
pyHanko

Release 0.14.0

Matthias Valvekens

Sep 17, 2022

CONTENTS:

1	CLI user's guide	3
1.1	Signing PDF files	4
1.1.1	Some background on PDF signatures	4
1.1.2	Creating signature fields	5
1.1.3	Creating simple signatures	5
1.1.4	Creating signatures with long lifetimes	8
1.1.5	Customising signature appearances	10
1.2	Validating PDF signatures	10
1.2.1	Basic use	10
1.2.2	Factors in play when validating a signature	11
1.3	Stamping PDF files	13
1.4	Configuration options	13
1.4.1	Config file location	13
1.4.2	Configuration options	13
2	Library (SDK) user's guide	21
2.1	Reading and writing PDF files	21
2.1.1	Reading files	21
2.1.2	Modifying files	22
2.2	Signature fields	23
2.2.1	General API design	23
2.2.2	Positioning	23
2.2.3	Seed value settings	24
2.2.4	Document modification policy settings	25
2.3	Signing functionality	26
2.3.1	General API design	27
2.3.2	A simple example	27
2.3.3	Signature appearance generation	29
2.3.4	Timestamp handling	33
2.3.5	Creating PAdES signatures	34
2.3.6	Using aiohttp for network I/O	35
2.3.7	Extending Signer	37
2.3.8	The low-level PdfCMSEmbedder API	37
2.3.9	Interrupted signing	39
2.3.10	Generic data signing	41
2.4	Validation functionality	42
2.4.1	General API design	42
2.4.2	Accessing signatures in a document	43
2.4.3	Validating a PDF signature	43
2.4.4	Long-term verifiability checking	44

2.4.5	Incremental update analysis	45
2.4.6	Probing different aspects of the validity of a signature	47
2.5	The pdf-utils package	47
2.5.1	Background and future perspectives	47
2.5.2	PDF object model	48
2.5.3	PDF content abstractions	48
2.6	Advanced examples	49
2.6.1	A custom Signer to use AWS KMS asynchronously	49
3	API reference	53
3.1	pyhanko package	53
3.1.1	Subpackages	53
3.1.2	Submodules	248
4	Release history	261
4.1	0.14.0	261
4.1.1	Note	261
4.1.2	Breaking changes	261
4.1.3	Dependency updates	262
4.1.4	Bugs fixed	262
4.1.5	New features and enhancements	262
4.2	0.13.2	263
4.2.1	Note	263
4.2.2	Dependency updates	264
4.2.3	Bugs fixed	264
4.3	0.13.1	264
4.3.1	Note	264
4.3.2	Dependency updates	264
4.4	0.13.0	264
4.4.1	Note	264
4.4.2	Dependency updates	265
4.4.3	Bugs fixed	265
4.4.4	New features and enhancements	265
4.5	0.12.1	266
4.5.1	Dependency updates	266
4.5.2	Bugs fixed	266
4.6	0.12.0	266
4.6.1	Note	266
4.6.2	New features and enhancements	266
4.6.3	Bugs fixed	267
4.7	0.11.0	267
4.7.1	Dependency changes	267
4.7.2	Breaking changes	267
4.7.3	New features and enhancements	268
4.7.4	Bugs fixed	268
4.8	0.10.0	269
4.8.1	Dependency changes	269
4.8.2	New features and enhancements	269
4.8.3	Bugs fixed	270
4.9	0.9.0	270
4.9.1	Dependency changes	270
4.9.2	API-breaking changes	270
4.9.3	New features and enhancements	271
4.9.4	Bugs fixed	272

4.10	0.8.0	272
4.10.1	Dependency changes	272
4.10.2	API-breaking changes	272
4.10.3	New features and enhancements	272
4.10.4	Bugs fixed	273
4.11	0.7.0	273
4.11.1	Dependency changes	274
4.11.2	API-breaking changes	274
4.11.3	New features and enhancements	275
4.11.4	Bugs fixed	276
4.12	0.6.1	276
4.12.1	Dependency changes	277
4.12.2	Bugs fixed	277
4.13	0.6.0	277
4.13.1	Dependency changes	277
4.13.2	New features and enhancements	278
4.13.3	Bugs fixed	279
4.14	0.5.1	279
4.14.1	Bugs fixed	279
4.15	0.5.0	279
4.15.1	Dependency changes	279
4.15.2	New features and enhancements	279
4.15.3	Bugs fixed	281
4.16	0.4.0	281
4.16.1	New features and enhancements	281
4.16.2	Bugs fixed	282
4.17	0.3.0	283
4.17.1	New features and enhancements	283
4.17.2	Bugs fixed	283
4.18	0.2.0	284
4.18.1	New features and enhancements	284
4.18.2	Bugs fixed	284
4.19	0.1.0	285
5	Frequently asked questions (FAQ)	287
5.1	Errors and other unexpected behaviour	287
5.1.1	I'm getting an error about hybrid reference files when trying to sign / validate a file. What gives?	287
5.1.2	Why am I getting path building errors?	287
6	Known issues	289
7	Licenses	291
7.1	pyHanko License	291
7.2	Original PyPDF2 license	291
8	Indices and tables	293
	Python Module Index	295
	Index	297

PyHanko is a tool for signing and stamping PDF files.

CLI USER'S GUIDE

This guide offers a high-level overview of pyHanko as a command-line tool.

(Under construction)

If you installed pyHanko using `pip`, you should be able to invoke pyHanko using the `pyhanko` command, like so:

```
pyhanko --help
```

If the `pyhanko` package is on your `PYTHONPATH` but the `pyhanko` executable isn't on your `PATH` for whatever reason, you can also invoke the CLI through

```
python -m pyhanko --help
```

This guide will adopt the former calling convention.

You can run `pyhanko` in verbose mode by passing the `--verbose` flag before specifying the subcommand to invoke.

```
pyhanko --verbose <subcommand>
```

Note: The CLI portion of pyHanko was implemented using [Click](#). In particular, this means that it comes with a built-in help function, which can be accessed through `pyhanko --help`.

Caution: The pyHanko CLI makes heavy use of Click's subcommand functionality. Due to the way this works, the precise position of a command-line parameter sometimes matters. In general, double-dash options (e.g. `--option`) should appear after the subcommand to which they apply, but before the next one.

Right now, the pyHanko CLI offers two subcommand groups, for *sign* and *stamp*, respectively. Additional configuration options are available in an optional YAML *config file*.

Warning: This guide assumes that pyHanko is installed with all optional dependencies, including those required for PKCS#11 support and image support.

1.1 Signing PDF files

Signing PDF files using pyHanko can be very simple or somewhat complicated, depending on the specific requirements of your use case. PyHanko offers support for both visible and invisible signatures, several baseline PAdES profiles, seed values, and creating signatures using PKCS#11 devices.

1.1.1 Some background on PDF signatures

In order to properly understand the way pyHanko operates, having some background on the way PDF signatures work is useful. The goal of this subsection is to provide a bird's eye view, and covers only the bare minimum. For further details, please refer to the relevant sections of the ISO 32000 standard(s).

A PDF signature is always contained in a signature *field* in the PDF's form structure. Freeware PDF readers that do not have form editing functionality will typically not allow you to manipulate signature fields directly, but might allow you to fill existing form fields with a signature, or create a signature together with its corresponding form field. Using pyHanko, you can both insert new (empty) signature fields, and fill in existing ones.

Separate from the signature field containing it, a signature may or may not have an *appearance* associated with it. Signatures without such an appearance are referred to as *invisible* signatures. Invisible signatures have the advantage of being comparatively simpler to implement and configure, but when a PDF containing an invisible signature is opened in a reader application without signature support, it may not be visually obvious that the PDF file even contains a signature at all.

The signature object itself contains some PDF-specific metadata, such as

- the byte range of the file that it covers;
- the hash function used to compute the document hash to be signed;
- a modification policy that indicates the ways in which the file can still be modified.

The actual cryptographic signature is embedded as a CMS object. General CMS objects are defined in [RFC 5652](#), but only a limited subset is meaningful in PDF. When creating a signature, the signer is authenticated using the private key associated with an X.509 certificate, as issued by most common PKI authorities nowadays. The precise way this private key is provisioned is immaterial: it can be read from a file on disk, or the signature can be generated by a hardware token; this has no impact on the structure of the signature object in the file.

In a typical signed PDF file with only one signature, the signed byte range covers the entire document, except for the area containing the actual CMS data of the signature. However, there are a number of legitimate reasons why this may *not* be the case:

- documents containing multiple signatures and/or timestamps;
- signatures that allow further modification, such as form filling or annotation.

Generally speaking, the signer decides what modifications are still permitted after a signature is made¹.

The cryptographically informed reader might ask how it is *at all* possible to modify a file without invalidating the signature. After all, hash functions are supposed to prevent exactly this kind of thing. The answer here lies in the *incremental update* feature of the PDF standard. The specification allows for updating files by appending data to the end of the file, keeping the original bytes in place. These incremental update sections can create and modify existing objects in the file, while still preserving the original version in some form. Such changes are typically opaque to the user that views the file. The byte range attached to the signature ensures that the document hash can still be computed over the original data, and thus the integrity of the signature can still be validated.

However, since incremental updates allow the final rendered document to be modified in essentially arbitrary ways, the onus is on the *validator* to ensure that all such incremental updates made after a signature was created actually

¹ There are some legitimate modifications that cannot be prohibited by any document modification policy, such as the addition of document timestamps and updates to the document security store.

are “legitimate” changes. What precisely constitutes a “legitimate” change depends on the signature’s modification policy, but is not rigorously defined in the standard². It goes without saying that this has led to various [exploits](#) where PDF readers could be duped into allowing illicit modifications to signed PDF files without raising suspicion. As a consequence of this, some signature validation tools do not even bother to do any such validation, and simply reject *all* signatures in documents that have been modified through incremental updates.

See [Validating PDF signatures](#) for an overview of pyHanko’s signature validation features.

Note: By default, pyHanko uses incremental updates for all operations, regardless of the presence of earlier signatures in the file.

1.1.2 Creating signature fields

Adding new (empty) signature fields is done through the `addfields` subcommand of `pyhanko sign`. The CLI only allows you to specify the page and coordinates of the field, but more advanced properties and metadata can be manipulated through the API.

The syntax of the `addfields` subcommand is as follows:

```
pyhanko sign addfields --field PAGE/X1,Y1,X2,Y2/NAME input.pdf output.pdf
```

The page numbering starts at 1, and the numbers specify the coordinates of two opposing corners of the bounding box of the signature field. The coordinates are Cartesian, i.e. the y-coordinate increases from bottom to top. Multiple signature fields may be created in one command, by passing the last argument multiple times.

Note: You can specify page numbers “in reverse” by providing a negative number for the PAGE entry. With this convention, page -1 refers to the last page of the document, page -2 the second-to-last, etc.

Note: Creating empty signature fields ahead of time isn’t always necessary. PyHanko’s signing functionality can also create them together with a signature, and Adobe Reader offers similar conveniences. As such, this feature is mainly useful to create fields for other people to sign.

1.1.3 Creating simple signatures

All operations relating to digital signatures are performed using the `pyhanko sign` subcommand. The relevant command group for adding signatures is `pyhanko sign addsig`.

Warning: The commands explained in this subsection do not attempt to validate the signer’s certificate by default. You’ll have to take care of that yourself, either through your PDF reader of choice, or the [validation functionality](#) in *pyHanko*.

² The author has it on good authority that a rigorous incremental update validation specification is beyond the scope of the PDF standard itself.

Signing a PDF file using key material on disk

There are two ways to sign a PDF file using a key and a certificate stored on disk. The signing is performed in the exact same way in either case, but the format in which the key material is stored differs somewhat.

To sign a file with key material sourced from loose PEM or DER-encoded files, the `pemder` subcommand is used.

```
pyhanko sign addsig --field Sig1 pemder \  
  --key key.pem --cert cert.pem input.pdf output.pdf
```

This would create a signature in `input.pdf` in the signature field `Sig1` (which will be created if it doesn't exist), with a private key loaded from `key.pem`, and a corresponding certificate loaded from `cert.pem`. The result is then saved to `output.pdf`. Note that the `--field` parameter is optional if the input file contains a single unfilled signature field.

Note: The `--field` parameter also accepts parameters of the form passed to `addfields`, see [Creating signature fields](#).

You will be prompted for a passphrase to unlock the private key, which can be read from another file using `--passfile`.

The same result can be obtained using data from a PKCS#12 file (these usually have a `.pfx` or `.p12` extension) as follows:

```
pyhanko sign addsig --field Sig1 pkcs12 \  
  input.pdf output.pdf secrets.pfx
```

By default, these calls create invisible signature fields, but if the field specified using the `--field` parameter exists and has a widget associated with it, a simple default appearance will be generated (see [Fig. 1.1](#)).

In many cases, you may want to embed extra certificates (e.g. for intermediate certificate authorities) into your signature, to facilitate validation. This can be accomplished using the `--chain` flag to either subcommand. When using the `pkcs12` subcommand, pyHanko will automatically embed any extra certificates found in the PKCS#12 archive passed in.

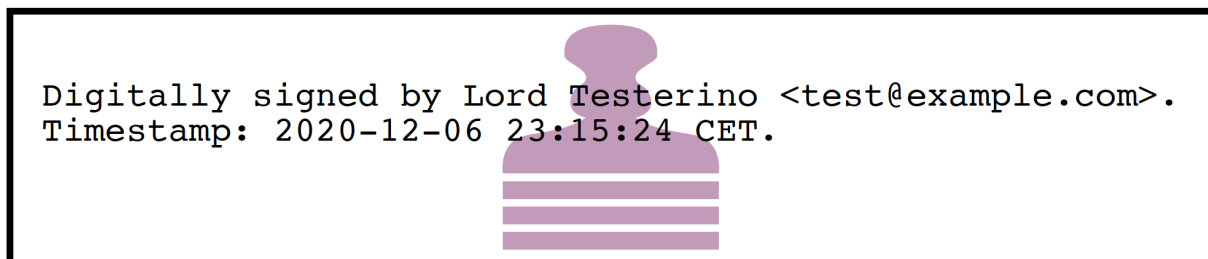


Fig. 1.1: The default appearance of a (visible) signature in pyHanko.

Signing a PDF file using a PKCS#11 token

PyHanko also supports creating signatures using PKCS#11 devices. In order to do so, you'll need the following information:

- The path to the PKCS#11 module, which is typically a shared object library (.so, .dll or .dylib, depending on your operating system)
- The label of the PKCS#11 token you're accessing (unless the token selection criteria are specified in the configuration file).
- The PKCS#11 label(s) of the certificate and key you're using, stored in the token. If the key and certificate labels are the same, you can omit the key label.

Most of these settings can be stored in the configuration file as well, see *Named PKCS#11 setups*. In fact, there are quite a few advanced settings that are not exposed as command-line switches, but can be specified in the configuration file. These include selecting tokens by serial number and selecting keys and certificates by ID.

With this information, producing a basic signature isn't very hard:

```
pyhanko sign addsig pkcs11 --lib /path/to/module.so \
--token-label testrsa --cert-label signer document.pdf output.pdf
```

Have a look at `pyhanko sign addsig pkcs11 --help` for a full list of options.

Signing a PDF file using a Belgian eID card

To sign a PDF file using your eID card, use the `beid` subcommand to `addsig`, with the `--lib` parameter to tell pyHanko where to look for the eID PKCS#11 library.

Note: Of course, you can also use the `pkcs11` subcommand, but `beid` provides an extra layer of convenience.

On Linux, it is named `libbeidpkcs11.so` and can usually be found under `/usr/lib` or `/usr/local/lib`. On macOS, it is named `libbeidpkcs11.dylib`, and can similarly be found under `/usr/local/lib`. The Windows version is typically installed to `C:\Windows\System32` and is called `beidpkcs11.dll`.

On Linux, this boils down to the following:

```
pyhanko sign addsig --field Sig1 beid \
--lib /path/to/libbeidpkcs11.so input.pdf output.pdf
```

On all platforms, the eID middleware will prompt you to enter your PIN to create the signature.

Warning: This command will produce a non-repudiable signature using the 'Signature' certificate on your eID card (as opposed to the 'Authentication' certificate). These signatures are legally equivalent to a normal "wet" signature wherever they are allowed, so use them with care.

