *or* the owner password can open the PDF, extract its contents, and produce a visually identical PDF;

- if the user password is an empty string, everyone has the user password;
- setting a user password and leaving the owner password blank is useless;
- the only thing you can not do if you have the user password and not the owner password is create a new PDF encrypted with the same owner password;
- `pikepdf.Permissions` restrictions depend entirely on the PDF viewer software to enforce the restrictions – libraries like pikepdf can bypass those restrictions;
- cracking PDF passwords is easier than many other forms of cracking because a motivated person has unlimited chances to guess the password on a static file.

  While the AES encryption algorithm is strong, password-protected PDFs have significant practical weaknesses.

  In view of all of this, the most useful option is to set the owner password to a strong password, and the user password to blank. This allows anyone to view the PDF while allowing you to prove that you (or your software's user) generated the PDF by producing the strong owner password.

Unicode in passwords

For widest compatibility, passwords should be composed of only characters in the ASCII character set, since the PDF reference manual is unclear about how non-ASCII passwords are supposed to be encoded. See the documentation on *pikepdf.Pdf.save()* for more details. pikepdf encodes passwords as UTF-8.

PDF content restrictions

If you are developing a PDF application, you should enforce the restrictions in *pikepdf.Permissions*, and not permit people who have only the user password to access restricted content. If the PDF is opened with the owner password, any content may be accessed without enforcing restrictions. *pikepdf.Pdf.user_password_matched* and *pikepdf.Pdf.owner_password_matched* can be used to check which password opened the PDF.

It is up to the application developer to implement the restrictions. pikepdf or any PDF manipulation library can be used to bypass restrictions.

Digital signatures and certificates

PDFs signed with a digital signature can mitigate some of these security issues. pikepdf does not support digital signatures at this time.

1.3.14 Main objects

class pikepdf.**Pdf**
In-memory representation of a PDF

property Root
The /Root object of the PDF.

add_blank_page(*, *page_size=(612.0, 792.0)*)
Add a blank page to this PDF. If pages already exist, the page will be added to the end. Pages may be reordered using `Pdf.pages`.

The caller may add content to the page by modifying its objects after creating it.

Parameters **page_size** (*tuple*) – The size of the page in PDF units (1/72 inch or 0.35mm). Default size is set to a US Letter 8.5" x 11" page.

Return type pikepdf.objects.Object

property allow
Report permissions associated with this PDF.

By default these permissions will be replicated when the PDF is saved. Permissions may also only be changed when a PDF is being saved, and are only available for encrypted PDFs. If a PDF is not encrypted, all operations are reported as allowed.

pikepdf has no way of enforcing permissions.

check()
Check if PDF is well-formed. Similar to `qpdf --check`.

Return type List[str]

check_linearization(*self:* pikepdf.Pdf, *stream: object = sys.stderr*) → bool
Reports information on the PDF's linearization.

Parameters **stream** – A stream to write this information too; must implement `.write()` and `.flush()` method. Defaults to `sys.stderr`.

Returns `True` if the file is correctly linearized, and `False` if the file is linearized but the linearization data contains errors or was incorrectly generated.

Raises **RuntimeError** – If the PDF in question is not linearized at all.

close()
Close a `Pdf` object and release resources acquired by pikepdf.

If pikepdf opened the file handle it will close it (e.g. when opened with a file path). If the caller opened the file for pikepdf, the caller close the file. `with` blocks will call close when exit.

pikepdf lazily loads data from PDFs, so some *pikepdf.Object* may implicitly depend on the *pikepdf.Pdf* being open. This is always the case for *pikepdf.Stream* but can be true for any object. Do not close the *Pdf* object if you might still be accessing content from it.

When an `Object` is copied from one `Pdf` to another, the `Object` is copied into the destination `Pdf` immediately, so after accessing all desired information from the source `Pdf` it may be closed.

> **Caution:** Closing the `Pdf` is currently implemented by resetting it to an empty sentinel. It is currently possible to edit the sentinel as if it were a live object. This behavior should not be relied on and is subject to change.

Return type None

copy_foreign (*self:* pikepdf.Pdf, *h:* pikepdf.Object) → *pikepdf.Object*

Copy an `Object` from a foreign `Pdf` to this one.

This function is used to copy a *`pikepdf.Object`* that is owned by some other `Pdf` into this one. This is performs a deep (recursive) copy and preserves circular references that may exist in the foreign object. It also copies all *`pikepdf.Stream`* objects. Since this may copy a large amount of data, it is not done implicitly. This function does not copy references to pages in the foreign PDF - it stops at page boundaries. Thus, if you use `copy_foreign()` on a table of contents (`/Outlines` dictionary), you may have to update references to pages.

Direct objects, including dictionaries, do not need `copy_foreign()`. pikepdf will automatically convert and construct them.

Note: pikepdf automatically treats incoming pages from a foreign PDF as foreign objects, so *`Pdf.pages`* does not require this treatment.

See also:

QPDF::copyForeignObject

property docinfo

Access the (deprecated) document information dictionary.

The document information dictionary is a brief metadata record that can store some information about the origin of a PDF. It is deprecated and removed in the PDF 2.0 specification (not deprecated from the perspective of pikepdf). Use the `.open_metadata()` API instead, which will edit the modern (and unfortunately, more complicated) XMP metadata object and synchronize changes to the document information dictionary.

This property simplifies access to the actual document information dictionary and ensures that it is created correctly if it needs to be created.

A new, empty dictionary will be created if this property is accessed and dictionary does not exist. (This is to ensure that convenient code like `pdf.docinfo[Name.Title] = "Title"` will work when the dictionary does not exist at all.)

You can delete the document information dictionary by deleting this property, `del pdf.docinfo`. Note that accessing the property after deleting it will re-create with a new, empty dictionary.

property encryption

Report encryption information for this PDF.

Encryption settings may only be changed when a PDF is saved.

property filename

The source filename of an existing PDF, when available.

flatten_annotations (*self:* pikepdf.Pdf, *mode: str = 'all'*) → None
Flattens all PDF annotations into regular PDF content.

Annotations are markup such as review comments, highlights, proofreading marks. User data entered into interactive form fields also counts as an annotation.

When annotations are flattened, they are "burned into" the regular content stream of the document and the fact that they were once annotations is deleted. This can be useful when preparing a document for printing, to ensure annotations are printed, or to finalize a form that should no longer be changed.

Parameters **mode** – One of the strings `'all'`, `'screen'`, `'print'`. If omitted or set to empty, treated as `'all'`. `'screen'` flattens all except those marked with the PDF flag /NoView. `'print'` flattens only those marked for printing.

generate_appearance_streams (*self:* pikepdf.Pdf) → None
Generates appearance streams for AcroForm forms and form fields.

Appearance streams describe exactly how annotations and form fields should appear to the user. If omitted, the PDF viewer is free to render the annotations and form fields according to its own settings, as needed.

For every form field in the document, this generates appearance streams, subject to the limitations of QPDF's ability to create appearance streams.

When invoked, this method will modify the `Pdf` in memory. It may be best to do this after the `Pdf` is opened, or before it is saved, because it may modify objects that the user does not expect to be modified.

See: https://github.com/qpdf/qpdf/blob/bf6b9ba1c681a6fac6d585c6262fb2778d4bb9d2/include/qpdf/QPDFFormFieldObjectHelper.hh#L216

get_object (**args*, ***kwargs*)
Overloaded function.

1. get_object(self: pikepdf.Pdf, objgen: Tuple[int, int]) -> pikepdf.Object

   Look up an object by ID and generation number

   Return type: pikepdf.Object

2. get_object(self: pikepdf.Pdf, objid: int, gen: int) -> pikepdf.Object

   Look up an object by ID and generation number

   Return type: pikepdf.Object

get_warnings (*self:* pikepdf.Pdf) → list

property is_linearized
Returns True if the PDF is linearized.

Specifically returns True iff the file starts with a linearization parameter dictionary. Does no additional validation.

make_indirect (**args*, ***kwargs*)
Overloaded function.