In particular, you should only allow software you trust³ to use the 'Signature' certificate!

³ This obviously also applies to pyHanko itself; be aware that pyHanko's *license* doesn't make any fitness-for-purpose guarantees, so making sure you know what you're running is 100% your own responsibility.

Warning: You should also be aware that your national registry number (rijksregisternummer, no. de registre national) is embedded into the metadata of the signature certificate on your eID card⁴. As such, it can also be **read off from any digital signature you create**. While national registry numbers aren't secret per se, they are nevertheless often considered sensitive personal information, so you may want to be careful where you send documents containing your eID signature or that of someone else.

1.1.4 Creating signatures with long lifetimes

Background

A simple PDF signature—or any CMS signature for that matter—is only cryptographically valid insofar as the certificate of the signer is valid. In most common trust models, this means that the signature ceases to be meaningful together with the expiration of the signer certificate, or the latter's revocation.

The principal reason for this is the fact that it is no longer practical to verify whether a certificate was valid at the time of signing, if validation happens after the certificate already expired or was revoked. This, in turn, has to do with the fact that it is not always reasonable for certificate authorities to publicly supply historical validity proofs for all certificates they ever signed at all possible points in time.

Hence, in order for a signature to remain valid long after signing, the signer needs to supply two additional pieces of data:

1. a trusted timestamp signed by a time stamping authority (TSA), to prove the time of signing to the validator;
2. revocation information (relevant CRLs or OCSP responses) for all certificates in the chain of trust of the signer's certificate, and of the TSA.

For both of these, it is crucial that the relevant data is collected at the time of signing and embedded into the signed document. The revocation information in particular can be delicate, since the validator needs to be able to verify the validity of not only the signer's certificate, but also that of all issuers in the chain of trust, the OCSP responder's certificates used to sign the embedded OCSP responses, etc.

Time stamp tokens are commonly obtained from TSA's via the HTTP-based protocol specified in [RFC 3161](#).

Within the PDF standard, there are two broad categories of such long-lived signatures.

- Signers can opt to embed revocation information into the CMS data structure of the signature, as a signed attribute.
 - In this case, the revocation info is a signed attribute, protected from tampering by the signer's own signature.
 - This scheme uses Adobe-specific extensions to the CMS standard, which are explicitly defined in the PDF specification, but may not be supported by generic CMS tools that are unaware of PDF.
- Signers can opt to embed revocation information into the Document Security Store (DSS).
 - In this case the revocation info is (a priori) not protected by a signature, although this is often remedied by appending a document time stamp after updating the DSS (see also [Long-term archival \(LTA\) needs](#)).
 - The above approach has the convenient side effect that it can be used to 'fix' non-LTV-enabled signatures by embedding the required revocation information after the fact, together with a document timestamp. Obviously, this is predicated on the certificate's still being valid when the revocation information is compiled. This workflow is not guaranteed to be acceptable in all X.509 validation models, but is supported in pyHanko through the `ltvfix` subcommand; see [Adding validation data to an existing signature](#).
 - This approach is used in the PAdES baseline profiles B-LT and B-LTA defined by ETSI, and the (mildly modified) versions subsumed into ISO 32000-2 (PDF 2.0). As such, it is not part of ISO 32000-1 'proper'.

⁴ The certificate's serial number is in fact equal to the holder's national registry number.

Note: The author generally prefers the DSS-based signature profiles over the legacy approach based on CMS attributes, but both are supported in pyHanko.

Timestamps in pyHanko

Embedding a timestamp token into a signature using pyHanko is as simple as passing the `--timestamp-url` parameter to `addsig`. The URL should resolve to an endpoint that responds to the HTTP-based protocol described in [RFC 3161](#).

```
pyhanko sign addsig --field Sig1 --timestamp-url http://tsa.example.com \
  pemder --key key.pem --cert cert.pem input.pdf output.pdf
```

Warning: In the CLI, only public time stamping servers are supported right now (i.e. those that do not require authentication). The API is more flexible.

Embedding revocation info with pyHanko

In order to embed validation info, use the `--with-validation-info` flag to the `addsig` command.

```
pyhanko sign addsig --field Sig1 --timestamp-url http://tsa.example.com \
  --with-validation-info --use-pades pemder \
  --key key.pem --cert cert.pem input.pdf output.pdf
```

This will validate the signer's signature, and embed the necessary revocation information into the signature. The resulting signature complies with the PAdES B-LT baseline profile. If you want to embed the revocation data into the CMS object instead of the document security store (see above), leave off the `--use-pades` flag.

Using the `--trust`, `--trust-replace` and `--other-certs` parameters, it is possible to fine tune the validation context that will be used to embed the validation data. You can also predefine validation contexts in the configuration file, and select them using the `--validation-context` parameter. See [Named validation contexts](#) for further information.

Warning: By default, pyHanko requires signer certificates to have the non-repudiation key usage extension bit set on signer certificates. If this is not suitable for your use case, take a look at [Key usage settings](#).

Long-term archival (LTA) needs

The observant reader may have noticed that embedding revocation information together with a timestamp merely *shifts* the validation problem: what if the TSA certificate used to sign the timestamp token is already expired by the time we try to validate the signature?

The PAdES B-LTA scheme provides a solution for this issue: by appending a new document timestamp whenever the most recent one comes close to expiring, we can produce a chain of timestamps that allows us to ensure the validity of both the signatures and their corresponding revocation data essentially indefinitely.

This does, however, require 'active' maintenance of the document. PyHanko provides for this through the `ltaupdate` subcommand of `pyhanko sign`.

```
pyhanko sign ltaupdate --timestamp-url http://tsa.example.com input.pdf
```

Note that `ltaupdate` modifies files in-place. It is also unnecessary to provide a field name for the new timestamp; the software will automatically generate one using Python’s `uuid` module.

Warning: It is important to note that pyHanko only validates the outermost timestamp when performing an LTA update. This means that the “garbage in, garbage out” principle is in effect: if the timestamp chain was already broken elsewhere in the input document, running `ltaupdate` will not detect that, let alone fix it.

Note: The reader may also wonder what happens if the trust anchor that guaranteed the signer’s certificate at the time of signing happens to expire. Answering this question is technically beyond the specifications of the PKI system, since root certificates are trusted by fiat, and (by definition) do not have some higher authority backing them to enforce their validity constraints.

Some hold the view that expiration dates on trust anchors should be taken as mere suggestions rather than hard cutoffs. Regardless of the merits of this view in general, for the purposes of point-in-time validation, the only sensible answer seems to be to leave this judgment call up to the discretion of the validator.

It is also useful to note that some certificate authorities implement key rollover by cross-signing their new roots with their old roots and vice-versa. Provided these cross-signed certificates are available to the validator, these should allow older chains of trust to be validated against the newer roots.

1.1.5 Customising signature appearances

To a limited degree, the appearance of a visible signature made with pyHanko can be customised. You can specify a named style using the `--style-name` parameter to `addsig`:

```
pyhanko sign addsig --field Sig1 --style-name mystyle pemder \  
  --key key.pem --cert cert.pem input.pdf output.pdf
```

This assumes that a style named `mystyle` is available in the configuration file. Defining styles works the same way as pyHanko’s stamping functionality; see *Stamping PDF files* and *Styles for stamping and signature appearances* for details.

1.2 Validating PDF signatures

1.2.1 Basic use

Validating signatures in a PDF file is done through the `validate` subcommand of `pyhanko sign`.

A simple use case might look like this:

```
pyhanko sign validate --pretty-print document.pdf
```

This will print a human-readable overview of the validity status of the signatures in `document.pdf`. The trust setup can be configured using the *same command-line parameters* and *configuration options* as for creating LTV signatures.

Warning: By default, pyHanko requires signer certificates to have the non-repudiation key usage extension bit set on signer certificates. If this is not suitable for your use case, take a look at *Key usage settings*.

1.2.2 Factors in play when validating a signature

In this subsection, we go over the various factors considered by pyHanko when evaluating the validity of a PDF signature.

Cryptographic integrity

The most fundamental aspect of any digital signature: verify that the bytes of the file covered by the signature produce the correct hash value, and that the signature object is a valid signature of that hash. By ‘valid’, we mean that the cryptographic signature should be verifiable using the public key in the certificate that is marked as the signer’s in the signature object. In other words, we need to check that the *purported* signer’s certificate actually produced the signature.

Authenticity: trust settings

Having verified that the signature was produced by the (claimed) signer’s certificate, we next have to validate the binding between the certificate and its owner. That is to say, we have to convince ourselves that the entity whose name is on the certificate is in control of the private key, i.e. that the signer is who they claim to be.

Technically, this is done by establishing a *chain of trust* to a trust anchor, which we rely on to judge the validity of cryptographic identity claims. This is where the *trust settings* mentioned above come into play.

Incremental updates: difference analysis

PDF files can be modified, even when signed, by appending data to the end of the previous revision. These are *incremental updates*. In particular, this is how forms with multiple signatures are implemented in PDF. These incremental updates can essentially modify the original document in arbitrary ways, which is a problem, since they are (by definition) not covered by any earlier signatures.

In short, validators have two options: either reject all incremental updates (and decline to support multiple-signer scenarios of any kind), or police incremental updates by itself. The exact way in which this is supposed to be done is not specified precisely in the PDF standard.

Warning: PyHanko attempts to run a difference analysis on incremental updates, and processes modifications on a reject-by-default basis (i.e. all updates that can’t be vetted as OK are considered suspect). However, this feature is (very) experimental, and shouldn’t be relied on too much.

Establishing the time of signing

There are a number of ways to indicate when a signature was made. These broadly fall into two categories:

- Self-reported timestamps: those are based on the signer’s word, and shouldn’t necessarily be trusted as accurate.
- Trusted timestamps: these derive from timestamp tokens issued by a trusted timestamping authority at the time of signing.

Especially in the context of long-term verifiability of signatures and preventing things like backdating of documents, having an accurate measure of when the timestamp was made can be of crucial importance. PyHanko will tell you when a signature includes a timestamp token, and validate it along with the signature.

Note: Strictly speaking, a timestamp token only provides proof that the signature existed when the timestamp token was created. The signature itself may have been generated long before that!

If you also need a “lower bound” on the signing time, you might want to look into signed content timestamps (see `ades_signed_attr_spec` and [timestamp_content](#)).

Right now, pyHanko supports these when signing, but does not take them into account in the validation process. They are also not available in the CLI yet.

Evaluating seed value constraints

Finally, the document author can put certain restrictions on future signatures when setting up the form fields. These are known as *seed values* in the PDF standard. Not all seed values represent constraints (some are intended as suggestions), but one especially useful use of them is to earmark signature fields for use by specific signers. When validating signatures, pyHanko will also report on whether (mandatory) seed value constraints were respected.

Warning: Not all digital signing software is capable of processing seed values, so some false positives are to be expected.

Obviously, seed value constraints are only *truly* reliable if the document author secures the document with a certification signature before sending it for signing. Otherwise, later signers can modify the seed values *before* putting their signatures in place. See [here](#) for other concerns to keep in mind when relying on seed values.

Warning: PyHanko currently does *not* offer validation of structural PAdES profile requirements, in the sense that it can't tell you if a signature complies with all the provisions required by a particular PAdES profile. Note that these are requirements on the signature itself, and have no bearing on possible later modifications to the document.

Adding validation data to an existing signature

Sometimes, the validation data on a signature that was meant to have a long lifetime can be incomplete. This can have many causes, ranging from implementation problems to simple, temporary network issues.

To remedy this problem, pyHanko can fetch and append current validation information through the `ltvfix` command.

```
pyhanko sign ltvfix --field Sig1 document.pdf
```

The `ltvfix` command supports the same arguments as `validate` to select a validation context and specify trust settings.

Warning: By default, pyHanko's point-in-time validation requires OCSP responses and CRLs to be valid at the time of signing. This is often problematic when revocation information is added after the fact.

To emulate the default behaviour of Acrobat and other PDF viewers, use the `--retroactive-revinfo` switch when validating. This will cause pyHanko to treat CRLs and OCSP responses as valid infinitely far back into the past.

Note: This *will* cause incorrect behaviour when validating signatures backed by CAs that make use of certificate holds, but given that content timestamps (i.e. timestamps proving that a signature was created *after* some given time) aren't accounted for in pyHanko's trust model, this is somewhat unavoidable for the time being.

1.3 Stamping PDF files

Besides signing, pyHanko can also apply its signature appearance styles as stamps to a PDF file. Essentially, this renders a small overlay on top of the existing PDF content, without involving any of the signing logic.

Warning: The usefulness of this feature is currently rather limited, since visual stamp styles are still quite primitive. Additionally, the current version of pyHanko’s CLI doesn’t make it easy to take advantage of the customisation features available in the API.

The basic syntax of a stamping command is the following:

```
pyhanko stamp --style-name some-style --page 2 input.pdf output.pdf 50 100
```

This will render a stamp in the named style `some-style` at coordinates `(50, 100)` on the second page of `input.pdf`, and write the output to `output.pdf`. For details on how to define named styles, see [Styles for stamping and signature appearances](#).

Note: In terms of rendering, there is one important difference between signatures and stamps: stamps added through the CLI are rendered at their “natural” size/aspect ratio, while signature appearances need to fit inside the predefined box of their corresponding form field widget. This may cause unexpected behaviour.

1.4 Configuration options

1.4.1 Config file location

PyHanko reads its configuration from a YAML file. By default, if a file named `pyhanko.yml` exists in the current directory, pyHanko will attempt to read and process it. You can manually specify a configuration file location via the `--config` parameter to `pyhanko`.

Note that a configuration file is usually not required, although some of pyHanko’s behaviour cannot be fully customised using command line options. In these cases, the configuration must be sourced from a config file.

1.4.2 Configuration options

Logging options

Under the `logging` key in the configuration file, you can set up the configuration for Python’s logging module. Here’s an example.

```
logging:
  root-level: ERROR
  root-output: stderr
  by-module:
    pyhanko_certvalidator:
      level: DEBUG
      output: pyhanko_certvalidator.log
    pyhanko.sign:
      level: DEBUG
```

The keys `root-level` and `root-output` allow you to set the log level and the output stream (respectively) for the root logger. The default log level is `INFO`, and the default output stream is `stderr`. The keys under `by-module` allow you to specify more granular per-module logging configuration. The `level` key is mandatory in this case.

Note: If `pyhanko` is invoked with `--verbose`, the root logger will have its log level set to `DEBUG`, irrespective of the value specified in the configuration.

Named validation contexts

Validation contexts can be configured under the `validation-contexts` top-level key. The example below defines two validation configs named `default` and `special-setup`, respectively:

```
validation-contexts:
  default:
    other-certs: some-cert.pem.cert
  special-setup:
    trust: customca.pem.cert
    trust-replace: true
    other-certs: some-cert.pem.cert
```

The parameters are the same as those used to define validation contexts in the CLI. This is how they are interpreted:

- `trust`: One or more paths to trust anchor(s) to be used.
- `trust-replace`: Flag indicating whether the `trust` setting should override the system trust (default `false`).
- `other-certs`: One or more paths to other certificate(s) that may be needed to validate an end entity certificate.

The certificates should be specified in DER or PEM-encoded form. Currently, `pyHanko` can only read trust information from files on disk, not from other sources.

Selecting a named validation context from the CLI can be done using the `--validation-context` parameter. Applied to the example from [here](#), this is how it works:

```
pyhanko sign addsig --field Sig1 --timestamp-url http://tsa.example.com \
  --with-validation-info --validation-context special-setup \
  --use-pades pemder --key key.pem --cert cert.pem input.pdf output.pdf
```

In general, you're free to choose whichever names you like. However, if a validation context named `default` exists in the configuration file, it will be used implicitly if `--validation-context` is absent. You can override the name of the default validation context using the `default-validation-context` top-level key, like so:

```
default-validation-context: setup-a
validation-contexts:
  setup-a:
    trust: customca.pem.cert
    trust-replace: true
    other-certs: some-cert.pem.cert
  setup-b:
    trust: customca.pem.cert
    trust-replace: false
```

Time drift tolerance

Changed in version 0.5.0: Allow overriding the global value locally.