1. make_indirect(self: pikepdf.Pdf, h: pikepdf.Object) -> pikepdf.Object

   Attach an object to the Pdf as an indirect object

Direct objects appear inline in the binary encoding of the PDF. Indirect objects appear inline as references (in English, "look up object 4 generation 0") and then read from another location in the file. The PDF specification requires that certain objects are indirect - consult the PDF specification to confirm.

Generally a resource that is shared should be attached as an indirect object. *pikepdf.Stream* objects are always indirect, and creating them will automatically attach it to the Pdf.

See Also: `pikepdf.Object.is_indirect()`

Return type: pikepdf.Object

2. make_indirect(self: pikepdf.Pdf, obj: object) -> pikepdf.Object

   Encode a Python object and attach to this Pdf as an indirect object

Return type: pikepdf.Object

make_stream(*data*, *d=None*, ***kwargs*)
Create a new pikepdf.Stream object that is attached to this PDF.

See: *pikepdf.Stream.__new__()*

Parameters **data** (*bytes*) –

Return type pikepdf.objects.Stream

static new() → *pikepdf.Pdf*
Create a new empty PDF from stratch.

property objects
Return an iterable list of all objects in the PDF.

After deleting content from a PDF such as pages, objects related to that page, such as images on the page, may still be present.

Retun type: pikepdf._ObjectList

open(*password=''*, *hex_password=False*, *ignore_xref_streams=False*, *suppress_warnings=True*, *attempt_recovery=True*, *inherit_page_attributes=True*, *access_mode=<AccessMode.default: 0>*, *allow_overwriting_input=False*)
Open an existing file at *filename_or_stream*.

If *filename_or_stream* is path-like, the file will be opened for reading. The file should not be modified by another process while it is open in pikepdf, or undefined behavior may occur. This is because the file may be lazily loaded. Despite this restriction, pikepdf does not try to use any OS services to obtain an exclusive lock on the file. Some applications may want to attempt this or copy the file to a temporary location before editing. This behaviour changes if *allow_overwriting_input* is set: the whole file is then read and copied to memory, so that pikepdf can overwrite it when calling `.save()`.

When this function is called with a stream-like object, you must ensure that the data it returns cannot be modified, or undefined behavior will occur.

Any changes to the file must be persisted by using `.save()`.

If *filename_or_stream* has `.read()` and `.seek()` methods, the file will be accessed as a readable binary stream. pikepdf will read the entire stream into a private buffer.

`.open()` may be used in a `with`-block; `.close()` will be called when the block exits, if applicable.

Whenever pikepdf opens a file, it will close it. If you open the file for pikepdf or give it a stream-like object to read from, you must release that object when appropriate.

Examples

```
>>> with Pdf.open("test.pdf") as pdf:
        ...
```

```
>>> pdf = Pdf.open("test.pdf", password="rosebud")
```

Parameters

- **filename_or_stream** (*Union[pathlib.Path, str, BinaryIO]*) – Filename of PDF to open.
- **password** (*Union[str, bytes]*) – User or owner password to open an encrypted PDF. If the type of this parameter is `str` it will be encoded as UTF-8. If the type is `bytes` it will be saved verbatim. Passwords are always padded or truncated to 32 bytes internally. Use ASCII passwords for maximum compatibility.
- **hex_password** (*bool*) – If True, interpret the password as a hex-encoded version of the exact encryption key to use, without performing the normal key computation. Useful in forensics.
- **ignore_xref_streams** (*bool*) – If True, ignore cross-reference streams. See qpdf documentation.
- **suppress_warnings** (*bool*) – If True (default), warnings are not printed to stderr. Use *pikepdf.Pdf.get_warnings()* to retrieve warnings.
- **attempt_recovery** (*bool*) – If True (default), attempt to recover from PDF parsing errors.
- **inherit_page_attributes** (*bool*) – If True (default), push attributes set on a group of pages to individual pages
- **access_mode** (*pikepdf._qpdf.AccessMode*) – If `.default`, pikepdf will decide how to access the file. Currently, it will always selected stream access. To attempt memory mapping and fallback to stream if memory mapping failed, use `.mmap`. Use `.mmap_only` to require memory mapping or fail (this is expected to only be useful for testing). Applications should be prepared to handle the SIGBUS signal on POSIX in the event that the file is successfully mapped but later goes away.
- **allow_overwriting_input** (*bool*) – If True, allows calling `.save()` to overwrite the input file. This is performed by loading the entire input file into memory at open time; this will use more memory and may recent performance especially when the opened file will not be modified.

Raises

- ***pikepdf.PasswordError*** – If the password failed to open the file.
- ***pikepdf.PdfError*** – If for other reasons we could not open the file.

- **TypeError** – If the type of `filename_or_stream` is not usable.
- **FileNotFoundError** – If the file was not found.

Return type pikepdf._qpdf.Pdf

open_metadata(*set_pikepdf_as_editor=True*, *update_docinfo=True*, *strict=False*)
Open the PDF's XMP metadata for editing.

There is no `.close()` function on the metadata object, since this is intended to be used inside a `with` block only.

For historical reasons, certain parts of PDF metadata are stored in two different locations and formats. This feature coordinates edits so that both types of metadata are updated consistently and "atomically" (assuming single threaded access). It operates on the `Pdf` in memory, not any file on disk. To persist metadata changes, you must still use `Pdf.save()`.

Example

```
>>> with pdf.open_metadata() as meta:
        meta['dc:title'] = 'Set the Dublic Core Title'
        meta['dc:description'] = 'Put the Abstract here'
```

Parameters

- **set_pikepdf_as_editor** (*bool*) – Automatically update the metadata `pdf:Producer` to show that this version of pikepdf is the most recent software to modify the metadata, and `xmp:MetadataDate` to timestamp the update. Recommended, except for testing.
- **update_docinfo** (*bool*) – Update the standard fields of DocumentInfo (the old PDF metadata dictionary) to match the corresponding XMP fields. The mapping is described in `PdfMetadata.DOCINFO_MAPPING`. Nonstandard DocumentInfo fields and XMP metadata fields with no DocumentInfo equivalent are ignored.
- **strict** (*bool*) – If `False` (the default), we aggressively attempt to recover from any parse errors in XMP, and if that fails we overwrite the XMP with an empty XMP record. If `True`, raise errors when either metadata bytes are not valid and well-formed XMP (and thus, XML). Some trivial cases that are equivalent to empty or incomplete "XMP skeletons" are never treated as errors, and always replaced with a proper empty XMP block. Certain errors may be logged.

Return type pikepdf.models.metadata.PdfMetadata

open_outline(*max_depth=15*, *strict=False*)
Open the PDF outline ("bookmarks") for editing.

Recommend for use in a `with` block. Changes are committed to the PDF when the block exits. (The `Pdf` must still be opened.)

Example

```
>>> with pdf.open_outline() as outline:
        outline.root.insert(0, OutlineItem('Intro', 0))
```

Parameters

- **max_depth** (*int*) – Maximum recursion depth of the outline to be imported and re-written to the document. `0` means only considering the root level, `1` the first-level sub-outline of each root element, and so on. Items beyond this depth will be silently ignored. Default is `15`.
- **strict** (*bool*) – With the default behavior (set to `False`), structural errors (e.g. reference loops) in the PDF document will only cancel processing further nodes on that particular level, recovering the valid parts of the document outline without raising an exception. When set to `True`, any such error will raise an `OutlineStructureError`, leaving the invalid parts in place. Similarly, outline objects that have been accidentally duplicated in the `Outline` container will be silently fixed (i.e. reproduced as new objects) or raise an `OutlineStructureError`.

Return type pikepdf.models.outlines.Outline

property owner_password_matched
Returns True if the owner password matched when the `Pdf` was opened.

It is possible for both the user and owner passwords to match.

property pages
Returns the list of pages.

Return type: pikepdf.PageList

property pdf_version
The version of the PDF specification used for this file, such as '1.7'.

remove_unreferenced_resources(*self:* pikepdf.Pdf) → None
Remove from /Resources of each page any object not referenced in page's contents

PDF pages may share resource dictionaries with other pages. If pikepdf is used for page splitting, pages may reference resources in their /Resources dictionary that are not actually required. This purges all unnecessary resource entries.