By default, pyHanko allows a drift of 10 seconds when comparing times. This value can be overridden in two ways: using the top-level `time-tolerance` configuration option, or by setting `time-tolerance` in a *named validation context*.

Given the example config below, using `setup-a` would set the time drift tolerance to 180 seconds. Since the global `time-tolerance` setting is set to 30 seconds, this value would be used with `setup-b`, or with any trust settings specified on the command line.

```
time-tolerance: 30
validation-contexts:
  setup-a:
    time-tolerance: 180
    trust: customca.pem.cert
    trust-replace: true
    other-certs: some-cert.pem.cert
  setup-b:
    trust: customca.pem.cert
    trust-replace: false
```

Allow revocation information to apply retroactively

New in version 0.5.0.

By default, `pyhanko-certvalidator` applies OCSP and CRL validity windows very strictly. For an OCSP response or a CRL to be considered valid, the validation time must fall within this window. In other words, with the default settings, an OCSP response fetched at some later date does not count for the purposes of establishing the revocation status of a certificate used with an earlier signature. However, pyHanko's conservative default position is often more strict than what's practically useful, so this behaviour can be overridden with a configuration setting (or the `--retroactive-revinfo` command line flag).

In the example config below, `retroactive-revinfo` is set to `true` globally, but to `false` in `setup-a` specifically. In either case, the `--retroactive-revinfo` flag can override this setting.

```
retroactive-revinfo: true
validation-contexts:
  setup-a:
    retroactive-revinfo: false
    trust: customca.pem.cert
    trust-replace: true
    other-certs: some-cert.pem.cert
  setup-b:
    trust: customca.pem.cert
    trust-replace: false
```

Named PKCS#11 setups

New in version 0.7.0.

Since the CLI parameters for signing files with a PKCS#11 token can get quite verbose, you might want to put the parameters in the configuration file. You can declare named PKCS#11 setups under the `pkcs11-setups` top-level key in pyHanko's configuration. Here's a minimal example:

```
pkcs11-setups:
  test-setup:
    module-path: /usr/lib/libsofthsm2.so
    token-criteria:
      label: testrsa
    cert-label: signer
```

If you need to, you can also put the user PIN right in the configuration:

```
pkcs11-setups:
  test-setup:
    module-path: /usr/lib/libsofthsm2.so
    token-criteria:
      label: testrsa
    cert-label: signer
    user-pin: 1234
```

Danger: If you do this, you should obviously take care to keep your configuration file in a safe place.

To use a named PKCS#11 configuration from the command line, invoke pyHanko like this:

```
pyhanko sign addsig pkcs11 --p11-setup test-setup input.pdf output.pdf
```

Named PKCS#11 setups also allow you to access certain advanced features that otherwise aren't available from the CLI directly. Here is an example.

```
pkcs11-setups:
  test-setup:
    module-path: /path/to/module.so
    token-criteria:
      serial: 17aa21784b9f
    cert-id: 1382391af78ac390
    key-id: 1382391af78ac390
```

This configuration will select a token based on the serial number instead of the label, and use PKCS#11 object IDs to select the certificate and the private key. All of these are represented as hex strings.

For a full overview of the parameters you can set on a PKCS#11 configuration, see the API reference documentation for [*PKCS11SignatureConfig*](#).

Note: Using the `--p11-setup` argument to `pkcs11` will cause pyHanko to ignore all other parameters to the `pkcs11` subcommand. In other words, you have to put everything in the configuration.

Named setups for on-disk key material

New in version 0.8.0.

Starting from version 0.8.0, you can also put parameters for on-disk key material into the configuration file in much the same way as for PKCS#11 tokens (see [Named PKCS#11 setups](#) above). This is done using the `pkcs12-setups` and `pemder-setups` top-level keys, depending on whether the key material is made available as a PKCS#12 file, or as individual PEM/DER-encoded files.

Here are some examples.

```
pkcs12-setups:
  foo:
    pfx-file: path/to/signer.pfx
    other-certs: path/to/more/certs.chain.pem
pemder-setups:
  bar:
    key-file: path/to/signer.key.pem
    cert-file: path/to/signer.cert.pem
    other-certs: path/to/more/certs.chain.pem
```

For non-interactive use, you can also put the passphrase into the configuration file (again, take care to set up your file access permissions correctly).

```
pkcs12-setups:
  foo:
    pfx-file: path/to/signer.pfx
    other-certs: path/to/more/certs.chain.pem
    pfx-passphrase: secret
pemder-setups:
  bar:
    key-file: path/to/signer.key.pem
    cert-file: path/to/signer.cert.pem
    other-certs: path/to/more/certs.chain.pem
    key-passphrase: secret
```

On the command line, you can use these named setups like this:

```
pyhanko sign addsig pkcs12 --p12-setup foo input.pdf output.pdf
pyhanko sign addsig pemder --pemder-setup bar input.pdf output.pdf
```

For a full overview of the parameters you can set in these configuration dictionaries, see the API reference documentation for [PKCS12SignatureConfig](#) and [PemDerSignatureConfig](#).

Key usage settings

New in version 0.5.0.

There are two additional keys that can be added to a named validation context: `signer-key-usage` and `signer-extd-key-usage`. Both either take a string argument, or an array of strings. These define the necessary key usage (resp. extended key usage) extensions that need to be present in signer certificates. For `signer-key-usage`, the possible values are as follows:

- `digital_signature`
- `non_repudiation`

- `key_encipherment`
- `data_encipherment`
- `key_agreement`
- `key_cert_sign`
- `crl_sign`
- `encipher_only`
- `decipher_only`

We refer to § 4.2.1.3 in [RFC 5280](#) for an explanation of what these values mean. By default, pyHanko requires signer certificates to have at least the `non_repudiation` extension, but you may want to change that depending on your requirements.

Values for extended key usage extensions can be specified as human-readable names, or as OIDs. The human-readable names are derived from the names in `asn1crypto.x509.KeyPurposeId` in `asn1crypto`. If you need a key usage extension that doesn't appear in the list, you can specify it as a dotted OID value instead. By default, pyHanko does not require any specific extended key usage extensions to be present on the signer's certificate.

This is an example showcasing key usage settings for a validation context named `setup-a`:

```
validation-contexts:
  setup-a:
    trust: customca.pem.cert
    trust-replace: true
    other-certs: some-cert.pem.cert
    signer-key-usage: ["digital_signature", "non_repudiation"]
    signer-extd-key-usage: ["code_signing", "2.999"]
```

Note: These key usage settings are mainly intended for use with validation, but are also checked when signing with an active validation context.

Styles for stamping and signature appearances

In order to use a style other than the default for a PDF stamp or (visible) signature, you'll have to write some configuration. New styles can be defined under the `stamp-styles` top-level key. Here are some examples:

```
stamp-styles:
  default:
    type: text
    background: __stamp__
    stamp-text: "Signed by %(signer)s\nTimestamp: %(ts)s"
    text-box-style:
      font: NotoSerif-Regular.otf
  noto-qr:
    type: qr
    background: background.png
    stamp-text: "Signed by %(signer)s\nTimestamp: %(ts)s\n%(url)s"
    text-box-style:
      font: NotoSerif-Regular.otf
      leading: 13
```


To select a named style at runtime, pass the `--style-name` parameter to `addsig` (when signing) or `stamp` (when stamping). As was the case for validation contexts, the style named `default` will be chosen if the `--style-name` parameter is absent. Similarly, the default style's name can be overridden using the `default-stamp-style` top-level key.

Let us now briefly go over the configuration parameters in the above example. All parameters have sane defaults.

- **type:** This can be either `text` or `qr`, for a simple text box or a stamp with a QR code, respectively. The default is `text`. Note that QR stamps require the `--stamp-url` parameter on the command line.
- **background:** Here, you can specify any of the following:
 - a path to a bitmap image;
 - a path to a PDF file (the first page will be used as the stamp background);
 - the special value `__stamp__`, which will render a simplified version of the pyHanko logo in the background of the stamp (using PDF graphics operators directly).

When using bitmap images, any file format natively supported by [Pillow](#) should be OK. If not specified, the stamp will not have a background.

- **stamp-text:** A template string that will be used to render the text inside the stamp's text box. Currently, the following variables can be used:
 - `signer`: the signer's name (only for signatures);
 - `ts`: the time of signing/stamping;
 - `url`: the URL associated with the stamp (only for QR stamps).
- **text-box-style:** With this parameter, you can fine-tune the text box's style parameters. The most important one is `font`, which allows you to specify an OTF font that will be used to render the text. If not specified, pyHanko will use a standard monospaced Courier font. See [TextBoxStyle](#) in the API reference for other customisable parameters.

The parameters used in the example styles shown above are not the only ones. The *dynamic configuration mechanism* used by pyHanko automatically exposes virtually all styling settings that are available to users of the (high-level) library API. For example, to use a stamp style where the text box is shifted to the right, and the background image is displayed on the left with custom margins, you could write something like the following:

```
stamp-styles:
  more-complex-demo:
    type: text
    stamp-text: "Test Test Test\n%(ts)s"
    background: image.png
    background-opacity: 1
    background-layout:
      x-align: left
      margins:
        left: 10
        top: 10
        bottom: 10
    inner-content-layout:
      x-align: right
      margins:
        right: 10
```

These settings are documented in the API reference documentation for [BaseStampStyle](#) and its subclasses.

Note: In general, the following rules apply when working with these “autoconfigurable” classes from within YAML.

- Underscores in field names (at the Python level) can be replaced with hyphens in YAML.
 - Some fields will in turn be of an autoconfigurable type, e.g. `background_layout` is a `SimpleBoxLayoutRule`, which can also be configured using a YAML dictionary (as shown in the example above).
 - In other cases, custom logic is provided to initialise certain fields, which is then documented on the (overridden) `process_entries()` method of the relevant class.
-

LIBRARY (SDK) USER'S GUIDE

This guide offers a high-level overview of pyHanko as a Python library. For the API reference docs generated from the source, see the [API reference](#).

The pyHanko library roughly consists of the following components.

- The `pyhanko.pdf_utils` package, which is essentially a (gutted and heavily modified) fork of PyPDF2, with various additions to support the kind of low-level operations that pyHanko needs to support its various signing and validation workflows.
- The `pyhanko.sign` package, which implements the general signature API supplied by pyHanko.
- The `pyhanko.stamp` module, which implements the signature appearance rendering & stamping functionality.
- Support modules to handle CLI and configuration: `pyhanko.config` and `pyhanko.cli`. These mostly consist of very thin wrappers around library functionality, and shouldn't really be considered public API.

2.1 Reading and writing PDF files

Note: This page only describes the read/write functionality of the `pdf_utils` package. See [The pdf-utils package](#) for further information.

2.1.1 Reading files

Opening PDF files for reading and writing in pyHanko is easy.

For example, to instantiate a `PdfFileReader` reading from `document.pdf`, it suffices to do the following.

```
from pyhanko.pdf_utils.reader import PdfFileReader

with open('document.pdf', 'rb') as doc:
    r = PdfFileReader(doc)
    # ... do stuff ...
```

In-memory data can be read in a similar way: if `buf` is a bytes object containing data from a PDF file, you can use it in a `PdfFileReader` as follows.

```
from pyhanko.pdf_utils.reader import PdfFileReader
from io import BytesIO
```

(continues on next page)

(continued from previous page)

```
buf = b'<PDF file data goes here>'
doc = BytesIO(buf)
r = PdfFileReader(doc)
# ... do stuff ...
```

2.1.2 Modifying files

If you want to modify a PDF file, use *IncrementalPdfFileWriter*, like so.

```
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter

with open('document.pdf', 'rb+') as doc:
    w = IncrementalPdfFileWriter(doc)
    # ... do stuff ...
    w.write_in_place()
```

Using *write_in_place()* will cause the generated update to be appended to the same stream as the input stream; this is why we open the file with 'rb+'. If you want the output to be written to a different file or buffer, use *write()* instead. Obviously, opening the input file with 'rb' is sufficient in this case.

Note: Due to the way PDF signing works, pyHanko's signing API will usually take care of calling *write* or *write_in_place* as appropriate, and do its own processing of the results. In most standard use cases, you probably don't need to worry about explicit writes too much.

Any *IncrementalPdfFileWriter* objects used in a signing operation should be discarded afterwards. If you want to continue appending updates to a signed document, create a new *IncrementalPdfFileWriter* on top of the output.

This should suffice to get you started with pyHanko's signing and validation functionality, but the reader/writer classes can do a lot more. To learn more about the inner workings of the low-level PDF manipulation layer of the library, take a look at *The pdf-utils package* or *the API reference*.

Warning: While the `pyhanko.pdf_utils` module is very powerful in that it allows you to modify objects in the PDF file in essentially arbitrary ways, and with a lot of control over the output, actually using it in this way requires some degree of familiarity with the PDF standard.

As things are now, pyHanko does *not* offer any facilities to help you format documents neatly, or to do any kind of layout work beyond the most basic operations. This may or may not change in the future. In the meantime, you're probably better off using typesetting software or a HTML to PDF converter for your more complex layout needs, and let pyHanko handle the signing step at the end.

2.2 Signature fields

The creation of signature fields—that is to say, *containers* for (future) signatures—is handled by the `pyhanko.sign.fields` module. Depending on your requirements, you may not need to call the functions in this module explicitly; in many simple cases, pyHanko’s *signing functionality* takes care of that for you.

However, if you want more control, or you need some of the more advanced functionality (such as seed value support or field locking) that the PDF standard offers, you might want to read on.

2.2.1 General API design

In general terms, a signature field is described by a `SigFieldSpec` object, which is passed to the `append_signature_field()` function for inclusion in a PDF file.

As the name suggests, a `SigFieldSpec` is a specification for a new signature field. These objects are designed to be immutable and stateless. A `SigFieldSpec` object is instantiated by calling `SigFieldSpec()` with the following keyword parameters.

- `sig_field_name`: the field’s name. This is the only mandatory parameter; it must not contain any period (.) characters.
- `on_page` and `box`: determine the position and page at which the signature field’s widget should be put (see *Positioning*).
- `seed_value_dict`: specify the seed value settings for the signature field (see *Seed value settings*).
- `field_mdp_spec` and `doc_mdp_update_value`: specify a template for the modification and field locking policy that the signer should apply (see *Document modification policy settings*).

Hence, to create a signature field specification for an invisible signature field named `Sig1`, and add it to a file document .pdf, you would proceed as follows.

```
from pyhanko.sign.fields import SigFieldSpec, append_signature_field
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter

with open('document.pdf', 'rb+') as doc:
    w = IncrementalPdfFileWriter(doc)
    append_signature_field(w, SigFieldSpec(sig_field_name="Sig1"))
    w.write_in_place()
```

2.2.2 Positioning

The position of a signature field is essentially only relevant for visible signatures. The following `SigFieldSpec` parameters determine where a signature widget will end up:

- `on_page`: index of the page on which the signature field should appear (default: 0);
- `box`: bounding box of the signature field, represented as a 4-tuple (`x1`, `y1`, `x2`, `y2`) in Cartesian coordinates (i.e. the vertical axis runs bottom to top).

Caution: In contrast with the CLI, pages are zero-indexed in the API.

2.2.3 Seed value settings

The PDF standard provides a way for document authors to provide so-called “seed values” for signature fields. These instruct the signer about the possible values for certain signature properties and metadata. They can be purely informative, but can also be used to restrict the signer in various ways.

Below is a non-exhaustive list of things that seed values can do.

- Put restrictions on the signer’s certificate, including
 - the issuer,
 - the subject’s distinguished name,
 - key usage extensions.
- Force the signer to embed a timestamp (together with a suggested time stamping server URL).
- Offer the signer a list of choices to choose from when selecting a reason for signing.
- Instruct the signer to use a particular signature (sub-)handler (e.g. tell the signer to produce PAdES-style signatures).

Most of these recommendations can be marked as mandatory using flags. In this case, they also introduce a validation burden.

Caution: Before deciding whether seed values are right for your use case, please consider the following factors.