For clarity, if all references to any type of object are removed, that object will be excluded from the output PDF on save. (Conversely, only objects that are discoverable from the PDF's root object are included.) This function removes objects that are referenced from the page /Resources dictionary, but never called for in the content stream, making them unnecessary.

Suggested before saving, if content streams or /Resources dictionaries are edited.

property root
Deprecated alias for .Root, the /Root object of the PDF.

save(*filename_or_stream=None*, *static_id=False*, *preserve_pdfa=True*, *min_version=''*, *force_version=''*, *fix_metadata_version=True*, *compress_streams=True*, *stream_decode_level=None*, *object_stream_mode=<ObjectStreamMode.preserve: 1>*, *normalize_content=False*, *linearize=False*, *qdf=False*, *progress=None*, *encryption=None*, *recompress_flate=False*)

Save all modifications to this *pikepdf.Pdf*.

Parameters

- **filename_or_stream** (*Optional[Union[pathlib.Path, str, BinaryIO]]*) – Where to write the output. If a file exists in this location it will be overwritten. If the file was opened with `allow_overwriting_input=True`, then it is permitted to overwrite the original file, and this parameter may be omitted to implicitly use the original filename. Otherwise, the filename may not be the same as the input file, as overwriting the input file would corrupt data since pikepdf using lazy loading.
- **static_id** (*bool*) – Indicates that the `/ID` metadata, normally calculated as a hash of certain PDF contents and metadata including the current time, should instead be generated deterministically. Normally for debugging.
- **preserve_pdfa** (*bool*) – Ensures that the file is generated in a manner compliant with PDF/A and other stricter variants. This should be True, the default, in most cases.
- **min_version** (*Union[str, Tuple[str, int]]*) – Sets the minimum version of PDF specification that should be required. If left alone QPDF will decide. If a tuple, the second element is an integer, the extension level. If the version number is not a valid format, QPDF will decide what to do.
- **force_version** (*Union[str, Tuple[str, int]]*) – Override the version recommend by QPDF, potentially creating an invalid file that does not display in old versions. See QPDF manual for details. If a tuple, the second element is an integer, the extension level.
- **fix_metadata_version** (*bool*) – If `True` (default) and the XMP metadata contains the optional PDF version field, ensure the version in metadata is correct. If the XMP metadata does not contain a PDF version field, none will be added. To ensure that the field is added, edit the metadata and insert a placeholder value in `pdf:PDFVersion`. If XMP metadata does not exist, it will not be created regardless of the value of this argument.
- **object_stream_mode** (*pikepdf._qpdf.ObjectStreamMode*) – `disable` prevents the use of object streams. `preserve` keeps object streams from the input file. `generate` uses object streams wherever possible, creating the smallest files but requiring PDF 1.5+.
- **compress_streams** (*bool*) – Enables or disables the compression of stream objects in the PDF that are created without specifying any compression setting. Metadata is never compressed. By default this is set to `True`, and should be except for debugging. Existing streams in the PDF or streams will not be modified. To decompress existing streams, you must set both `compress_streams=False` and `stream_decode_level` to the desired decode level (e.g. `.generalized` will decompress most non-image content).
- **stream_decode_level** (*Optional[pikepdf._qpdf.StreamDecodeLevel]*) – Specifies how to encode stream objects. See documentation for `StreamDecodeLevel`.
- **recompress_flate** (*bool*) – When disabled (the default), qpdf does not uncompress and recompress streams compressed with the Flate compression algorithm. If True, pikepdf will instruct qpdf to do this, which may be useful if recompressing streams to a higher compression level.

- **normalize_content** (*bool*) – Enables parsing and reformatting the content stream within PDFs. This may debugging PDFs easier.
- **linearize** (*bool*) – Enables creating linear or "fast web view", where the file's contents are organized sequentially so that a viewer can begin rendering before it has the whole file. As a drawback, it tends to make files larger.
- **qdf** (*bool*) – Save output QDF mode. QDF mode is a special output mode in QPDF to allow editing of PDFs in a text editor. Use the program `fix-qdf` to fix convert back to a standard PDF.
- **progress** (*Optional[Callable[int, None]]*) – Specify a callback function that is called as the PDF is written. The function will be called with an integer between 0-100 as the sole parameter, the progress percentage. This function may not access or modify the PDF while it is being written, or data corruption will almost certainly occur.
- **encryption** (*Optional[Union[pikepdf.models.encryption.Encryption, bool]]*) – If `False` or omitted, existing encryption will be removed. If `True` encryption settings are copied from the originating PDF. Alternately, an `Encryption` object may be provided that sets the parameters for new encryption.

Raises

- ***PdfError*** –
- ***ForeignObjectError*** –

Return type None

You may call `.save()` multiple times with different parameters to generate different versions of a file, and you *may* continue to modify the file after saving it. `.save()` does not modify the `Pdf` object in memory, except possibly by updating the XMP metadata version with `fix_metadata_version`.

Note: *pikepdf.Pdf.remove_unreferenced_resources()* before saving may eliminate unnecessary resources from the output file if there are any objects (such as images) that are referenced in a page's Resources dictionary but never called in the page's content stream.

Note: pikepdf can read PDFs with incremental updates, but always coalesces any incremental updates into a single non-incremental PDF file when saving.

---

show_xref_table(*self:* pikepdf.Pdf) → None

Pretty-print the Pdf's xref (cross-reference table)

property trailer

Provides access to the PDF trailer object.

See section 7.5.5 of the PDF reference manual. Generally speaking, the trailer should not be modified with pikepdf, and modifying it may not work. Some of the values in the trailer are automatically changed when a file is saved.

property user_password_matched

Returns True if the user password matched when the `Pdf` was opened.

It is possible for both the user and owner passwords to match.

pikepdf.**open**()
Alias for *pikepdf.Pdf.open()*.

pikepdf.**new**()
Alias for *pikepdf.Pdf.new()*.

class pikepdf.**ObjectStreamMode**
Options for saving streams within PDFs, which are more a compact way of saving certains types of data that was added in PDF 1.5. All modern PDF viewers support object streams, but some third party tools and libraries cannot read them.

disable
Disable the use of object streams. If any object streams exist in the file, remove them when the file is saved.

preserve
Preserve any existing object streams in the original file. This is the default behavior.

generate
Generate object streams.

class pikepdf.**StreamDecodeLevel**
Options for decoding streams within PDFs.

none
Do not attempt to apply any filters. Streams remain as they appear in the original file. Note that uncompressed streams may still be compressed on output. You can disable that by saving with `.save(..., compress_streams=False)`.

generalized
This is the default. libqpdf will apply LZWDecode, ASCII85Decode, ASCIIHexDecode, and FlateDecode filters on the input. When saved with `compress_streams=True`, the default, the effect of this is that streams filtered with these older and less efficient filters will be recompressed with the Flate filter. As a special case, if a stream is already compressed with FlateDecode and `compress_streams=True`, the original compressed data will be preserved.

specialized
In addition to uncompressing the generalized compression formats, supported non-lossy compression will also be be decoded. At present, this includes the RunLengthDecode filter.

all
In addition to generalized and non-lossy specialized filters, supported lossy compression filters will be applied. At present, this includes DCTDecode (JPEG) compression. Note that compressing the resulting data with DCTDecode again will accumulate loss, so avoid multiple compression and decompression cycles. This is mostly useful for (low-level) retrieving image data; see *pikepdf.PdfImage* for the preferred method.

class pikepdf.**Encryption**(*, *owner*, *user*, *R=6*, *allow=Permissions(accessibility=True, extract=True, modify_annotation=True, modify_assembly=False, modify_form=True, modify_other=True, print_lowres=True, print_highres=True)*, *aes=True*, *metadata=True*)
Specify the encryption settings to apply when a PDF is saved.