1. Seed values are a (relatively) obscure feature of the PDF specification, and not all PDF software offers support for it. Using mandatory seed values is therefore probably only viable in a closed, controlled environment with well-defined document workflows. When using seed values in an advisory manner, you may want to provide alternative hints, perhaps in the form of written instructions in the document, or in the form of other metadata.
2. At this time, pyHanko only supports a subset of the seed value specification in the standard, but this should be resolved in due time. The extent of what is supported is recorded in the API reference for [SigSeedValFlags](#).
3. Since incremental updates can modify documents in arbitrary ways, mandatory seed values can only be (reliably) enforced if the author includes a certification signature, to prevent later signers from surreptitiously changing the rules.

If this is not an option for whatever reason, then you’ll have to make sure that the entity validating the signatures is aware of the restrictions the author intended through out-of-band means.

4. Consider whether using signatures with explicitly identified signature policies would be more appropriate (see e.g. [RFC 5126](#), § 5.8). Processing signature policies requires more specialised validation tools, but they are standardised much more rigorously than seed values in PDF. In particular, it is the superior choice when working with signatures in an AdES context. However, pyHanko’s support for these workflows is currently limited¹.

Seed values for a new signature field are configured through the `seed_value_dict` attribute of `SigFieldSpec`. This attribute takes a `SigSeedValueSpec` object, containing the desired seed value configuration. For a detailed overview of the seed values that can be specified, follow the links to the API reference; we only discuss the most important points below.

The mandatory seed values are indicated by the `flags` attribute, which takes a `SigSeedValFlags` object as its value. This is a subclass of `Flag`, so you can combine different flags using bitwise operations.

¹ Currently, pyHanko doesn’t yet support automatic enforcement of signature policies (to the extent that they can be machine-verified in the first place, obviously). This goes for both the signer and the validator. However, you can still *declare* signature policies by extending your favourite `Signer` subclass and adding the relevant signed attributes. Validators that do not support signature policy processing will typically ignore the policy setting altogether.

Restrictions and suggestions pertaining to the signer's certificate deserve special mention, since they're a bit special. These are encoded the `cert` attribute of `SigSeedValueSpec`, in the form of a `SigCertConstraints` object. This class has a `flags` attribute of its own, indicating which of the `SigCertConstraints` are to be enforced. Its value is a `SigCertConstraintFlags` object. In other words, the enforceability of certificate constraints is *not* controlled by the `flags` attribute of `SigSeedValueSpec`, but by the `flags` attribute of the `SigCertConstraints` object inside the `cert` attribute. This mirrors the way in which these restrictions are defined in the PDF specification.

Since this is all rather abstract, let's discuss a concrete example. The code below shows how you might instantiate a signature field specification for a ballot form of sorts, subject to the following requirements.

- Only people with voting rights should be able to sign the ballot. This is enforced by requiring that the certificates be issued by a specific certificate authority.
- The signer can either vote for or against the proposed measure, or abstain. For the sake of the example, let's encode that by one of three possible reasons for signing.
- Since we want to avoid cast ballots being modified after the fact, we require a strong hash function to be used (at least sha256).

```
from pyhanko.sign import fields
from pyhanko.sign.general import load_cert_from_pemder

franchising_ca = load_cert_from_pemder('path/to/certfile')
sv = fields.SigSeedValueSpec(
    reasons=[
        'I vote in favour of the proposed measure',
        'I vote against the proposed measure',
        'I formally abstain from voting on the proposed measure'
    ],
    cert=fields.SigCertConstraints(
        issuers=[franchising_ca],
        flags=fields.SigCertConstraintFlags.ISSUER
    ),
    digest_methods=['sha256', 'sha384', 'sha512'],
    flags=fields.SigSeedValFlags.REASONS | fields.SigSeedValFlags.DIGEST_METHOD
)

sp = fields.SigFieldSpec('BallotSignature', seed_value_dict=sv)
```

Note the use of the bitwise-or operator `|` to combine multiple flags.

2.2.4 Document modification policy settings

Broadly speaking, the PDF specification outlines two ways to specify the degree to which a document may be modified after a signature is applied, *without* these modifications affecting the validity of the signature.

- The **document modification detection policy** (DocMDP) is an integer between one and three, indicating on a document-wide level which classes of modification are permissible. The three levels are defined as follows:
 - level 1: no modifications are allowed;
 - level 2: form filling and signing are allowed;
 - level 3: form filling, signing and commenting are allowed.

The default value is 2.

- The **field modification detection policy** (FieldMDP), as the name suggests, specifies the form fields that can be modified after signing. FieldMDPs can be inclusive or exclusive, and as such allow fairly granular control.

When creating a signature field, the document author can suggest policies that the signer should apply in the signature object.

Warning: There are a number of caveats that apply to MDP settings in general; see [Some background on PDF signatures](#).

Traditionally, the DocMDP settings are exclusive to certification signatures (i.e. the first, specially marked signature included by the document author), but in PDF 2.0 it is possible for approval (counter)signatures to set the DocMDP level to a stricter value than the one already in force—although this uses a setting in the field’s locking dictionary rather than an explicit DocMDP dictionary on the signature itself.

In pyHanko, these settings are controlled by the `field_mdp_spec` and `doc_mdp_update_value` parameters of `SigFieldSpec`. The example below specifies a field with instructions for the signer to lock a field called `SomeTextField`, and set the DocMDP value for that signature to `FORM_FILLING` (i.e. level 2). PyHanko will respect these settings when signing, but other software might not.

```
from pyhanko.sign import fields

fields.SigFieldSpec(
    'Sig1', box=(10, 74, 140, 134),
    field_mdp_spec=fields.FieldMDPSpec(
        fields.FieldMDPAction.INCLUDE, fields=['SomeTextField']
    ),
    doc_mdp_update_value=fields.MDPPerm.FORM_FILLING
)
```

The `doc_mdp_update_value` value is more or less self-explanatory, since it’s little more than a numerical constant. The value passed to `field_mdp_spec` is an instance of `FieldMDPSpec`. `FieldMDPSpec` objects take two parameters:

- `fields`: The fields that are subject to the policy, which can be specified exclusively or inclusively, depending on the value of `action` (see below).
- `action`: This is an instance of the enum `FieldMDPAction`. The possible values are as follows.
 - `ALL`: all fields should be locked after signing. In this case, the value of the `fields` parameter is irrelevant.
 - `INCLUDE`: all fields specified in `fields` should be locked, while the others remain unlocked (in the absence of other more restrictive policies).
 - `EXCLUDE`: all fields *except* the ones specified in `fields` should be locked.

2.3 Signing functionality

This page describes pyHanko’s signing API.

Note: Before continuing, you may want to take a look at the [background on PDF signatures](#) in the CLI documentation.

2.3.1 General API design

The value entry (`/V`) of a signature field in a PDF file is given by a PDF dictionary: the “signature object”. This signature object in turn contains a `/Contents` key (a byte string) with a DER-encoded rendition of the CMS object (see [RFC 5652](#)) containing the actual cryptographic signature. To avoid confusion, the latter will be referred to as the “signature CMS object”, and we’ll reserve the term “signature object” for the PDF dictionary that is the value of the signature field.

The signature object contains a `/ByteRange` key outlining the bytes of the document that should be hashed to validate the signature. As a general rule, the hash of the PDF file used in the signature is computed over all bytes in the file, except those under the `/Contents` key. In particular, the `/ByteRange` key of the signature object is actually part of the signed data, which implies that the size of the signature CMS object needs to be estimated ahead of time. As we’ll see soon, this has some minor implications for the API design (see [this subsection](#) in particular).

The pyHanko signing API is spread across several modules in the `pyhanko.sign` package. Broadly speaking, it has three aspects:

- `PdfSignatureMetadata` specifies high-level metadata & structural requirements for the signature object and (to a lesser degree) the signature CMS object.
- `Signer` and its subclasses are responsible for the construction of the signature CMS object, but are in principle “PDF-agnostic”.
- `PdfSigner` is the “steering” class that invokes the `Signer` on an `IncrementalPdfFileWriter` and takes care of formatting the resulting signature object according to the specifications of a `PdfSignatureMetadata` object.

This summary, while a bit of an oversimplification, provides a decent enough picture of the separation of concerns in the signing API. In particular, the fact that construction of the CMS object is delegated to another class that doesn’t need to bother with any of the PDF-specific minutiae makes it relatively easy to support other signing technology (e.g. particular HSMs).

2.3.2 A simple example

Changed in version 0.9.0: New async-first API.

Virtually all parameters of `PdfSignatureMetadata` have sane defaults. The only exception is the one specifying the signature field to contain the signature—this parameter is always mandatory if the number of empty signature fields in the document isn’t exactly one.

In simple cases, signing a document can therefore be as easy as this:

```
from pyhanko.sign import signers
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter

cms_signer = signers.SimpleSigner.load(
    'path/to/signer/key.pem', 'path/to/signer/cert.pem',
    ca_chain_files=('path/to/relevant/certs.pem',),
    key_passphrase=b'secret'
)

with open('document.pdf', 'rb') as doc:
    w = IncrementalPdfFileWriter(doc)
    out = signers.sign_pdf(
        w, signers.PdfSignatureMetadata(field_name='Signature1'),
        signer=cms_signer,
```

(continues on next page)

(continued from previous page)

```
)

# do stuff with 'out'
# ...
```

The `sign_pdf()` function is a thin convenience wrapper around `PdfSigner`'s `sign_pdf()` method, with essentially the same API. The following code is more or less equivalent.

```
from pyhanko.sign import signers
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter

cms_signer = signers.SimpleSigner.load(
    'path/to/signer/key.pem', 'path/to/signer/cert.pem',
    ca_chain_files=('path/to/relevant/certs.pem',),
    key_passphrase=b'secret'
)

with open('document.pdf', 'rb') as doc:
    w = IncrementalPdfFileWriter(doc)
    out = signers.PdfSigner(
        signers.PdfSignatureMetadata(field_name='Signature1'),
        signer=cms_signer,
    ).sign_pdf(w)

# do stuff with 'out'
# ...
```

The advantages of instantiating the `PdfSigner` object yourself include reusability and more granular control over the signature's appearance.

In the above examples, `out` ends up containing a byte buffer (`io.BytesIO` object) with the signed output. You can control the output stream using the `output` or `in_place` parameters; see the documentation for `sign_pdf()`.

Danger: Any `IncrementalPdfFileWriter` used in the creation of a signature should be discarded afterwards. Further modifications would simply invalidate the signature anyway.

For a full description of the optional parameters, see the API reference documentation for `PdfSignatureMetadata` and `PdfSigner`.

Warning: If there is no signature field with the name specified in the `field_name` parameter of `PdfSignatureMetadata`, pyHanko will (by default) create an invisible signature field to contain the signature. This behaviour can be turned off using the `existing_fields_only` parameter to `sign_pdf()`, or you can supply a custom field spec when initialising the `PdfSigner`.

For more details on signature fields and how to create them, take a look at *Signature fields*.

Note that, from version 0.9.0 onwards, pyHanko can also be called asynchronously. In fact, this is now the preferred mode of invocation for most lower-level functionality. Anyway, the example from this section could have been written asynchronously as follows.

```

import asyncio
from pyhanko.sign import signers
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter

async def async_demo(signer, fname):
    with open(fname, 'rb') as doc:
        w = IncrementalPdfFileWriter(doc)
        out = await signers.async_sign_pdf(
            w, signers.PdfSignatureMetadata(field_name='Signature1'),
            signer=signer,
        )

    return out

cms_signer = signers.SimpleSigner.load(
    'path/to/signer/key.pem', 'path/to/signer/cert.pem',
    ca_chain_files=('path/to/relevant/certs.pem',),
    key_passphrase=b'secret'
)
asyncio.run(async_demo(cms_signer, 'document.pdf'))

```

For a signing process with *SimpleSigner* that doesn't perform any certificate validation, pyHanko's move towards a more async-focused API probably doesn't buy you all that much. However, using an asynchronous calling conventions allow for more efficient I/O when the signing code needs to access resources over a network. This typically becomes relevant when

- the cryptographic operations are performed by a remote signing service, or
- revocation info for the chain of trust needs to be embedded.

While you don't strictly *need* to use the new asynchronous APIs to reap all the benefits of this move, there are quite a few scenarios where it makes a lot of sense to do so, especially if your project is already structured around nonblocking/concurrent I/O operations.

2.3.3 Signature appearance generation

See also:

Styles for stamping and signature appearances in the CLI documentation for the CLI equivalent, and *Signature fields* for information on how to create signature fields in general.

When creating visible signatures, you can control the visual appearance to a degree, using different stamp types. This can be done in one of several ways.

Text-based stamps

PyHanko's standard stamp type is the *text stamp*. At its core, a text stamp appearance is simply some text in a box, possibly with interpolated parameters. Text stamps can use TrueType and OpenType fonts (or fall back to a generic monospaced font by default). Additionally, text stamps can also have backgrounds.

Text stamp styles are (unsurprisingly) described by a *TextStampStyle* object. Here's a code sample demonstrating basic usage, with some custom text using a TrueType font, and a bitmap background.

```
from pyhanko import stamp
from pyhanko.pdf_utils import text, images
from pyhanko.pdf_utils.font import opentype
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter
from pyhanko.sign import signers

signer = signers.SimpleSigner.load(...)
with open('document.pdf', 'rb') as inf:
    w = IncrementalPdfFileWriter(inf)
    fields.append_signature_field(
        w, sig_field_spec=fields.SigFieldSpec(
            'Signature', box=(200, 600, 400, 660)
        )
    )

    meta = signers.PdfSignatureMetadata(field_name='Signature')
    pdf_signer = signers.PdfSigner(
        meta, signer=signer, stamp_style=stamp.TextStampStyle(
            # the 'signer' and 'ts' parameters will be interpolated by pyHanko, if present
            stamp_text='This is custom text!\nSigned by: %(signer)s\nTime: %(ts)s',
            text_box_style=text.TextBoxStyle(
                font=opentype.GlyphAccumulatorFactory('path/to/NotoSans-Regular.ttf')
            ),
            background=images.PdfImage('stamp.png')
        ),
    )
    with open('document-signed.pdf', 'wb') as outf:
        pdf_signer.sign_pdf(w, output=outf)
```

Fig. 2.1 shows what the result might look like. Obviously, the final result will depend on the size of the bounding box, font properties, background size etc.

The layout of a text stamp can be tweaked to some degree, see *TextStampStyle*.

Note: You can define values for your own custom interpolation parameters using the *appearance_text_params* argument to *sign_pdf()*.



Fig. 2.1: A text stamp in Noto Sans Regular with an image background.

QR code stamps

Besides text stamps, pyHanko also supports signature appearances with a QR code embedded in them. Here's a variation of the previous example that leaves out the background, but includes a QR code in the end result.

```
from pyhanko import stamp
from pyhanko.pdf_utils import text
from pyhanko.pdf_utils.font import opentype
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter
from pyhanko.sign import signers

signer = signers.SimpleSigner.load(...)
with open('document.pdf', 'rb') as inf:
    w = IncrementalPdfFileWriter(inf)
    fields.append_signature_field(
        w, sig_field_spec=fields.SigFieldSpec(
            'Signature', box=(200, 600, 400, 660)
        )
    )

    meta = signers.PdfSignatureMetadata(field_name='Signature')
    pdf_signer = signers.PdfSigner(
        meta, signer=signer, stamp_style=stamp.QRStampStyle(
            # Let's include the URL in the stamp text as well
            stamp_text='Signed by: %(signer)s\nTime: %(ts)s\nURL: %(url)s',
            text_box_style=text.TextBoxStyle(
                font=opentype.GlyphAccumulatorFactory('path/to/NotoSans-Regular.ttf')
            ),
        ),
    )
    with open('document-signed.pdf', 'wb') as outf:
        # with QR stamps, the 'url' text parameter is special-cased and mandatory, even
        ↪ if it
        ↪ doesn't occur in the stamp text: this is because the value of the 'url'
        ↪ parameter is
        ↪ also used to render the QR code.
```

(continues on next page)

(continued from previous page)

```
pdf_signer.sign_pdf(
    w, output=outf,
    appearance_text_params={'url': 'https://example.com'}
)
```

Fig. 2.2 shows some possible output obtained with these settings.

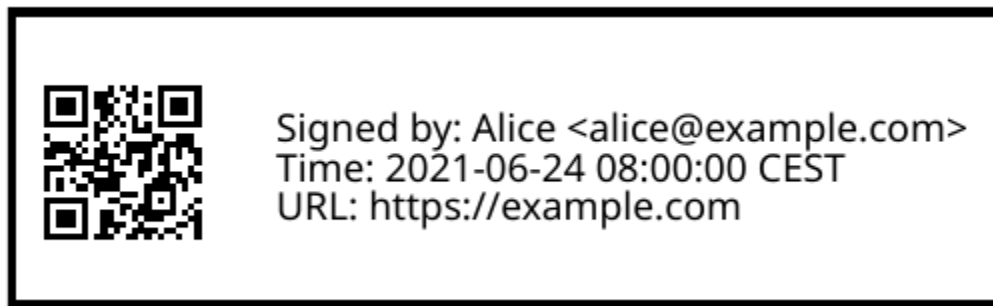


Fig. 2.2: A QR stamp in Noto Sans Regular, pointing to <https://example.com>

Static content stamps

PyHanko is mainly a signing library, and as such, its appearance generation code is fairly primitive. If you want to go beyond pyHanko's default signature appearances, you have the option to import an entire page from an external PDF file to use as the appearance, without anything else overlaid on top. Here's how that works.

```
from pyhanko import stamp
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter
from pyhanko.sign import fields, signers

signer = signers.SimpleSigner.load(...)
with open('document.pdf', 'rb') as inf:
    w = IncrementalPdfFileWriter(inf)
    fields.append_signature_field(
        w, sig_field_spec=fields.SigFieldSpec(
            'Signature', box=(200, 600, 400, 660)
        )
    )

    meta = signers.PdfSignatureMetadata(field_name='Signature')
    pdf_signer = signers.PdfSigner(
        meta, signer=signer,
        stamp_style=stamp.StaticStampStyle.from_pdf_file('my-fancy-appearance.pdf')
    )
    with open('document-signed.pdf', 'wb') as outf:
        pdf_signer.sign_pdf(w, output=outf)
```

The result of this snippet with a file from pyHanko's test suite is shown in Fig. 2.3. Essentially, this way of working allows you to use whatever tools you like to generate the signature appearance, and use the result with pyHanko's

signing tools. The bounding box of the content is derived from the imported page’s MediaBox (i.e. the principal page bounding box), so take that into account when designing your own appearances.

Note: The external PDF content is imported “natively”: all vector operations will remain vector operations, embedded fonts are copied over, etc. There is no rasterisation involved.



Fig. 2.3: Example of a signature appearance using a stamp imported from an external PDF file.

2.3.4 Timestamp handling

Cryptographic timestamps (specified by [RFC 3161](#)) play a role in PDF signatures in two different ways.

- They can be used as part of a PDF signature (embedded into the signature CMS object) to establish a (verifiable) record of the time of signing.
- They can also be used in a stand-alone way to provide document timestamps (PDF 2.0).

From a PDF syntax point of view, standalone document timestamps are formally very similar to PDF signatures. PyHanko implements these using the `timestamp_pdf()` method of `PdfTimeStamper`.

Timestamp tokens (TST) embedded into PDF signatures are arguably the more common occurrence. These function as countersignatures to the signer’s signature, proving that a signature existed at a certain point in time. This is a necessary condition for (most) long-term verifiability schemes.

Typically, such timestamp tokens are provided over HTTP, from a trusted time stamping authority (TSA), using the protocol specified in [RFC 3161](#). PyHanko provides a client for this protocol; see `HTTPTimeStamper`.

A `PdfSigner` can specify a default `TimeStamper` to procure timestamp tokens from some TSA, but sometimes pyHanko can infer a TSA endpoint from the signature field’s seed values.

The example from the previous section doesn’t need to be modified by a lot to include a trusted timestamp in the signature.

```
from pyhanko.sign import signers, timestamps
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter

cms_signer = signers.SimpleSigner.load(
    'path/to/signer/key.pem', 'path/to/signer/cert.pem',
    ca_chain_files=('path/to/relevant/certs.pem',),
```

(continues on next page)

(continued from previous page)

```

    key_passphrase=b'secret'
)

tst_client = timestamps.HTTPTimeStamper('http://example.com/tsa')

with open('document.pdf', 'rb') as doc:
    w = IncrementalPdfFileWriter(doc)
    out = signers.sign_pdf(
        w, signers.PdfSignatureMetadata(field_name='Signature1'),
        signer=cms_signer, timestamp=tst_client
    )

    # do stuff with 'out'
    # ...

```

As a general rule, pyHanko will attempt to obtain a timestamp token whenever a `TimeStamper` is available, but you may sometimes see more TST requests go over the wire than the number of signatures you’re creating. This is normal: since the timestamps are to be embedded into the signature CMS object of the signature, pyHanko needs a sample token to estimate the CMS object’s size². These “dummy tokens” are cached on the `TimeStamper`, so you can cut down on the number of such unnecessary requests by reusing the same `TimeStamper` for many signatures.

2.3.5 Creating PAdES signatures

Creating signatures conforming to various PAdES baseline profiles is also fairly straightforward using the pyHanko API.

To create a PAdES B-LTA signature, you can follow the template of the example below. This is the most advanced PAdES baseline profile. For other PAdES baseline profiles, tweak the parameters of the `PdfSignatureMetadata` object accordingly.

```

from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter
from pyhanko.sign import signers, timestamps
from pyhanko.sign.fields import SigSeedSubFilter
from pyhanko.certvalidator import ValidationContext

# Load signer key material from PKCS#12 file
# This assumes that any relevant intermediate certs are also included
# in the PKCS#12 file.
signer = signers.SimpleSigner.load_pkcs12(
    pfx_file='signer.pfx', passphrase=b'secret'
)

# Set up a timestamping client to fetch timestamps tokens
timestamp = timestamps.HTTPTimeStamper(
    url='http://tsa.example.com/timestampService'
)

# Settings for PAdES-LTA
signature_meta = signers.PdfSignatureMetadata(

```

(continues on next page)

² The size of a timestamp token is difficult to predict ahead of time, since it depends on many unknown factors, including the number & form of the various certificates that might come embedded within them.

(continued from previous page)

```

    field_name='Signature', md_algorithm='sha256',
    # Mark the signature as a PAdES signature
    subfilter=SigSeedSubFilter.PADES,
    # We'll also need a validation context
    # to fetch & embed revocation info.
    validation_context=ValidationContext(allow_fetching=True),
    # Embed relevant OCSP responses / CRLs (PAdES-LT)
    embed_validation_info=True,
    # Tell pyHanko to put in an extra DocumentTimeStamp
    # to kick off the PAdES-LTA timestamp chain.
    use_pades_lta=True
)

with open('input.pdf', 'rb') as inf:
    w = IncrementalPdfFileWriter(inf)
    with open('output.pdf', 'wb') as outf:
        signers.sign_pdf(
            w, signature_meta=signature_meta, signer=signer,
            timestamp=timestamp, output=outf
        )

```

2.3.6 Using aiohttp for network I/O

New in version 0.9.0.

In version 0.9.0, pyHanko’s lower-level APIs were reworked from an “async-first” perspective. For backwards compatibility reasons, the default implementation pyHanko’s network I/O code (for fetching revocation info, timestamps, etc.) still uses the `requests` library with some crude `asyncio` plumbing around it. However, to take maximal advantage of the new `asyncio` facilities, you need to use a networking library that actually supports asynchronous I/O natively. In principle, nothing stops you from plugging in an async-friendly library of your choosing, but pyHanko (and its dependency `pyhanko-certvalidator`) can already be used with `aiohttp` without much additional effort—`aiohttp` is a widely-used library for asynchronous HTTP.

Note: The reason why the `aiohttp` backend isn’t the default one is simple: using `aiohttp` requires the caller to manage a connection pool, which was impossible to properly retrofit into pyHanko without causing major breakage in the higher-level APIs as well.

Also note that `aiohttp` is an optional dependency.

Here’s an example demonstrating how you could use `aiohttp`-based networking in pyHanko to create a PAdES-B-LTA signature.

```

import aiohttp
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter
from pyhanko.sign import signers
from pyhanko.sign.fields import SigSeedSubFilter
from pyhanko.sign.timestamps.aiohttp_client import AIOHttpTimeStamper
from pyhanko_certvalidator import ValidationContext
from pyhanko_certvalidator.fetchers.aiohttp_fetchers \
    import AIOHttpFetcherBackend

```

(continues on next page)

(continued from previous page)

```

# Load signer key material from PKCS#12 file
# (see earlier examples)
signer = signers.SimpleSigner.load_pkcs12(
    pfx_file='signer.pfx', passphrase=b'secret'
)

# This demo async function takes an aiohttp session, an input
# file name and an output file name.
async def sign_doc_demo(session, input_file, output_file):
    # Use the aiohttp fetcher backend provided by pyhanko-certvalidator,
    # and tell it to use our client session.
    validation_context = ValidationContext(
        fetcher_backend=AIOHttpFetcherBackend(session),
        allow_fetching=True
    )

    # Similarly, we choose an RFC 3161 client implementation
    # that uses AIOHttp under the hood
    timestamp = AIOHttpTimeStamper(
        'http://tsa.example.com/timestampService',
        session=session
    )

    # The signing config is otherwise the same
    settings = signers.PdfSignatureMetadata(
        field_name='AsyncSignatureExample',
        validation_context=validation_context,
        subfilter=SigSeedSubFilter.PADES,
        embed_validation_info=True
    )

    with open(input_file, 'rb') as inf:
        w = IncrementalPdfFileWriter(inf)
        with open(output_file, 'wb') as outf:
            await signers.async_sign_pdf(
                w, settings, signer=signer, timestamp=timestamp,
                output=outf
            )

async def demo():
    # Set up our aiohttp session
    async with aiohttp.ClientSession() as session:
        await sign_doc_demo(session, 'input.pdf', 'output.pdf')

```

Note: Best practices for managing aiohttp sessions are beyond the scope of this guide. Have a look at [the documentation](#) for more information on how to use the aiohttp library effectively.

2.3.7 Extending Signer

Changed in version 0.9.0: New async-first API.

Providing detailed guidance on how to implement your own *Signer* subclass is beyond the scope of this guide—the implementations of *SimpleSigner* and *PKCS11Signer* should help. You might also want to take a look at *the AWS KMS example* on the *advanced examples page*. This subsection merely highlights some of the issues you should keep in mind.

First, if all you want to do is implement a signing device or technique that’s not supported by pyHanko, it should be sufficient to implement *async_sign_raw()*. This method computes the raw cryptographic signature of some data (typically a document hash) with the appropriate key material. It also takes a *dry_run* flag, signifying that the returned object should merely have the correct size, but the content doesn’t matter¹.

If your requirements necessitate further modifications to the structure of the CMS object, you’ll most likely have to override *async_sign()*, which is responsible for the construction of the CMS object itself.

2.3.8 The low-level PdfCMSEmbedder API

New in version 0.3.0.

Changed in version 0.7.0: Digest wrapped in *PreparedByteRangeDigest* in step 3; output returned in step 3 instead of step 4.

If even extending *Signer* doesn’t cover your use case (e.g. because you want to take the construction of the signature CMS object out of pyHanko’s hands entirely), all is not lost. The lowest-level “managed” API offered by pyHanko is the one provided by *PdfCMSEmbedder*. This class offers a coroutine-based interface that takes care of all PDF-specific operations, but otherwise gives you full control over what data ends up in the signature object’s */Contents* entry.

Note: *PdfSigner* uses *PdfCMSEmbedder* under the hood, so you’re still mostly using the same code paths with this API.

Danger: Some advanced features aren’t available this deep in the API (mainly seed value checking). Additionally, *PdfCMSEmbedder* doesn’t really do any input validation; you’re on your own in that regard. See also *Interrupted signing* for a more middle-of-the-road solution.

Here is an example demonstrating its use, sourced more or less directly from the test suite. For details, take a look at the API docs for *PdfCMSEmbedder*.

```
from datetime import datetime
from pyhanko.sign import signers
from pyhanko.sign.signers import cms_embedder
from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter

from io import BytesIO

input_buf = BytesIO(b'<input file goes here>')
w = IncrementalPdfFileWriter(input_buf)
```

(continues on next page)

¹ The *dry_run* flag is used in the estimation of the CMS object’s size. With key material held in memory it doesn’t really matter all that much, but if the signature is provided by a HSM, or requires additional input on the user’s end (such as a PIN), you typically don’t want to use the “real” signing method in dry-run mode.

(continued from previous page)

```

# Phase 1: coroutine sets up the form field, and returns a reference
cms_writer = cms_embedder.PdfCMSEmbedder().write_cms(
    field_name='Signature', writer=w
)
sig_field_ref = next(cms_writer)

# just for kicks, let's check
assert sig_field_ref.get_object()['/T'] == 'Signature'

# Phase 2: make a placeholder signature object,
# wrap it up together with the MDP config we want, and send that
# on to cms_writer
timestamp = datetime.now(tz=tzlocal.get_localzone())
sig_obj = signers.SignatureObject(timestamp=timestamp, bytes_reserved=8192)

md_algorithm = 'sha256'
# for demonstration purposes, let's do a certification signature instead
# of a plain old approval signature here
cms_writer.send(
    cms_embedder.SigObjSetup(
        sig_placeholder=sig_obj,
        mdp_setup=cms_embedder.SigMDPSetup(
            md_algorithm=md_algorithm, certify=True,
            docmdp_perms=fields.MDPPerm.NO_CHANGES
        )
    )
)

# Phase 3: write & hash the document (with placeholder)
prep_digest, output = cms_writer.send(
    cms_embedder.SigIOSetup(md_algorithm=md_algorithm, in_place=True)
)
# The `output` variable is a handle to the stream that contains
# the document to be signed, with a placeholder allocated to hold
# the actual signature contents.

# Phase 4: construct the CMS object, and pass it on to cms_writer

# NOTE: I'm using a regular SimpleSigner here, but you can substitute
# whatever CMS supplier you want.

signer: signers.SimpleSigner = FROM_CA
# let's supply the CMS object as a raw bytestring
cms_bytes = signer.sign(
    data_digest=prep_digest.document_digest,
    digest_algorithm=md_algorithm, timestamp=timestamp
).dump()
sig_contents = cms_writer.send(cms_bytes)

# The (signed) output document is in `output` now.
# `sig_contents` holds the content of the signature container
# in the PDF file, including any padding.

```

2.3.9 Interrupted signing

New in version 0.7.0.

Changed in version 0.9.0: The new async-first API requires some changes to the workflow at this (relatively low) level of abstraction.

Changed in version 0.14.0: It is no longer mandatory to make the signer’s certificate available from the start of the workflow, although this comes at the cost of some convenience (signature size estimation and revocation info collection being two major ones). This makes it easier to implement remote signing scenarios where the signer’s certificate is unknown until the remote signing service produces its response.

There are use cases where trying to run the entire signing process in one go isn’t feasible. Think of a remote signing scenario with pyHanko running on a server, and calling an external signing service to perform the cryptographic operations, or a case where pyHanko needs to wait for interactive user input to proceed with signing.

In cases like this, there are several points where you can interrupt the signing process partway through, save the state, and pick up where you left off some time later—this conserves valuable resources in some scenarios. We refer to [pyhanko.sign.signers.pdf_signer](#) for a full overview of what’s possible; below, we describe the most common use case: a scenario where pyHanko prepares a document for signing, computes the digest, sends it off to somewhere else for signing, and finishes the signing process once the response comes in (potentially in an entirely different thread).