Parameters

- **owner** – The owner password to use. This allows full control of the file. If blank, the PDF will be encrypted and present as "(SECURED)" in PDF viewers. If the owner password is blank, the user password should be as well.
- **user** – The user password to use. With this password, some restrictions will be imposed by a typical PDF reader. If blank, the PDF can be opened by anyone, but only modified as allowed by the permissions in `allow`.
- **R** – Select the security handler algorithm to use. Choose from: `2`, `3`, `4` or `6`. By default, the highest version of is selected (`6`). `5` is a deprecated algorithm that should not be used.
- **allow** – The permissions to set. If omitted, all permissions are granted to the user.
- **aes** – If True, request the AES algorithm. If False, use RC4. If omitted, AES is selected whenever possible (R >= 4).
- **metadata** – If True, also encrypt the PDF metadata. If False, metadata is not encrypted. Reading document metadata without decryption may be desirable in some cases. Requires `aes=True`. If omitted, metadata is encrypted whenever possible.

Object construction

class pikepdf.**Object**

append(*self:* pikepdf.Object, *arg0: object*) → None
Append another object to an array; fails if the object is not an array.

as_dict(*self:* pikepdf.Object) → pikepdf._ObjectMapping

as_list(*self:* pikepdf.Object) → pikepdf._ObjectList

emplace(*other*, *retain=(pikepdf.Name('/Parent'))*)
Copy all items from other without making a new object.

Particularly when working with pages, it may be desirable to remove all of the existing page's contents and emplace (insert) a new page on top of it, in a way that preserves all links and references to the original page. (Or similarly, for other Dictionary objects in a PDF.)

Any Dictionary keys in the iterable *retain* are preserved. By default, /Parent is retained.

When a page is assigned (`pdf.pages[0] = new_page`), only the application knows if references to the original the original page are still valid. For example, a PDF optimizer might restructure a page object into another visually similar one, and references would be valid; but for a program that reorganizes page contents such as a N-up compositor, references may not be valid anymore.

This method takes precautions to ensure that child objects in common with `self` and `other` are not inadvertently deleted.

Example

```
>>> pdf.pages[0].objgen
(16, 0)
>>> pdf.pages[0].emplace(pdf.pages[1])
>>> pdf.pages[0].objgen
(16, 0)  # Same object
```

Parameters **other** (*pikepdf.objects.Object*) –

extend (*self:* pikepdf.Object, *arg0: Iterable*) → None
Extend a pikepdf.Array with an iterable of other objects.

get (**args*, ***kwargs*)
Overloaded function.

1. get(self: pikepdf.Object, key: str, default: object = None) -> object

   For `pikepdf.Dictionary` or `pikepdf.Stream` objects, behave as `dict.get(key, default=None)`

2. get(self: pikepdf.Object, key: pikepdf.Object, default: object = None) -> object

   For `pikepdf.Dictionary` or `pikepdf.Stream` objects, behave as `dict.get(key, default=None)`

get_raw_stream_buffer (*self:* pikepdf.Object) → pikepdf.Buffer
Return a buffer protocol buffer describing the raw, encoded stream

get_stream_buffer (*self: pikepdf.Object*, *decode_level: pikepdf.StreamDecodeLevel* = <*StreamDecodeLevel.generalized: 1*>) → pikepdf.Buffer
Return a buffer protocol buffer describing the decoded stream.

is_owned_by (*self:* pikepdf.Object, *possible_owner:* pikepdf.Pdf) → bool
Test if this object is owned by the indicated *possible_owner*.

property is_rectangle
Returns True if the object is a rectangle (an array of 4 numbers)

items (*self:* pikepdf.Object) → Iterable

keys (*self:* pikepdf.Object) → Set[str]
For `pikepdf.Dictionary` or `pikepdf.Stream` objects, obtain the keys.

property objgen
Return the object-generation number pair for this object.

If this is a direct object, then the returned value is `(0, 0)`. By definition, if this is an indirect object, it has a "objgen", and can be looked up using this in the cross-reference (xref) table. Direct objects cannot necessarily be looked up.

The generation number is usually 0, except for PDFs that have been incrementally updated. Incrementally updated PDFs are now uncommon, since it does not take too long for modern CPUs to reconstruct an entire PDF. pikepdf will consolidate all incremental updates when saving.

page_contents_add (*self:* pikepdf.Object, *contents:* pikepdf.Object, *prepend: bool = False*) → None
Append or prepend to an existing page's content stream.

page_contents_coalesce (*self:* pikepdf.Object) → None
Coalesce an array of page content streams into a single content stream.

The PDF specification allows the `/Contents` object to contain either an array of content streams or a single content stream. However, it simplifies parsing and editing if there is only a single content stream. This function merges all content streams.

static parse (*stream: str*, *description: str* = '') → *pikepdf.Object*
Parse PDF binary representation into PDF objects.

read_bytes (*self: pikepdf.Object*, *decode_level: pikepdf.StreamDecodeLevel* = <*StreamDecodeLevel.generalized: 1*>) → bytes
Decode and read the content stream associated with this object.

read_raw_bytes (*self:* pikepdf.Object) → bytes
Read the content stream associated with this object without decoding

same_owner_as (*self:* pikepdf.Object, *arg0:* pikepdf.Object) → bool
Test if two objects are owned by the same *pikepdf.Pdf*.

property stream_dict
Access the dictionary key-values for a *pikepdf.Stream*.

to_json (*self:* pikepdf.Object, *dereference: bool* = *False*) → bytes
Convert to a QPDF JSON representation of the object.

See the QPDF manual for a description of its JSON representation. http://qpdf.sourceforge.net/files/qpdf-manual.html#ref.json

Not necessarily compatible with other PDF-JSON representations that exist in the wild.

- Names are encoded as UTF-8 strings
- Indirect references are encoded as strings containing `obj gen R`
- Strings are encoded as UTF-8 strings with unrepresentable binary characters encoded as `\uHHHH`
- Encoding streams just encodes the stream's dictionary; the stream data is not represented
- Object types that are only valid in content streams (inline image, operator) as well as "reserved" objects are not representable and will be serialized as `null`.

Parameters **dereference** (*bool*) – If True, dereference the object is this is an indirect object.

Returns JSON bytestring of object. The object is UTF-8 encoded and may be decoded to a Python str that represents the binary values `\x00-\xFF` as `U+0000` to `U+00FF`; that is, it may contain mojibake.

Return type bytes

unparse (*self:* pikepdf.Object, *resolved: bool* = *False*) → bytes
Convert PDF objects into their binary representation, optionally resolving indirect objects.

wrap_in_array (*self:* pikepdf.Object) → *pikepdf.Object*
Return the object wrapped in an array if not already an array.

write(*data*, *, *filter=None*, *decode_parms=None*, *type_check=True*)
Replace stream object's data with new (possibly compressed) *data*.

filter and *decode_parms* specify that compression that is present on the input *data*.

When writing the PDF in `pikepdf.Pdf.save()`, pikepdf may change the compression or apply compression to data that was not compressed, depending on the parameters given to that function. It will never change lossless to lossy encoding.

PNG and TIFF images, even if compressed, cannot be directly inserted into a PDF and displayed as images.

Parameters

- **data** (*bytes*) – the new data to use for replacement
- **filter** (*Optional[Union[pikepdf.objects.Name, pikepdf.objects.Array]]*) – The filter(s) with which the data is (already) encoded
- **decode_parms** (*Optional[Union[pikepdf.objects.Dictionary, pikepdf.objects.Array]]*) – Parameters for the filters with which the object is encode
- **type_check** (*bool*) – Check arguments; use False only if you want to intentionally create malformed PDFs.

If only one *filter* is specified, it may be a name such as *Name('/FlateDecode')*. If there are multiple filters, then array of names should be given.

If there is only one filter, *decode_parms* is a Dictionary of parameters for that filter. If there are multiple filters, then *decode_parms* is an Array of Dictionary, where each array index is corresponds to the filter.

class pikepdf.**Name**(*name*)
Constructs a PDF Name object

Names can be constructed with two notations:

1. `Name.Resources`
2. `Name('/Resources')`

The two are semantically equivalent. The former is preferred for names that are normally expected to be in a PDF. The latter is preferred for dynamic names and attributes.

static __new__(*cls*, *name*)
Create and return a new object. See help(type) for accurate signature.

Parameters **name** (*str*) –

class pikepdf.**String**(*s*)
Constructs a PDF String object

static __new__(*cls*, *s*)

Parameters **s** (*Union[str, bytes]*) – The string to use. String will be encoded for PDF, bytes will be constructed without encoding.