In the example scenario, we use [ExternalSigner](#) to format the signed attributes and the final CMS object, but the same principle applies (*mutatis mutandis*) to remote signers that supply complete CMS objects.

```
from pyhanko.sign import signers, fields, timestamps
from pyhanko.sign.signers.pdf_signer import PdfTBSDocument
from pyhanko_certvalidator import ValidationContext
from pyhanko.pdf_utils.writer import BasePdfFileWriter

# Skeleton code for an interrupted PAdES signature

async def prep_document(w: BasePdfFileWriter):
    vc = ValidationContext(...)
    pdf_signer = signers.PdfSigner(
        signers.PdfSignatureMetadata(
            field_name='SigNew', embed_validation_info=True, use_pades_lta=True,
            subfilter=fields.SigSeedSubFilter.PADES,
            validation_context=vc,
            md_algorithm='sha256'
        ),
        # note: this signer will not perform any cryptographic operations,
        # it's just there to bundle certificates with the generated CMS
        # object and to provide size estimates
        signer=signers.ExternalSigner(
            signing_cert=..., ...,
            # placeholder value, appropriate for a 2048-bit RSA key
            # (for example's sake)
            signature_value=bytes(256),
        ),
        timestamp=timestamps.HTTPTimeStamper('http://tsa.example.com')
    )
    prep_digest, tbs_document, output = \
        await pdf_signer.async_digest_doc_for_signing(w)
    md_algorithm = tbs_document.md_algorithm
```

(continues on next page)

(continued from previous page)

```

psi = tbs_document.post_sign_instructions

signed_attrs = await ext_signer.signed_attrs(
    prep_digest.document_digest, 'sha256', use_pades=True
)
psi = tbs_document.post_sign_instructions
return prep_digest, signed_attrs, psi, output

# After prep_document finishes, you can serialise the contents
# of prep_digest, signed_attrs and psi somewhere.
# The output stream can also be stored in a temporary file, for example.
# You could now call the remote signing service, and once the response
# comes back, proceed with finish_signing() after deserialising
# all the intermediate outputs from the previous step.

async def finish_signing(sig_value: bytes, prep_digest, signed_attrs,
                        psi, output_handle):
    # Here, assume sig_value is the signed digest of the signed_attrs
    # bytes, obtained from some remote signing service

    # use ExternalSigner to format the CMS given the signed value
    # we obtained from the remote signing service
    ext_signer = instantiate_external_signer(sig_value)
    sig_cms = await ext_signer.async_sign_prescribed_attributes(
        'sha256', signed_attrs=signed_attrs,
        timestamp=DUMMY_HTTP_TS
    )

    validation_context = ValidationContext(...)
    await PdfTBSDocument.async_finish_signing(
        output_handle, prepared_digest=prep_digest,
        signature_cms=sig_cms,
        post_sign_instr=psi,
        validation_context=validation_context
    )

```

The above example below also showcases how to apply proper post-signature processing in an interrupted PAdES signature. This is only necessary for PAdES-LT and PAdES-LTA signatures. In other scenarios, you can replace the `async_finish_signing` call with the following one-liner:

```
prep_digest.fill_with_cms(output_handle, sig_cms)
```

In particular, you don't have to bother with *PostSignInstructions* at all.

Note that, starting with pyHanko 0.14.0, the signer's certificate need no longer be provided at the start of the signing process, if you supply some additional parameters yourself. Here's what that might look like in a toy example.

```

w = IncrementalPdfFileWriter(pdf_file_handle)
pdf_signer = signers.PdfSigner(
    # Specifying a digest algorithm (or signature mechanism)
    # is necessary if the signing cert is not available
    signers.PdfSignatureMetadata(
        field_name='Signature',

```

(continues on next page)

(continued from previous page)

```

    ),
    signer=ExternalSigner(
        # note the 'None's
        signing_cert=None, cert_registry=None,
        signature_value=256,
    )
)

# Since estimation is disabled without a certificate
# available, bytes_reserved becomes mandatory.
prep_digest, tbs_document, output = await pdf_signer\
    .async_digest_doc_for_signing(w, bytes_reserved=8192)

# Call the external service
# note: the signing certificate is in the returned payload,
# but we don't (necessarily) need to do anything with it.
signature_container = \
    await call_external_service(prepare_digest.document_digest)

# Note: in the meantime, we could've serialised and deserialised
# the contents of 'output', of course
await PdfTBSDocument.async_finish_signing(output, prep_digest)

# If you want, you can now proceed to tack on additional revisions
# with revocation information, document timestamps and the like.

```

2.3.10 Generic data signing

New in version 0.7.0.

Changed in version 0.9.0: New `async-first` API.

If you need to produce CMS signatures that are not intended to be consumed as traditional PDF signatures (for whatever reason), the *Signer* classes in pyHanko expose a more flexible API that you can use.

The *Signer* class's `async_sign_general_data()` method is a fairly thin wrapper around `async_sign()` that performs some of the bookkeeping operations on the payload being signed. It outputs a CMS object with essentially the same set of attributes that would be expected in a typical PDF signature, but the actual payload can be arbitrary data.

It can take either an IO-type object, or simply a bytes payload. For advanced uses (e.g. those requiring a custom-set *contentType*), passing in a `cms.ContentInfo` (or `cms.EncapsulatedContentInfo` object) also works. This has a number of caveats; carefully review the API documentation for `async_sign_general_data()` and section 5.1 of [RFC 5652](#) first.

The signer can operate in “detached” or “encapsulating” mode. In the former case, the payload being signed is not encoded as part of the resulting CMS object. When in doubt, use detached mode—it’s the default.

Here is an example showcasing a typical invocation, combined with a call to `embed_payload_with_cms()` to embed the resulting payload as a signed attachment in a PDF file.

```

from pyhanko.sign.signers.pdf_cms import SimpleSigner
from pyhanko.sign.signers.functions import embed_payload_with_cms
from pyhanko.pdf_utils import embed, writer

```

(continues on next page)

(continued from previous page)

```

async def demo():
    data = b'Hello world!'
    # instantiate a SimpleSigner
    sgn = SimpleSigner(...)
    # Sign some data
    signature = \
        await sign.async_sign_general_data(data, 'sha256', detached=False)

    # Embed the payload into a PDF file, with the signature
    # object as a related file.
    w = writer.PdfFileWriter() # fresh writer, for demonstration's sake
    embed_payload_with_cms(
        w, file_spec_string='attachment.txt',
        file_name='attachment.txt',
        payload=embed.EmbeddedFileObject.from_file_data(
            w, data=data, mime_type='text/plain',
        ),
        cms_obj=signature,
        file_spec_kwargs={'description': "Signed attachment test"}
    )

```

Warning: This way of signing attachments is not standard, and chances are that your PDF reader won't process the signature at all. This snippet is simply a demonstration of the general principle behind CMS signing, and doesn't really represent any particular PDF feature.

2.4 Validation functionality

Note: Before reading this, you may want to take a look at *Factors in play when validating a signature* for some background on the validation process.

Danger: In addition to the caveats outlined in *Validating PDF signatures*, you should be aware that the validation API is still very much in flux, and likely to change by the time pyHanko reaches its beta stage.

2.4.1 General API design

PyHanko's validation functionality resides in the *validation* module. Its most important components are

- the *EmbeddedPdfSignature* class (responsible for modelling existing signatures in PDF documents);
- the various subclasses of *SignatureStatus* (encoding the validity status of signatures and timestamps);
- *validate_pdf_signature()* and *validate_pdf_ltv_signature()*, for running the actual validation logic.
- the *DocumentSecurityStore* class and surrounding auxiliary classes (responsible for handling DSS updates in documents).

While you probably won't need to interface with `DocumentSecurityStore` directly, knowing a little about `EmbeddedPdfSignature` and `SignatureStatus` is useful.

2.4.2 Accessing signatures in a document

There is a convenience property on `PdfFileReader`, aptly named `embedded_signatures`. This property produces an array of `EmbeddedPdfSignature` objects, in the order that they were applied to the document. The result is cached on the reader object.

These objects can be used to inspect the signature manually, if necessary, but they are mainly intended to be used as input for `validate_pdf_signature()` and `validate_pdf_ltv_signature()`.

2.4.3 Validating a PDF signature

All validation in pyHanko is done with respect to a certain *validation context* (an object of type `pyhanko_certvalidator.ValidationContext`). This object tells pyHanko what the trusted certificates are, and transparently provides mechanisms to request and keep track of revocation data. For LTV validation purposes, a `ValidationContext` can also specify a point in time at which the validation should be carried out.

Warning: PyHanko currently uses a forked version of the `certvalidator` library, registered as `pyhanko-certvalidator` on PyPI. The changes in the forked version are minor, and the API is intended to be backwards-compatible with the “mainline” version.

The principal purpose of the `ValidationContext` is to let the user explicitly specify their own trust settings. However, it may be necessary to juggle several *different* validation contexts over the course of a validation operation. For example, when performing LTV validation, pyHanko will first validate the signature's timestamp against the user-specified validation context, and then build a new validation context relative to the signing time specified in the timestamp.

Here's a simple example to illustrate the process of validating a PDF signature w.r.t. a specific trust root.

```
from pyhanko.sign.general import load_cert_from_pemder
from pyhanko_certvalidator import ValidationContext
from pyhanko.pdf_utils.reader import PdfFileReader
from pyhanko.sign.validation import validate_pdf_signature

root_cert = load_cert_from_pemder('path/to/certfile')
vc = ValidationContext(trust_roots=[root_cert])

with open('document.pdf', 'rb') as doc:
    r = PdfFileReader(doc)
    sig = r.embedded_signatures[0]
    status = validate_pdf_signature(sig, vc)
    print(status.pretty_print_details())
```

2.4.4 Long-term verifiability checking

As explained [here](#) and [here](#) in the CLI documentation, making sure that PDF signatures remain verifiable over long time scales requires special care. Signatures that have this property are often called “LTV enabled”, where LTV is short for *long-term verifiable*.

To verify a LTV-enabled signature, you should use `validate_pdf_ltv_signature()` instead of `validate_pdf_signature()`. The API is essentially the same, but `validate_pdf_ltv_signature()` takes a required `validation_type` parameter. The `validation_type` is an instance of the enum `pyhanko.sign.validation.RevocationInfoValidationType` that tells pyHanko where to find and how to process the revocation data for the signature(s) involved¹. See the documentation for `pyhanko.sign.validation.RevocationInfoValidationType` for more information on the available profiles.

In the initial `ValidationContext` passed to `validate_pdf_ltv_signature()` via `bootstrap_validation_context`, you typically want to leave `moment` unset (i.e. verify the signature at the current time).

This is the validation context that will be used to establish the time of signing. When this step is done, pyHanko will construct a new validation context pointed towards that point in time. You can specify keyword arguments to the `ValidationContext` constructor using the `validation_context_kwargs` parameter of `validate_pdf_ltv_signature()`. In typical situations, you can leave the `bootstrap_validation_context` parameter off entirely, and let pyHanko construct an initial validation context using `validation_context_kwargs` as input.

The PAdES B-LTA validation example below should clarify that.

```
from pyhanko.sign.general import load_cert_from_pemder
from pyhanko.pdf_utils.reader import PdfFileReader
from pyhanko.sign.validation import (
    validate_pdf_ltv_signature, RevocationInfoValidationType
)

root_cert = load_cert_from_pemder('path/to/certfile')

with open('document.pdf', 'rb') as doc:
    r = PdfFileReader(doc)
    sig = r.embedded_signatures[0]
    status = validate_pdf_ltv_signature(
        sig, RevocationInfoValidationType.PAdES_LTA,
        validation_context_kwargs={'trust_roots': [root_cert]}
    )
    print(status.pretty_print_details())
```

Notice how, rather than passing a `ValidationContext` object directly, the example code only supplies `validation_context_kwargs`. These keyword arguments will be used both to construct an initial validation context (at the current time), and to construct any subsequent validation contexts for point-of-time validation once the signing time is known.

In the example, the `validation_context_kwargs` parameter ensures that all validation will happen w.r.t. one specific trust root.

If all this sounds confusing, that’s because it is. You may want to take a look at the source of `validate_pdf_ltv_signature()` and its tests, and/or play around a little.

¹ Currently, pyHanko can’t figure out by itself which LTV strategy is being used, so the caller has to specify it explicitly.

Warning: Even outside the LTV context, pyHanko always distinguishes between validation of the signing time and validation of the signature itself. In fact, `validate_pdf_signature()` reports both (see the docs for `timestamp_validity`).

However, since the LTV adjudication process is entirely moot without a trusted record of the signing time, `validate_pdf_ltv_signature()` will raise a `SignatureValidationError` if the timestamp token (or timestamp chain) fails to validate. Otherwise, `validate_pdf_ltv_signature()` returns a `PdfSignatureStatus` as usual.

2.4.5 Incremental update analysis

Changed in version 0.2.0: The initial ad-hoc approach was replaced by a more extensible and maintainable rule-based validation system. See `pyhanko.sign.diff_analysis`.

As explained in *the CLI documentation*, the PDF standard has provisions that allow files to be updated by appending so-called “incremental updates”. This also works for signed documents, since appending data does not destroy the cryptographic integrity of the signed data.

That being said, since incremental updates can change essentially any aspect of the resulting document, validators need to be careful to evaluate whether these updates were added for a legitimate reason. Examples of such legitimate reasons could include the following:

- adding a second signature,
- adding comments,
- filling in (part of) a form,
- updating document metadata,
- performing cryptographic “bookkeeping work” such as appending fresh document timestamps and/or revocation information to ensure the long-term verifiability of a signature.

Not all of these reasons are necessarily always valid: the signer can tell the validator which modifications they allow to go ahead without invalidating their signature. This can either be done through the “DocMDP” setting (see `MDPPerm`), or for form fields, more granularly using FieldMDP settings (see `FieldMDPSpec`).

That being said, the standard does not specify a concrete procedure for validating any of this. PyHanko takes a reject-by-default approach: the difference analysis tool uses rules to compare document revisions, and judge which object updating operations are legitimate (at a given `MDPPerm` level). Any modifications for which there is no justification invalidate the signature.

The default diff policy is defined in `DEFAULT_DIFF_POLICY`, but you can define your own, either by implementing your own subclass of `DiffPolicy`, or by defining your own rules and passing those to an instance of `StandardDiffPolicy`. `StandardDiffPolicy` takes care of some boilerplate for you, and is the mechanism backing `DEFAULT_DIFF_POLICY`. Explaining precisely how to implement custom diff rules is beyond the scope of this guide, but you can take a look at the source of the `diff_analysis` module for more information.

To actually use a custom diff policy, you can proceed as follows.

```
from pyhanko.sign.general import load_cert_from_pemder
from pyhanko.certvalidator import ValidationContext
from pyhanko.pdf_utils.reader import PdfFileReader
from pyhanko.sign.validation import validate_pdf_signature

from my_awesome_module import CustomDiffPolicy
```

(continues on next page)

(continued from previous page)

```

root_cert = load_cert_from_pemder('path/to/certfile')
vc = ValidationContext(trust_roots=[root_cert])

with open('document.pdf', 'rb') as doc:
    r = PdfFileReader(doc)
    sig = r.embedded_signatures[0]
    status = validate_pdf_signature(sig, vc, diff_policy=CustomDiffPolicy())
    print(status.pretty_print_details())

```

The `modification_level` and `docmdp_ok` attributes on `PdfSignatureStatus` will tell you to what degree the signed file has been modified after signing (according to the diff policy used).

Warning: The most lenient MDP level, `ANNOTATE`, is currently not supported by the default diff policy.

Danger: Due to the lack of standardisation when it comes to signature validation, correctly adjudicating incremental updates is inherently somewhat risky and ill-defined, so until pyHanko matures, you probably shouldn't rely on its judgments too heavily.

Should you run into unexpected results, by all means file an issue. All information helps!

If necessary, you can opt to turn off difference analysis altogether. This is sometimes a very reasonable thing to do, e.g. in the following cases:

- you don't trust pyHanko to correctly evaluate the changes;
- the (sometimes rather large) performance cost of doing the diff analysis is not worth the benefits;
- you need validate only one signature, after which the document shouldn't change at all.

In these cases, you might want to rely on the `coverage` property of `PdfSignatureStatus` instead. This property describes the degree to which a given signature covers a file, and is much cheaper/easier to compute.

Anyhow, to disable diff analysis completely, it suffices to pass the `skip_diff` parameter to `validate_pdf_signature()`.

```

from pyhanko.sign.general import load_cert_from_pemder
from pyhanko.certvalidator import ValidationContext
from pyhanko.pdf_utils.reader import PdfFileReader
from pyhanko.sign.validation import validate_pdf_signature

root_cert = load_cert_from_pemder('path/to/certfile')
vc = ValidationContext(trust_roots=[root_cert])

with open('document.pdf', 'rb') as doc:
    r = PdfFileReader(doc)
    sig = r.embedded_signatures[0]
    status = validate_pdf_signature(sig, vc, skip_diff=True)
    print(status.pretty_print_details())

```

2.4.6 Probing different aspects of the validity of a signature

The `PdfSignatureStatus` objects returned by `validate_pdf_signature()` and `validate_pdf_ltv_signature()` provide a fairly granular account of the validity of the signature.