Returns pikepdf.Object

class pikepdf.**Array**(*a=None*)
Constructs a PDF Array object

static **__new__**(*cls*, *a=None*)

Parameters **a** (*Optional[Iterable]*) – An iterable of objects. All objects must be either *pikepdf.Object* or convertible to *pikepdf.Object*.

Returns pikepdf.Object

class pikepdf.**Dictionary**(*d=None*, ***kwargs*)
Constructs a PDF Dictionary object

static **__new__**(*cls*, *d=None*, ***kwargs*)
Constructs a PDF Dictionary from either a Python `dict` or keyword arguments.

These two examples are equivalent:

```
pikepdf.Dictionary({'/NameOne': 1, '/NameTwo': 'Two'})

pikepdf.Dictionary(NameOne=1, NameTwo='Two')
```

In either case, the keys must be strings, and the strings correspond to the desired Names in the PDF Dictionary. The values must all be convertible to *pikepdf.Object*.

Returns pikepdf.Object

class pikepdf.**Stream**(*owner*, *data=None*, *d=None*, ***kwargs*)
Constructs a PDF Stream object

static **__new__**(*cls*, *owner*, *data=None*, *d=None*, ***kwargs*)
Create a new stream object, which stores arbitrary binary data and may or may not be compressed.

A stream dictionary is like a pikepdf.Dictionary or Python dict, except it has a binary payload of data attached. The dictionary describes how the data is compressed or encoded.

The dictionary may be initialized just like pikepdf.Dictionary is initialized, using a mapping object or keyword arguments.

Parameters

- **owner** (Pdf) – The Pdf to which this stream shall be attached.
- **data** (*bytes*) – The data bytes for the stream.
- **d** – An optional mapping object that will be used to construct the stream's dictionary.
- **kwargs** – Keyword arguments that will define the stream dictionary. Do not set /Length here as pikepdf will manage this value. Set /Filter if the data is already encoded in some format.
- **obj** – Deprecated alias for *data*.

Returns pikepdf.Object

Examples

Using kwargs:

```
>>> s1 = pikepdf.Stream(
        pdf,
        b"uncompressed image data",
        BitsPerComponent=8,
        ColorSpace=Name.DeviceRGB,
        ...
    )
```

Using dict:

```
>>> d = pikepdf.Dictionary(...)
>>> s2 = pikepdf.Stream(
        pdf,
        b"data",
        d
    )
```

class pikepdf.**Operator**(*name*)

Constructs an operator for use in a content stream.

An Operator is one of a limited set of commands that can appear in PDF content streams (roughly the mini-language that draws objects, lines and text on a virtual PDF canvas). The commands *parse_content_stream()* and unparse_content_stream() create and expect Operators respectively, along with their operands.

pikepdf uses the special Operator "INLINE IMAGE" to denote an inline image in a content stream.

Internal objects

These objects are returned by other pikepdf objects. They are part of the API, but not intended to be created explicitly.

class pikepdf._qpdf.**PageList**

A list-like object enumerating a range of pages in a *pikepdf.Pdf*. It may be all of the pages or a subset.

append(*self: pikepdf.PageList*, *page: object*) → None

Add another page to the end.

extend(**args*, ***kwargs*)

Overloaded function.

1. extend(self: pikepdf.PageList, other: pikepdf.PageList) -> None

   Extend the Pdf by adding pages from another Pdf.pages.

2. extend(self: pikepdf.PageList, iterable: Iterable) -> None

   Extend the Pdf by adding pages from an iterable of pages.

index(**args*, ***kwargs*)

Overloaded function.

1. index(self: pikepdf.PageList, arg0: pikepdf.Object) -> int

Given a pikepdf.Object that is a page, find the index.

That is, returns `n` such that `pdf.pages[n] == this_page`. A `ValueError` exception is thrown if the page does not belong to to this `Pdf`.

2. index(self: pikepdf.PageList, arg0: pikepdf.Page) -> int

   Given a pikepdf.Page (page helper), find the index.

   That is, returns `n` such that `pdf.pages[n] == this_page`. A `ValueError` exception is thrown if the page does not belong to to this `Pdf`.

insert (*self: pikepdf.PageList*, *index: int*, *obj: object*) → None
Insert a page at the specified location.

Parameters

- **index** (*int*) – location at which to insert page, 0-based indexing
- **obj** (`pikepdf.Object`) – page object to insert

p (*self: pikepdf.PageList*, *pnum: int*) → *pikepdf.Object*
Convenience - look up page number in ordinal numbering, `.p(1)` is first page

remove (*self: pikepdf.PageList*, ***kwargs*) → None
Remove a page (using 1-based numbering)

Parameters **p** (*int*) – 1-based page number

reverse (*self: pikepdf.PageList*) → None
Reverse the order of pages.

1.3.15 Support models

Support models are abstracts over "raw" objects within a Pdf. For example, a page in a PDF is a Dictionary with set to `/Type` of `/Page`. The Dictionary in that case is the "raw" object. Upon establishing what type of object it is, we can wrap it with a support model that adds features to ensure consistency with the PDF specification.

pikepdf does not currently apply support models to "raw" objects automatically, but might do so in a future release (this would break backward compatibility).

For example, to initialize a `Page` support model:

```
from pikepdf import Pdf, Page

Pdf = open(...)
page_support_model = Page(pdf.pages[0])
```

class pikepdf.**Page**

add_content_token_filter (*self: pikepdf.Page*, *tf: pikepdf.Object::TokenFilter*) → None
Attach a *pikepdf.TokenFilter* to a page's content stream.

This function applies token filters lazily, if/when the page's content stream is read for any reason, such as when the PDF is saved. If never access, the token filter is not applied.

Multiple token filters may be added to a page/content stream.

If the page's contents is an array of streams, it is coalesced.

add_resource(*res*, *res_type*, *name=None*, ***, *prefix=''*, *replace_existing=True*)
Adds a new resource to the page's Resources dictionary.

If the Resources dictionaries do not exist, they will be created.

Parameters

- **self** – The object to add to the resources dictionary.
- **res** (*pikepdf.objects.Object*) – The resource dictionary object to add.
- **res_type** (*pikepdf.objects.Name*) – Should be one of the following Resource dictionary types: ExtGState, ColorSpace, Pattern, Shading, XObject, Font, Properties.
- **name** (*Optional[pikepdf.objects.Name]*) – The name of the object. If omitted, a random name will be generated with enough randomness to be globally unique.
- **prefix** (*str*) – A prefix for the name of the object. Allows conveniently namespacing when using random names, e.g. prefix="Im" for images. Mutually exclusive with name parameter.
- **replace_existing** (*bool*) – If the name already exists in one of the resource dictionaries, remove it.

Returns The name of the object.

Return type str

Example

```
>>> resource_name␣
↪= Page(pdf.pages[0]).add_resource(formxobj, Name.XObject)
```

as_form_xobject(*self:* pikepdf.Page, *handle_transformations: bool = True*) → *pikepdf.Object*
Return a form XObject that draws this page.

This is useful for n-up operations, underlay, overlay, thumbnail generation, or any other case in which it is useful to replicate the contents of a page in some other context. The dictionaries are shallow copies of the original page dictionary, and the contents are coalesced from the page's contents. The resulting object handle is not referenced anywhere.

Parameters **handle_transformations** (*bool*) – If True, the resulting form XObject's `/Matrix` will be set to replicate rotation (`/Rotate`) and scaling (`/UserUnit`) in the page's dictionary. In this way, the page's transformations will be preserved when placing this object on another page.

contents_coalesce(*self:* pikepdf.Page) → None
Coalesce a page's content streams.

A page's content may be a stream or an array of streams. If this page's content is an array, concatenate the streams into a single stream. This can be useful when working with files that split content streams in arbitrary spots, such as in the middle of a token, as that can confuse some software.

externalize_inline_images(*self:* pikepdf.Page, *min_size: int = 0*) → None
Convert inlines image to normal (external) images.

Parameters **min_size** (*int*) – minimum size in bytes

get_filtered_contents (*self:* pikepdf.Page, *tf:* TokenFilter) → bytes
Apply a *pikepdf.TokenFilter* to a content stream, without modifying it.

This may be used when the results of a token filter do not need to be applied, such as when filtering is being used to retrieve information rather than edit the content stream.

Note that it is possible to create a subclassed `TokenFilter` that saves information of interest to its object attributes; it is not necessary to return data in the content stream.

To modify the content stream, use *pikepdf.Page.add_content_token_filter()*.

Returns the modified content stream

Return type bytes

property index
Returns the zero-based index of this page in the pages list.

That is, returns `n` such that `pdf.pages[n] == this_page`. A `ValueError` exception is thrown if the page is not attached to a `Pdf`.

Requires O(n) search.

property label
Returns the page label for this page, accounting for section numbers.

For example, if the PDF defines a preface with lower case Roman numerals (i, ii, iii...), followed by standard numbers, followed by an appendix (A-1, A-2, ...), this function returns the appropriate label as a string.

It is possible for a PDF to define page labels such that multiple pages have the same labels. Labels are not guaranteed to be unique.

Note that this requires a O(n) search over all pages, to look up the page's index.

property obj
Get the underlying *pikepdf.Object*.

parse_contents (*self:* pikepdf.Page, *arg0: pikepdf.StreamParser*) → None
Parse a page's content streams using a `pikepdf.StreamParser`.

The content stream may be interpreted by the StreamParser but is not altered.

If the page's contents is an array of streams, it is coalesced.

remove_unreferenced_resources (*self:* pikepdf.Page) → None
Removes from the resources dictionary any object not referenced in the content stream.

A page's resources dictionary maps names to objects elsewhere in the file. This method walks through a page's contents and keeps tracks of which resources are referenced somewhere in the contents. Then it removes from the resources dictionary any object that is not referenced in the contents. This method is used by page splitting code to avoid copying unused objects in files that used shared resource dictionaries across multiple pages.

rotate (*self:* pikepdf.Page, *angle: int*, *relative: bool*) → None
Rotate a page.

If `relative` is `False`, set the rotation of the page to angle. Otherwise, add angle to the rotation of the page. `angle` must be a multiple of `90`. Adding `90` to the rotation rotates clockwise by `90` degrees.

class pikepdf.**PdfMatrix**(**args*)

Support class for PDF content stream matrices

PDF content stream matrices are 3x3 matrices summarized by a shorthand `(a, b, c, d, e, f)` which correspond to the first two column vectors. The final column vector is always `(0, 0, 1)` since this is using homogenous coordinates.

PDF uses row vectors. That is, `vr @ A'` gives the effect of transforming a row vector `vr=(x, y, 1)` by the matrix `A'`. Most textbook treatments use `A @ vc` where the column vector `vc=(x, y, 1)'`.

(`@` is the Python matrix multiplication operator.)

Addition and other operations are not implemented because they're not that meaningful in a PDF context (they can be defined and are mathematically meaningful in general).

PdfMatrix objects are immutable. All transformations on them produce a new matrix.

a

b

c

d

e

f

Return one of the six "active values" of the matrix.

encode()

Encode this matrix in binary suitable for including in a PDF

static identity()

Constructs and returns an identity matrix

rotated(*angle_degrees_ccw*)

Concatenates a rotation matrix on this matrix

scaled(*x*, *y*)

Concatenates a scaling matrix on this matrix

property shorthand

Return the 6-tuple (a,b,c,d,e,f) that describes this matrix

translated(*x*, *y*)

Translates this matrix

class pikepdf.**PdfImage**(*obj*)

Support class to provide a consistent API for manipulating PDF images

The data structure for images inside PDFs is irregular and flexible, making it difficult to work with without introducing errors for less typical cases. This class addresses these difficulties by providing a regular, Pythonic API similar in spirit (and convertible to) the Python Pillow imaging library.

as_pil_image()
Extract the image as a Pillow Image, using decompression as necessary

Returns PIL.Image.Image

extract_to(*, *stream=None*, *fileprefix=''*)
Attempt to extract the image directly to a usable image file

If possible, the compressed data is extracted and inserted into a compressed image file format without transcoding the compressed content. If this is not possible, the data will be decompressed and extracted to an appropriate format.

Because it is not known until attempted what image format will be extracted, users should not assume what format they are getting back. When saving the image to a file, use a temporary filename, and then rename the file to its final name based on the returned file extension.

Examples

```
>>> im.extract_to(stream=bytes_io)
'.png'
```

```
>>> im.extract_to(fileprefix='/tmp/image00')
'/tmp/image00.jpg'
```

Parameters

- **stream** – Writable stream to write data to.
- **fileprefix** (*str or Path*) – The path to write the extracted image to, without the file extension.

Returns If *fileprefix* was provided, then the fileprefix with the appropriate extension. If no *fileprefix*, then an extension indicating the file type.

Return type: str

get_stream_buffer(*decode_level=<StreamDecodeLevel.specialized: 2>*)
Access this image with the buffer protocol

property icc
If an ICC profile is attached, return a Pillow object that describe it.

Most of the information may be found in `icc.profile`.

Returns PIL.ImageCms.ImageCmsProfile

property is_inline
`False` for image XObject

read_bytes(*decode_level=<StreamDecodeLevel.specialized: 2>*)
Decompress this image and return it as unencoded bytes

show()
Show the image however PIL wants to

class pikepdf.**PdfInlineImage**(*, *image_data*, *image_object*)
Support class for PDF inline images

class pikepdf.models.**PdfMetadata**(*pdf*, *pikepdf_mark=True*, *sync_docinfo=True*, *overwrite_invalid_xml=True*)
Read and edit the metadata associated with a PDF

The PDF specification contain two types of metadata, the newer XMP (Extensible Metadata Platform, XML-based) and older DocumentInformation dictionary. The PDF 2.0 specification removes the DocumentInformation dictionary.

This primarily works with XMP metadata, but includes methods to generate XMP from DocumentInformation and will also coordinate updates to DocumentInformation so that the two are kept consistent.

XMP metadata fields may be accessed using the full XML namespace URI or the short name. For example `metadata['dc:description']` and `metadata['{http://purl.org/dc/elements/1.1/}description']` both refer to the same field. Several common XML namespaces are registered automatically.

See the XMP specification for details of allowable fields.

To update metadata, use a with block.

Example

```
>>> with pdf.open_metadata() as records:
        records['dc:title'] = 'New Title'
```

See also:

pikepdf.Pdf.open_metadata()

load_from_docinfo(*docinfo*, *delete_missing=False*, *raise_failure=False*)
Populate the XMP metadata object with DocumentInfo

Parameters

- **docinfo** – a DocumentInfo, e.g pdf.docinfo
- **delete_missing** (*bool*) – if the entry is not DocumentInfo, delete the equivalent from XMP
- **raise_failure** (*bool*) – if True, raise any failure to convert docinfo; otherwise warn and continue

Return type None

A few entries in the deprecated DocumentInfo dictionary are considered approximately equivalent to certain XMP records. This method copies those entries into the XMP metadata.

property pdfa_status
Returns the PDF/A conformance level claimed by this PDF, or False

A PDF may claim to PDF/A compliant without this being true. Use an independent verifier such as veraPDF to test if a PDF is truly conformant.

Returns The conformance level of the PDF/A, or an empty string if the PDF does not claim PDF/A conformance. Possible valid values are: 1A, 1B, 2A, 2B, 2U, 3A, 3B, 3U.

Return type str

property pdfx_status

Returns the PDF/X conformance level claimed by this PDF, or False

A PDF may claim to PDF/X compliant without this being true. Use an independent verifier such as veraPDF to test if a PDF is truly conformant.

Returns The conformance level of the PDF/X, or an empty string if the PDF does not claim PDF/X conformance.

Return type str

class pikepdf.models.**Encryption**(*, *owner*, *user*, *R=6*, *allow=Permissions(accessibility=True, extract=True, modify_annotation=True, modify_assembly=False, modify_form=True, modify_other=True, print_lowres=True, print_highres=True)*, *aes=True*, *metadata=True*)

Specify the encryption settings to apply when a PDF is saved.