You can print a human-readable validity report by calling `pretty_print_details()`, and if all you’re interested in is a yes/no judgment, use the `bottom_line` property.

Should you ever need to know more, a `PdfSignatureStatus` object also includes information on things like

- the certificates making up the chain of trust,
- the validity of the embedded timestamp token (if present),
- the invasiveness of incremental updates applied after signing,
- seed value constraint compliance.

For more information, take a look at `PdfSignatureStatus` in the API reference.

2.5 The pdf-utils package

The `pdf_utils` package is the part of pyHanko that implements the logic for reading & writing PDF files.

2.5.1 Background and future perspectives

The core of the `pdf_utils` package is based on code from PyPDF2. I forked/vendored PyPDF2 because it was the Python PDF library that would be the easiest to adapt to the low-level needs of a digital signing tool like pyHanko.

The “inherited” parts mostly consist of the PDF parsing logic, filter implementations (though they’ve been heavily rewritten) and RC4 cryptography support. I stripped out most of the functionality that I considered “fluff” for the purposes of designing a DigSig tool, for several reasons:

- When I started working on pyHanko, the PyPDF2 project was all but dead, the codebase largely untested and the internet was rife with complaints about all kinds of bugs. Removing code that I didn’t need served primarily as a way to reduce my maintenance burden, and to avoid attaching my name to potential bugs that I wasn’t willing to fix myself.
- PyPDF2 included a lot of compatibility logic to deal with Python 2. I never had any interest in supporting Python versions prior to 3.7, so I ditched all that.
- Stripping out unnecessary code left me with greater freedom to deviate from the PyPDF2 API where I considered it necessary to do so.

I may or may not split off the `pdf_utils` package into a fully-fledged Python PDF library at some point, but for now, it merely serves as pyHanko’s PDF toolbox. That said, if you need bare-bones access to PDF structures outside pyHanko’s digital signing context, you might find some use for it even in its current state.

This page is intended as a companion to the API reference for `pyhanko.pdf_utils`, rather than a detailed standalone guide.

Danger: For the reasons specified above, most of `pyhanko.pdf_utils` should be considered private API.

The internal data model for PDF objects isn’t particularly likely to change, but the text handling and layout code is rather primitive and immature, so I’m not willing to commit to freezing that API (yet).

Danger: There are a number of stream encoding schemes (or “filters”) that aren’t supported (yet), most notably the LZW compression scheme. Additionally, we don’t have support for all PNG predictors in the Flate decoder/encoder.

2.5.2 PDF object model

The `pyhanko.pdf_utils.generic` module maps PDF data structures to Python objects. PDF arrays, dictionaries and strings are largely interoperable with their native Python counterparts, and can (usually) be interfaced with in the same manner.

When dealing with indirect references, the package distinguishes between the following two kinds:

- *IndirectObject*: this represents an indirect reference as embedded into another PDF object (e.g. a dictionary value given by an indirect object);
- *Reference*: this class represents an indirect reference by itself, i.e. not as a PDF object.

This distinction is rarely relevant, but the fact that *IndirectObject* inherits from *PdfObject* means that it supports the `container_ref` API, which is meaningless for “bare” *Reference* objects.

As a general rule, use *Reference* whenever you’re using indirect objects as keys in a Python dictionary or collecting them into a set, but use *IndirectObject* if you’re writing indirect objects into PDF output.

2.5.3 PDF content abstractions

The `pyhanko.pdf_utils.content` module provides a fairly bare-bones abstraction for handling content that “compiles down” to PDF graphics operators, namely the *PdfContent* class. Among other things, it takes care of some of the PDF resource management boilerplate. It also allows you to easily encapsulate content into form XObjects when necessary.

Below, we briefly go over the uses of *PdfContent* within the library itself. These also serve as a template for implementing your own *PdfContent* subclasses.

Images

PyHanko relies on Pillow for image support. In particular, we currently support pretty much all RGB bitmap types that Pillow can handle. Other colour spaces are not (yet) available. Additionally, we currently don’t take advantage of PDF’s native JPEG support, or some of its more clever image compression techniques.

The `pyhanko.pdf_utils.images` module provides a *PdfContent* subclass (aptly named `pyhanko.pdf_utils.images.PdfImage`) as a convenience.

Text & layout

The layout code in pyHanko is currently very, very primitive, fragile and likely to change significantly going forward. That said, pyHanko can do some basic text box rendering, and is capable of embedding CID-keyed OTF fonts for use with CJK text, for example. Given the (for now) volatile state of the API, I won’t document it here, but you can take a look at `pyhanko.pdf_utils.text` and `pyhanko.pdf_utils.font`, or the code in `pyhanko.stamp`.

2.6 Advanced examples

2.6.1 A custom Signer to use AWS KMS asynchronously

New in version 0.9.0.

This example demonstrates how to use aioboto3 to set up a custom *Signer* implementation that invokes the *AWS KMS* API to sign documents, and does so in an asynchronous manner.

The example implementation is relatively minimal, but it should be sufficient to get an idea of what's possible. Further information on aioboto3 is available [from the project's GitHub page](#).

The ideas in this snippet can be combined with other async-native components to set up an asynchronous signing workflow. For example, if you're looking for a way to fetch & embed revocation information asynchronously, have a look at [this section in the signing docs](#) to learn more about aiohttp usage and resource management.

```
import asyncio

import aioboto3

from asn1crypto import x509, algos
from cryptography.hazmat.primitives import hashes

from pyhanko.pdf_utils.incremental_writer import IncrementalPdfFileWriter
from pyhanko.sign import Signer, signers
from pyhanko.sign.general import get_pyca_cryptography_hash, \
    load_cert_from_pemder
from pyhanko_certvalidator.registry import SimpleCertificateStore

class AsyncKMSSigner(Signer):

    def __init__(self, session: aioboto3.session, key_id: str,
                 signing_cert: x509.Certificate,
                 signature_mechanism: algos.SignedDigestAlgorithm,
                 # this can be derived from the above, obviously
                 signature_mechanism_aws_id: str,
                 other_certs=()):
        self.session = session
        self.signing_cert = signing_cert
        self.key_id = key_id
        self.signature_mechanism = signature_mechanism
        self.signature_mechanism_aws_id = signature_mechanism_aws_id
        self.cert_registry = cr = SimpleCertificateStore()
        cr.register_multiple(other_certs)
        super().__init__()

    async def async_sign_raw(self, data: bytes,
                             digest_algorithm: str, dry_run=False) -> bytes:
        if dry_run:
            return bytes(256)

        # Send hash to server instead of raw data
        hash_spec = get_pyca_cryptography_hash(
```

(continues on next page)

(continued from previous page)

```

        self.signature_mechanism.hash_algo
    )
    md = hashes.Hash(hash_spec)
    md.update(data)

    async with self.session.client('kms') as kms_client:
        result = await kms_client.sign(
            KeyId=self.key_id,
            Message=md.finalize(),
            MessageType='DIGEST',
            SigningAlgorithm=self.signature_mechanism_aws_id
        )
        signature = result['Signature']
        assert isinstance(signature, bytes)
        return signature

async def run():

    # Load relevant certificates
    # Note: the AWS KMS does not provide certificates by itself,
    # so the details of how certificates are provisioned are beyond
    # the scope of this example.
    cert = load_cert_from_pemder('path/to/your/signing-cert.pem')
    chain = list(load_certs_from_pemder('path/to/chain.pem'))

    # AWS credentials
    kms_key_id = "KEY_ID_GOES_HERE"
    aws_access_key_id = "ACCESS_KEY_GOES_HERE"
    aws_secret_access_key = "SECRET_GOES_HERE"

    # Set up aioboto3 session with provided credentials & region
    session = aioboto3.Session(
        aws_access_key_id=aws_access_key_id,
        aws_secret_access_key=aws_secret_access_key,
        # substitute your region here
        region_name='eu-central-1'
    )

    # Set up our signer
    signer = AsyncKMSSigner(
        session=session, key_id=kms_key_id,
        signing_cert=cert, other_certs=chain,
        # change the signature mechanism according to your key type
        # I'm using an ECDSA key over the NIST-P384 (secp384r1) curve here.
        signature_mechanism=algos.SignedDigestAlgorithm(
            {'algorithm': 'sha384_ecdsa'}
        ),
        signature_mechanism_aws_id='ECDSA_SHA_384'
    )

    with open('input.pdf', 'rb') as inf:

```

(continues on next page)

(continued from previous page)

```
w = IncrementalPdfFileWriter(inf)
meta = signers.PdfSignatureMetadata(
    field_name='AWSKMSEExampleSig'
)
with open('output.pdf', 'wb') as outf:
    await signers.async_sign_pdf(
        w, meta, signer=signer, output=outf
    )

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(run())
```


API REFERENCE

This is the API reference for pyHanko, compiled from the docstrings present in the Python source files. For a more high-level overview, see the [library user guide](#). If you are interested in using pyHanko as a command-line application, please refer to the [CLI user guide](#).

Warning: Any function, class or method that is *not* covered by this documentation is considered private API by definition.

Until pyHanko goes into beta, *any* part of the API is subject to change without notice, but this applies doubly to the undocumented parts. Tread with caution.

3.1 pyhanko package

3.1.1 Subpackages

pyhanko.pdf_utils package

pyhanko.pdf_utils.barcodes module

class pyhanko.pdf_utils.barcodes.**BarcodeBox**(*barcode_type*, *code*)

Bases: [PdfContent](#)

Thin wrapper around python-barcode functionality.

This will render a barcode of the specified type as PDF graphics operators.

render() → bytes

Compile the content to graphics operators.

class pyhanko.pdf_utils.barcodes.**PdfStreamBarcodeWriter**

Bases: [BaseWriter](#)

Implementation of writer class for the python-barcode library to output PDF graphics operators. Note: `_paint_text` is intentionally dummied out. Please use the functionality implemented in `pyhanko.pdf_utils.text` instead.

property `command_stream`: bytes

save(*filename*, *output*)

Saves the rendered output to *filename*.

Parameters**filename**

[String] Filename without extension.

output

[String] The rendered output.

Returns

The full filename with extension.

Return type

String

pyhanko.pdf_utils.config_utils module

This module contains utilities for allowing dataclasses to be populated by user-provided configuration (e.g. from a Yaml file).

Note: On naming conventions: this module converts hyphens in key names to underscores as a matter of course.

exception pyhanko.pdf_utils.config_utils.**ConfigurationError**

Bases: `ValueError`

Signal configuration errors.

class pyhanko.pdf_utils.config_utils.**ConfigurableMixin**

Bases: `object`

General configuration mixin for dataclasses

classmethod `process_entries(config_dict)`

Hook method that can modify the configuration dictionary to overwrite or tweak some of their values (e.g. to convert string parameters into more complex Python objects)

Subclasses that override this method should call `super().process_entries()`, and leave keys that they do not recognise untouched.

Parameters

config_dict – A dictionary containing configuration values.

Raises

`ConfigurationError` – when there is a problem processing a relevant entry.

classmethod `check_config_keys(keys_supplied: Set[str])`

Check whether all supplied keys are meaningful.

Parameters

keys_supplied – The keys supplied in the configuration.

Raises

`ConfigurationError` – if at least one key does not make sense.

classmethod `from_config(config_dict)`

Attempt to instantiate an object of the class on which it is called, by means of the configuration settings passed in.

First, we check that the keys supplied in the dictionary correspond to data fields on the current class. Then, the dictionary is processed using the `process_entries()` method. The resulting dictionary is passed to the initialiser of the current class as a kwargs dict.

Parameters

config_dict – A dictionary containing configuration values.

Returns

An instance of the class on which it is called.

Raises

ConfigurationError – when an unexpected configuration key is encountered or left unfilled, or when there is a problem processing one of the config values.

```
pyhanko.pdf_utils.config_utils.check_config_keys(config_name, expected_keys, supplied_keys)
```

```
pyhanko.pdf_utils.config_utils.process_oid(asn1crypto_class: Type[ObjectIdentifier], id_string,  
                                             param_name)
```

```
pyhanko.pdf_utils.config_utils.process_oids(asn1crypto_class: Type[ObjectIdentifier], strings,  
                                              param_name)
```

```
pyhanko.pdf_utils.config_utils.process_bit_string_flags(asn1crypto_class: Type[BitString], strings,  
                                                         param_name)
```

pyhanko.pdf_utils.content module

```
class pyhanko.pdf_utils.content.ResourceType(value)
```

Bases: Enum

Enum listing resources that can be used as keys in a resource dictionary.

See ISO 32000-1, § 7.8.3 Table 34.

```
EXT_G_STATE = '/ExtGState'
```

External graphics state specifications. See ISO 32000-1, § 8.4.5.

```
COLOR_SPACE = '/ColorSpace'
```

Colour space definitions. See ISO 32000-1, § 8.6.

```
PATTERN = '/Pattern'
```

Pattern definitions. See ISO 32000-1, § 8.7.

```
SHADING = '/Shading'
```

Shading definitions. See ISO 32000-1, § 8.7.4.3.

```
XOBJECT = '/XObject'
```

External object definitions (images and form XObjects). See ISO 32000-1, § 8.8.

```
FONT = '/Font'
```

Font specifications. See ISO 32000-1, § 9.

```
PROPERTIES = '/Properties'
```

Marked content properties. See ISO 32000-1, § 14.6.2.

```
exception pyhanko.pdf_utils.content.ResourceManagementError
```

Bases: ValueError

Used to signal problems with resource dictionaries.

class pyhanko.pdf_utils.content.PdfResources

Bases: object

Representation of a PDF resource dictionary.

This class implements `__getitem__()` with *ResourceType* keys for dynamic access to its attributes. To merge two instances of *PdfResources* into one another, the class overrides `__iadd__()`, so you can write.

```
res1 += res2
```

Note: Merging two resource dictionaries with conflicting resource names will produce a *ResourceManagementError*.

Note: This class is currently only used for new resource dictionaries.

as_pdf_object() → *DictionaryObject*

Render this instance of *PdfResources* to an actual resource dictionary.

class pyhanko.pdf_utils.content.PdfContent(*resources: Optional[PdfResources] = None, box: Optional[BoxConstraints] = None, writer=None*)

Bases: object

Abstract representation of part of a PDF content stream.

Warning: Whether *PdfContent* instances can be reused or not is left up to the subclasses.

writer = None

The `__init__()` method comes with an optional `writer` parameter that can be used to let subclasses register external resources with the writer by themselves.

It can also be set after the fact by calling *set_writer()*.

set_resource()(*category: ResourceType, name: NameObject, value: PdfObject*)

Set a value in the resource dictionary associated with this content fragment.

Parameters

- **category** – The resource category to which the resource belongs.
- **name** – The resource’s (internal) name.
- **value** – The resource’s value.

import_resources()(*resources: PdfResources*)

Import resources from another resource dictionary.

Parameters

resources – An instance of *PdfResources*.

Raises

ResourceManagementError – Raised when there is a resource name conflict.

property resources: *PdfResources*

Returns

The *PdfResources* instance associated with this content fragment.

render() → bytes

Compile the content to graphics operators.

as_form_xobject() → *StreamObject*

Render the object to a form XObject to be referenced by another content stream. See ISO 32000-1, § 8.8.

Note: Even if *writer* is set, the resulting form XObject will not be registered. This is left up to the caller.

Returns

A *StreamObject* instance representing the resulting form XObject.

set_writer(writer)

Override the currently registered writer object.

Parameters

writer – An instance of *BasePdfFileWriter*.

class `pyhanko.pdf_utils.content.RawContent`(*data: bytes, resources: Optional[PdfResources] = None, box: Optional[BoxConstraints] = None*)

Bases: *PdfContent*

Raw byte sequence to be used as PDF content.

render() → bytes

Compile the content to graphics operators.

class `pyhanko.pdf_utils.content.ImportedPdfPage`(*file_name, page_ix=0*)

Bases: *PdfContent*

Import a page from another PDF file (lazily)

render() → bytes

Compile the content to graphics operators.

pyhanko.pdf_utils.crypt package

Changed in version 0.13.0: Refactor `crypt` module into package.