Parameters

- **owner** – The owner password to use. This allows full control of the file. If blank, the PDF will be encrypted and present as "(SECURED)" in PDF viewers. If the owner password is blank, the user password should be as well.
- **user** – The user password to use. With this password, some restrictions will be imposed by a typical PDF reader. If blank, the PDF can be opened by anyone, but only modified as allowed by the permissions in `allow`.
- **R** – Select the security handler algorithm to use. Choose from: `2`, `3`, `4` or `6`. By default, the highest version of is selected (`6`). `5` is a deprecated algorithm that should not be used.
- **allow** – The permissions to set. If omitted, all permissions are granted to the user.
- **aes** – If True, request the AES algorithm. If False, use RC4. If omitted, AES is selected whenever possible (R >= 4).
- **metadata** – If True, also encrypt the PDF metadata. If False, metadata is not encrypted. Reading document metadata without decryption may be desirable in some cases. Requires `aes=True`. If omitted, metadata is encrypted whenever possible.

class pikepdf.models.**Outline**(*pdf*, *max_depth=15*, *strict=False*)

Maintains a intuitive interface for creating and editing PDF document outlines, according to the PDF reference manual (ISO32000:2008) section 12.3.

Parameters

- **pdf** – PDF document object.
- **max_depth** – Maximum recursion depth to consider when reading the outline.
- **strict** – If set to `False` (default) silently ignores structural errors. Setting it to `True` raises a *pikepdf.OutlineStructureError* if any object references re-occur while the outline is being read or written.

See also:

pikepdf.Pdf.open_outline()

class pikepdf.models.**OutlineItem**(*title*, *destination=None*, *page_location=None*, *action=None*, *obj=None*, ***, *left=None*, *top=None*, *right=None*, *bottom=None*, *zoom=None*)

Manages a single item in a PDF document outlines structure, including nested items.

Parameters

- **title** – Title of the outlines item.
- **destination** – Page number, destination name, or any other PDF object to be used as a reference when clicking on the outlines entry. Note this should be `None` if an action is used instead. If set to a page number, it will be resolved to a reference at the time of writing the outlines back to the document.
- **page_location** – Supplemental page location for a page number in `destination`, e.g. `PageLocation.Fit`. May also be a simple string such as `'FitH'`.
- **action** – Action to perform when clicking on this item. Will be ignored during writing if `destination` is also set.
- **obj** – `Dictionary` object representing this outlines item in a `Pdf`. May be `None` for creating a new object. If present, an existing object is modified in-place during writing and original attributes are retained.
- **left** – Describes the viewport position associated with a destination.
- **top** – Describes the viewport position associated with a destination.
- **bottom** – Describes the viewport position associated with a destination.
- **right** – Describes the viewport position associated with a destination.
- **zoom** – Describes the viewport position associated with a destination.

  This object does not contain any information about higher-level or neighboring elements.

classmethod from_dictionary_object(*obj*)

Creates a `OutlineItem` from a PDF document's `Dictionary` object. Does not process nested items.

Parameters **obj** (*pikepdf.objects.Dictionary*) – `Dictionary` object representing a single outline node.

to_dictionary_object(*pdf*, *create_new=False*)

Creates a `Dictionary` object from this outline node's data, or updates the existing object. Page numbers are resolved to a page reference on the input `Pdf` object.

Parameters

- **pdf** (*pikepdf._qpdf.Pdf*) – PDF document object.
- **create_new** (*bool*) – If set to `True`, creates a new object instead of modifying an existing one in-place.

Return type pikepdf.objects.Dictionary

class pikepdf.**Permissions**(*accessibility=True*, *extract=True*, *modify_annotation=True*, *modify_assembly=False*, *modify_form=True*, *modify_other=True*, *print_lowres=True*, *print_highres=True*)

Stores the user-level permissions for an encrypted PDF.

A compliant PDF reader/writer should enforce these restrictions on people who have the user password and not the owner password. In practice, either password is sufficient to decrypt all document contents. A person who has the owner password should be allowed to modify the document in any way. pikepdf does not enforce the restrictions in any way; it is up to application developers to enforce them as they see fit.

Unencrypted PDFs implicitly have all permissions allowed. Permissions can only be changed when a PDF is saved.

property accessibility

Can users use screen readers and accessibility tools to read the PDF?

property extract

Can users extract contents?

property modify_annotation

Can users modify annotations?

property modify_assembly

Can users arrange document contents?

property modify_form

Can users fill out formts?

property modify_other

Can users modify the document?

property print_highres

Can users print the document at high resolution?

property print_lowres

Can users print the document at low resolution?

class pikepdf.models.**EncryptionMethod**

Describes which encryption method was used on a particular part of a PDF. These values are returned by `pikepdf.EncryptionInfo` but are not currently used to specify how encryption is requested.

none

Data was not encrypted.

unknown

An unknown algorithm was used.

rc4

The RC4 encryption algorithm was used (obsolete).

aes

The AES-based algorithm was used as described in the PDF 1.7 reference manual.

aesv3

An improved version of the AES-based algorithm was used as described in the Adobe Supplement to the ISO 32000, requiring PDF 1.7 extension level 3. This algorithm still uses AES, but allows both AES-128 and AES-256, and improves how the key is derived from the password.

class pikepdf.models.**EncryptionInfo**(*encdict*)
Reports encryption information for an encrypted PDF.

This information may not be changed, except when a PDF is saved. This object is not used to specify the encryption settings to save a PDF, due to non-overlapping information requirements.

property P
Encoded permission bits.

See `Pdf.allow()` instead.

property R
Revision number of the security handler.

property V
Version of PDF password algorithm.

property bits
The number of encryption bits.

property encryption_key
The RC4 or AES encryption key used for this file.

property file_method
Encryption method used to encode the whole file.

property stream_method
Encryption method used to encode streams.

property string_method
Encryption method used to encode strings.

property user_password
If possible, return the user password.

The user password can only be retrieved when a PDF is opened with the owner password and when older versions of the encryption algorithm are used.

The password is always returned as `bytes` even if it has a clear Unicode representation.

1.3.16 Content streams

In PDF, drawing operations are all performed in content streams that describe the positioning and drawing order of all graphics (including text, images and vector drawing).

pikepdf (and libqpdf) provide two tools for interpreting content streams: a parser and filter. The parser returns higher level information, conveniently grouping all commands with their operands. The parser is useful when one wants to retrieve information from a content stream, such as determine the position of an element. The parser should not be used to edit or reconstruct the content stream because some subtleties are lost in parsing.

The token filter works at a lower level, considering each token including comments, and distinguishing different types of spaces. This allows modifying content streams. A TokenFilter must be subclassed; the specialized version describes how it should transform the stream of tokens.

pikepdf.**parse_content_stream**(*page_or_stream*, *operators=''*)

Parse a PDF content stream into a sequence of instructions.

A PDF content stream is list of instructions that describe where to render the text and graphics in a PDF. This is the starting point for analyzing PDFs.

If the input is a page and page.Contents is an array, then the content stream is automatically treated as one coalesced stream.

Each instruction contains at least one operator and zero or more operands.

This function does not have anything to do with opening a PDF file itself or processing data from a whole PDF. It is for processing a specific object inside a PDF that is already opened.

Parameters

- **page_or_stream** (*pikepdf.objects.Object*) – A page object, or the content stream attached to another object such as a Form XObject.
- **operators** (*str*) – A space-separated string of operators to whitelist. For example 'q Q cm Do' will return only operators that pertain to drawing images. Use 'BI ID EI' for inline images. All other operators and associated tokens are ignored. If blank, all tokens are accepted.

Returns

List of (operands, command) tuples where command is an operator (str) and `operands` is a tuple of str; the PDF drawing command and the command's operands, respectively.

Return type list

Example

```
>>> pdf = pikepdf.Pdf.open(input_pdf)
>>> page = pdf.pages[0]
>>> for operands, command in parse_content_stream(page):
>>>     print(command)
```

class pikepdf.**Token**

property raw_value

The binary representation of a token.

Return type: bytes

property type_

Returns the type of token.

Return type: pikepdf.TokenType

property value

Interprets the token as a string.

Return type: str or bytes

class pikepdf.**TokenType**

When filtering content streams, each token is labeled according to the role in plays.

Standard tokens

array_open

array_close

brace_open

brace_close

dict_open

dict_close
These tokens mark the start and end of an array, text string, and dictionary, respectively.

integer

real

null

bool
The token data represents an integer, real number, null or boolean, respectively.

Name
The token is the name of an object. In practice, these are among the most interesting tokens.

inline_image
An inline image in the content stream. The whole inline image is represented by the single token.

Lexical tokens

comment
Signifies a comment that appears in the content stream.

word
Otherwise uncategorized bytes are returned as `word` tokens. PDF operators are words.

bad
An invalid token.

space
Whitespace within the content stream.

eof
Denotes the end of the tokens in this content stream.

class pikepdf.**TokenFilter**

handle_token(*self:* pikepdf.TokenFilter, *token:* pikepdf.Token = *pikepdf.Token()*) → object
Handle a *pikepdf.Token*.

This is an abstract method that must be defined in a subclass of `TokenFilter`. The method will be called for each token. The implementation may return either `None` to discard the token, the original token to include it, a new token, or an iterable containing zero or more tokens. An implementation may also buffer tokens and release them in groups (for example, it could collect an entire PDF command with all of its operands, and then return all of it).

The final token will always be a token of type `TokenType.eof`, (unless an exception is raised).

If this method raises an exception, the exception will be caught by C++, consumed, and repalced with a less informative exception. Use *pikepdf.Pdf.get_warnings()* to view the original.

Return type: None or list or pikepdf.Token

1.3.17 Exceptions

exception pikepdf.**PdfError**
General pikepdf-specific exception.

exception pikepdf.**PasswordError**
Exception thrown when the supplied password is incorrect.

exception pikepdf.**ForeignObjectError**
Exception thrown when a complex object was copied into a foreign PDF without using *Pdf.copy_foreign()*.

exception pikepdf.**OutlineStructureError**
Exception throw when an `/Outlines` object violates constraints imposed by the PDF reference manual.

exception pikepdf.**UnsupportedImageTypeError**
Exception throw when attempting to manipulate a PDF image of a complex type that pikepdf does not currently support.

1.3.18 Architecture

pikepdf uses pybind11 to bind the C++ interface of QPDF. pybind11 was selected after evaluating Cython, CFFI and SWIG as possible binding solutions.

In addition to bindings pikepdf includes support code written in a mix of C++ and Python, mainly to present a clean Pythonic interface to C++ and implement higher level functionality.

Internals

Internally the package presents a module named `pikepdf` from which objects can be imported. The C++ extension module is currently named `pikepdf._qpdf`. Users of `pikepdf` should not directly access `_qpdf` since it is an internal interface.

In general, modules or objects behind an underscore are private (although they may be returned in some situations).

Thread safety

Because of the global interpreter lock (GIL), it is safe to read pikepdf objects across Python threads. Also because of the GIL, there may not be much performance gain from doing so.

If one or more threads will be modifying pikepdf objects, you will have to coordinate read and write access with a `threading.Lock`.

It is not currently possible to pickle pikepdf objects or marshall them across process boundaries (as would be required to use pikepdf in `multiprocessing`). If this were implemented, it would not be much more efficient than saving a full PDF and sending it to another process. Parallelizing work (for example, by dividing work by PDF pages) can still be achieved by having each worker process open the same file.

File handles

Because of technical limitations in underlying libraries, pikepdf keeps the source PDF file open when a content is copied from it to another PDF, even when all Python variables pointing to the source are removed. If a PDF is being assembled from many sources, then all of those sources are held open in memory.

1.3.19 Contributing guidelines

Contributions are welcome!

Big changes

Please open a new issue to discuss or propose a major change. Not only is it fun to discuss big ideas, but we might save each other's time too. Perhaps some of the work you're contemplating is already half-done in a development branch.

Code style: Python

We use PEP8, `black` for code formatting and `isort` for import sorting. The settings for these programs are in `pyproject.toml` and `setup.cfg`. Pull requests should follow the style guide. One difference we use from "black" style is that strings shown to the user are always in double quotes (`"`) and strings for internal uses are in single quotes (`'`).

Code style: C++

In lieu of a C++ autoformatter that is half as good as `black`, formatting is more lax.

We have no idea whether to put the pointer designator beside the type or the variable. It logically belongs to the type, but looks better beside the variable, and ugly in between.

As a general rule for code style, PEP8-style naming conventions should be used. That is, variable and method names are snake_case, class names

are CamelCase. Our coding conventions are closer to pybind11's than QPDF's. When a C++ object wraps is a Python object, it should follow the Python naming conventions for that type of object, e.g. `auto Decimal = py::module_::import("decimal").attr("Decimal")` for a reference to the Python `Decimal` class.

We don't like the traditional C++ .cpp/.h separation that results in a lot of repetition. Headers that are included by only one .cpp can contain a complete class.

Use RAII. Avoid naked pointers. Use the STL, use `std::string` instead of `char *`. Use `#pragma once` as a header guard; it's been around for 25 years.

Tests

New features should come with tests that confirm their correctness.

New dependencies

If you are proposing a change that will require a new dependency, we prefer dependencies that are already packaged by Debian or Red Hat. This makes life much easier for our downstream package maintainers.

Dependencies must also be compatible with the source code license.

English style guide

pikepdf is always spelled "pikepdf", and never capitalized even at the beginning of a sentence.

Periodic allusions to fish are required, and the writer shall be energetic and mildly amusing.

Known ports/packagers

pikepdf has been ported to many platforms already. If you are interesting in porting to a new platform, check with Repology to see the status of that platform.

1.3.20 Debugging

pikepdf does a complex job in providing bindings from Python to a C++ library, both of which have different ideas about how to manage memory. This page documents some methods that may help should it be necessary to debug the Python C++ extension (`pikepdf._qpdf`).

Enabling QPDF tracing

Setting the environment variables `TC_SCOPE=qpdf` and `TC_FILENAME=your_log_file.txt` will cause libqpdf to log debug messages to the designated file. For example:

```
env TC_SCOPE=qpdf␣
↪TC_FILENAME=libqpdf_log.txt python my_pikepdf_script.py
```

Compiling a debug build of QPDF

It may be helpful to create a debug build of QPDF.

Download QPDF and compile a debug build:

```
# in QPDF source tree
cd $QPDF_SOURCE_TREE
./configure␣
↪CFLAGS='-g -O0' CPPFLAGS='-g -O0' CXXFLAGS='-g -O0'
make -j
```

Compile and link against QPDF source tree

Build `pikepdf._qpdf` against the version of QPDF above, rather than the system version:

```
env QPDF_SOURCE_TREE=
↪<location of QPDF> python setup.py build_ext --inplace
```

In addition to building against the QPDF source, you'll need to force your operating system to load the locally compiled version of QPDF instead of the installed version:

```
# Linux
env LD_LIBRARY_
↪PATH=$QPDF_SOURCE_TREE/libqpdf/build/.libs python ...
```

```
# macOS - may require disabling System Integrity Protection
env DYLD_LIBRARY_
↪PATH=$QPDF_SOURCE_TREE/libqpdf/build/.libs python ...
```

On macOS you can make the library persistent by changing the name of the library to use in pikepdf's binary extension module:

```
install_name_tool -change /usr/local/lib/libqpdf*.dylib \
    $QPDF_SOURCE_TREE/libqpdf/build/.libs/libqpdf*.dylib \
    src/pikepdf/_qpdf.cpython*.so
```

You can also run Python through a debugger (`gdb` or `lldb`) in this manner, and you will have access to the source code for both pikepdf's C++ and QPDF.

Valgrind

Valgrind may also be helpful - see the Python documentation for information on setting up Python and Valgrind.

1.3.21 Resources

- QPDF Manual
- PDF 1.7 ISO Specification PDF 32000-1:2008
- Adobe Supplement to ISO 32000 BaseVersion 1.7 ExtensionLevel 3, Adobe Acrobat 9.0, June 2008, for AESv3
- Other Adobe extensions to the PDF specification

For information about copyrights and licenses, including those associated with the images in this documentation, see the file `debian/copyright`.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W