Changed in version 0.3.0: Added support for PDF 2.0 encryption standards and crypt filters.

Utilities for PDF encryption. This module covers all methods outlined in the standard:

- Legacy RC4-based encryption (based on PyPDF2 code).
- AES-128 encryption with legacy key derivation (partly based on PyPDF2 code).
- PDF 2.0 AES-256 encryption.
- Public key encryption backed by any of the above.

Following the language in the standard, encryption operations are backed by subclasses of the *SecurityHandler* class, which provides a more or less generic API.

Danger: The members of this package are all considered internal API, and are therefore subject to change without notice.

Danger: One should also be aware that the legacy encryption scheme implemented here is (very) weak, and we only support it for compatibility reasons. Under no circumstances should it still be used to encrypt new files.

About crypt filters

Crypt filters are objects that handle encryption and decryption of streams and strings, either for all of them, or for a specific subset (e.g. streams representing embedded files). In the context of the PDF standard, crypt filters are a notion that only makes sense for security handlers of version 4 and up. In pyHanko, however, *all* encryption and decryption operations pass through crypt filters, and the serialisation/deserialisation logic in [SecurityHandler](#) and its subclasses transparently deals with staying backwards compatible with earlier revisions.

Internally, pyHanko loosely distinguishes between implicit and explicit uses of crypt filters:

- Explicit crypt filters are used by directly referring to them from the `/Filter` entry of a stream dictionary. These are invoked in the usual stream decoding process.
- Implicit crypt filters are set by the `/StmF` and `/StrF` entries in the security handler's crypt filter configuration, and are invoked by the object reading/writing procedures as necessary. These filters are invisible to the stream encoding/decoding process: the `encoded_data` attribute of an “implicitly encrypted” stream will therefore contain decrypted data ready to be decoded in the usual way.

As long as you don't require access to encoded object data and/or raw encrypted object data, this distinction should be irrelevant to you as an API user.

```
class pyhanko.pdf_utils.crypt.SecurityHandler(version: SecurityHandlerVersion, legacy_keylen,  
                                              crypt_filter_config: CryptFilterConfiguration,  
                                              encrypt_metadata=True, compat_entries=True)
```

Bases: object

Generic PDF security handler interface.

This class contains relatively little actual functionality, except for some common initialisation logic and book-keeping machinery to register security handler implementations.

Parameters

- **version** – Indicates the version of the security handler to use, as described in the specification. See [SecurityHandlerVersion](#).
- **legacy_keylen** – Key length in bytes (only relevant for legacy encryption handlers).
- **crypt_filter_config** – The crypt filter configuration for the security handler, in the form of a [CryptFilterConfiguration](#) object.

Note: PyHanko implements legacy security handlers (which, according to the standard, aren't crypt filter-aware) using crypt filters as well, even though they aren't serialised to the output file.

- **encrypt_metadata** – Flag indicating whether document (XMP) metadata is to be encrypted.

Warning: Currently, PyHanko does not manage metadata streams, so until that changes, it is the responsibility of the API user to mark metadata streams using the `/Identity` crypt filter as required.

Nonetheless, the value of this flag is required in key derivation computations, so the security handler needs to know about it.

- **compat_entries** – Write deprecated but technically unnecessary configuration settings for compatibility with certain implementations.

static register(cls: Type[SecurityHandler])

Register a security handler class. Intended to be used as a decorator on subclasses.

See [build\(\)](#) for further information.

Parameters

cls – A subclass of [SecurityHandler](#).

static build(encrypt_dict: DictionaryObject) → SecurityHandler

Instantiate an appropriate [SecurityHandler](#) from a PDF document’s encryption dictionary.

PyHanko will search the registry for a security handler with a name matching the `/Filter` entry. Failing that, a security handler implementing the protocol designated by the `/SubFilter` entry (see [support_generic_subfilters\(\)](#)) will be chosen.

Once an appropriate [SecurityHandler](#) subclass has been selected, pyHanko will invoke the subclass’s [instantiate_from_pdf_object\(\)](#) method with the original encryption dictionary as its argument.

Parameters

encrypt_dict – A PDF encryption dictionary.

Returns

classmethod get_name() → str

Retrieves the name of this security handler.

Returns

The name of this security handler.

extract_credential() → Optional[SerializableCredential]

Extract a serialisable credential for later use, if the security handler supports it. It should allow the security handler to be unlocked with the same access level as the current one.

Returns

A serialisable credential, or None.

classmethod support_generic_subfilters() → Set[str]

Indicates the generic `/SubFilter` values that this security handler supports.

Returns

A set of generic protocols (indicated in the `/SubFilter` entry of an encryption dictionary) that this [SecurityHandler](#) class implements. Defaults to the empty set.

classmethod instantiate_from_pdf_object(encrypt_dict: DictionaryObject)

Instantiate an object of this class using a PDF encryption dictionary as input.

Parameters

encrypt_dict – A PDF encryption dictionary.

Returns

is_authenticated() → bool

Return True if the security handler has been successfully authenticated against for document encryption purposes.

The default implementation just attempts to call [get_file_encryption_key\(\)](#) and returns True if that doesn’t raise an error.

as_pdf_object() → DictionaryObject

Serialise this security handler to a PDF encryption dictionary.

Returns

A PDF encryption dictionary.

authenticate(*credential*, *id1*=None) → *AuthResult*

Authenticate a credential holder with this security handler.

Parameters

- **credential** – A credential. The type of the credential is left up to the subclasses.
- **id1** – The first part of the document ID of the document being accessed.

Returns

An *AuthResult* object indicating the level of access obtained.

get_string_filter() → *CryptFilter*

Returns

The crypt filter responsible for decrypting strings for this security handler.

get_stream_filter(*name*=None) → *CryptFilter*

Parameters

name – Optionally specify a crypt filter by name.

Returns

The default crypt filter responsible for decrypting streams for this security handler, or the crypt filter named *name*, if not None.

get_embedded_file_filter()

Returns

The crypt filter responsible for decrypting embedded files for this security handler.

get_file_encryption_key() → bytes

Retrieve the global file encryption key (used for streams and/or strings). If there is no such thing, or the key is not available, an error should be raised.

Raises

PdfKeyNotAvailableError – when the key is not available

classmethod read_cf_dictionary(*cfdict*: *DictionaryObject*, *acts_as_default*: bool) →
Optional[*CryptFilter*]

Interpret a crypt filter dictionary for this type of security handler.

Parameters

- **cfdict** – A crypt filter dictionary.
- **acts_as_default** – Indicates whether this filter is intended to be used in /StrF or /StmF.

Returns

An appropriate *CryptFilter* object, or None if the crypt filter uses the /None method.

Raises

NotImplementedError – Raised when the crypt filter's /CFM entry indicates an unknown crypt filter method.

classmethod process_crypt_filters(*encrypt_dict*: *DictionaryObject*) →
Optional[*CryptFilterConfiguration*]

classmethod `register_crypt_filter`(*method*: [NameObject](#), *factory*: *Callable*[[[DictionaryObject](#), *bool*], [CryptFilter](#)])

get_min_pdf_version() → *Optional*[*Tuple*[*int*, *int*]]

class `pyhanko.pdf_utils.crypt.StandardSecurityHandler`(*version*: [SecurityHandlerVersion](#), *revision*: [StandardSecuritySettingsRevision](#), *legacy_keylen*, *perm_flags*: *int*, *odata*, *udata*, *oeseed*=*None*, *ueseed*=*None*, *encrypted_perms*=*None*, *encrypt_metadata*=*True*, *crypt_filter_config*: *Optional*[[CryptFilterConfiguration](#)] = *None*, *compat_entries*=*True*)

Bases: [SecurityHandler](#)

Implementation of the standard (password-based) security handler.

You shouldn't have to instantiate [StandardSecurityHandler](#) objects yourself. For encrypting new documents, use [build_from_pw\(\)](#) or [build_from_pw_legacy\(\)](#).

For decrypting existing documents, pyHanko will take care of instantiating security handlers through [SecurityHandler.build\(\)](#).

classmethod `get_name`() → *str*

Retrieves the name of this security handler.

Returns

The name of this security handler.

classmethod `build_from_pw_legacy`(*rev*: [StandardSecuritySettingsRevision](#), *id1*, *desired_owner_pass*, *desired_user_pass*=*None*, *keylen_bytes*=16, *use_aes128*=*True*, *perms*: *int* = -4, *crypt_filter_config*=*None*, *encrypt_metadata*=*True*, ***kwargs*)

Initialise a legacy password-based security handler, to attach to a [PdfFileWriter](#). Any remaining keyword arguments will be passed to the constructor.

Danger: The functionality implemented by this handler is deprecated in the PDF standard. We only provide it for testing purposes, and to interface with legacy systems.

Parameters

- **rev** – Security handler revision to use, see [StandardSecuritySettingsRevision](#).
- **id1** – The first part of the document ID.
- **desired_owner_pass** – Desired owner password.
- **desired_user_pass** – Desired user password.
- **keylen_bytes** – Length of the key (in bytes).
- **use_aes128** – Use AES-128 instead of RC4 (default: *True*).
- **perms** – Permission bits to set (defined as an integer)
- **crypt_filter_config** – Custom crypt filter configuration. PyHanko will supply a reasonable default if none is specified.

Returns

A [StandardSecurityHandler](#) instance.

classmethod `build_from_pw(desired_owner_pass, desired_user_pass=None, perms=-4, encrypt_metadata=True, **kwargs)`

Initialise a password-based security handler backed by AES-256, to attach to a [PdfFileWriter](#). This handler will use the new PDF 2.0 encryption scheme.

Any remaining keyword arguments will be passed to the constructor.

Parameters

- **desired_owner_pass** – Desired owner password.
- **desired_user_pass** – Desired user password.
- **perms** – Desired usage permissions.
- **encrypt_metadata** – Whether to set up the security handler for encrypting metadata as well.

Returns

A [StandardSecurityHandler](#) instance.

classmethod `gather_encryption_metadata(encrypt_dict: DictionaryObject) → dict`

Gather and preprocess the “easy” metadata values in an encryption dictionary, and turn them into constructor kwargs.

This function processes `/Length`, `/P`, `/Perms`, `/O`, `/U`, `/OE`, `/UE` and `/EncryptMetadata`.

classmethod `instantiate_from_pdf_object(encrypt_dict: DictionaryObject)`

Instantiate an object of this class using a PDF encryption dictionary as input.

Parameters

encrypt_dict – A PDF encryption dictionary.

Returns

`as_pdf_object()`

Serialise this security handler to a PDF encryption dictionary.

Returns

A PDF encryption dictionary.

authenticate(*credential*, *id1*: *Optional[bytes]* = *None*) → [AuthResult](#)

Authenticate a user to this security handler.

Parameters

- **credential** – The credential to use (a password in this case).
- **id1** – First part of the document ID. This is mandatory for legacy encryption handlers, but meaningless otherwise.

Returns

An [AuthResult](#) object indicating the level of access obtained.

get_file_encryption_key() → bytes

Retrieve the (global) file encryption key for this security handler.

Returns

The file encryption key as a bytes object.

Raises

[misc.PdfReadError](#) – Raised if this security handler was instantiated from an encryption dictionary and no credential is available.

```
class pyhanko.pdf_utils.crypt.PubKeySecurityHandler(version: SecurityHandlerVersion,
                                                    pubkey_handler_subfilter:
                                                    PubKeyAdbeSubFilter, legacy_keylen,
                                                    encrypt_metadata=True, crypt_filter_config:
                                                    Optional[CryptFilterConfiguration] = None,
                                                    recipient_objs: Optional[list] = None,
                                                    compat_entries=True)
```

Bases: [SecurityHandler](#)

Security handler for public key encryption in PDF.

As with the standard security handler, you essentially shouldn't ever have to instantiate these yourself (see [build_from_certs\(\)](#)).

```
classmethod build_from_certs(certs: List[Certificate], keylen_bytes=16,
                             version=SecurityHandlerVersion.AES256, use_aes=True,
                             use_crypt_filters=True, perms: int = -4, encrypt_metadata=True,
                             ignore_key_usage=False, **kwargs) → PubKeySecurityHandler
```

Create a new public key security handler.

This method takes many parameters, but only `certs` is mandatory. The default behaviour is to create a public key encryption handler where the underlying symmetric encryption is provided by AES-256. Any remaining keyword arguments will be passed to the constructor.

Parameters

- **certs** – The recipients' certificates.
- **keylen_bytes** – The key length (in bytes). This is only relevant for legacy security handlers.
- **version** – The security handler version to use.
- **use_aes** – Use AES-128 instead of RC4 (only meaningful if the `version` parameter is [RC4_OR_AES128](#)).
- **use_crypt_filters** – Whether to use crypt filters. This is mandatory for security handlers of version [RC4_OR_AES128](#) or higher.
- **perms** – Permission flags (as a 4-byte signed integer).
- **encrypt_metadata** – Whether to encrypt document metadata.

Warning: See [SecurityHandler](#) for some background on the way pyHanko interprets this value.

- **ignore_key_usage** – If `False`, the *keyEncipherment* key usage extension is required.

Returns

An instance of [PubKeySecurityHandler](#).

```
classmethod get_name() → str
```

Retrieves the name of this security handler.

Returns

The name of this security handler.

```
classmethod support_generic_subfilters() → Set[str]
```

Indicates the generic /SubFilter values that this security handler supports.

Returns

A set of generic protocols (indicated in the `/SubFilter` entry of an encryption dictionary) that this [SecurityHandler](#) class implements. Defaults to the empty set.

classmethod `read_cf_dictionary`(*cfdict*: [DictionaryObject](#), *acts_as_default*: *bool*) → [CryptFilter](#)

Interpret a crypt filter dictionary for this type of security handler.

Parameters

- **cfdict** – A crypt filter dictionary.
- **acts_as_default** – Indicates whether this filter is intended to be used in `/StrF` or `/StmF`.

Returns

An appropriate [CryptFilter](#) object, or `None` if the crypt filter uses the `/None` method.

Raises

NotImplementedError – Raised when the crypt filter's `/CFM` entry indicates an unknown crypt filter method.

classmethod `process_crypt_filters`(*encrypt_dict*: [DictionaryObject](#)) →
Optional[[CryptFilterConfiguration](#)]

classmethod `gather_pub_key_metadata`(*encrypt_dict*: [DictionaryObject](#))

classmethod `instantiate_from_pdf_object`(*encrypt_dict*: [DictionaryObject](#))

Instantiate an object of this class using a PDF encryption dictionary as input.

Parameters

encrypt_dict – A PDF encryption dictionary.

Returns

as_pdf_object()

Serialise this security handler to a PDF encryption dictionary.

Returns

A PDF encryption dictionary.

add_recipients(*certs*: *List*[[Certificate](#)], *perms*=-4, *ignore_key_usage*=False)

authenticate(*credential*: [EnvelopeKeyDecrypter](#), *id1*=None) → [AuthResult](#)

Authenticate a user to this security handler.

Parameters

- **credential** – The credential to use (an instance of [EnvelopeKeyDecrypter](#) in this case).
- **id1** – First part of the document ID. Public key encryption handlers ignore this key.

Returns

An [AuthResult](#) object indicating the level of access obtained.

get_file_encryption_key() → bytes

Retrieve the global file encryption key (used for streams and/or strings). If there is no such thing, or the key is not available, an error should be raised.

Raises

[PdfKeyNotAvailableError](#) – when the key is not available

```
class pyhanko.pdf_utils.crypt.AuthResult(status: AuthStatus, permission_flags: Optional[int] = None)
```

Bases: `object`

Describes the result of an authentication attempt.

status: *AuthStatus*

Authentication status after the authentication attempt.

permission_flags: `Optional[int]` = `None`

Granular permission flags. The precise meaning depends on the security handler.

```
class pyhanko.pdf_utils.crypt.AuthStatus(value)
```

Bases: *OrderedEnum*

Describes the status after an authentication attempt.

FAILED = 0

USER = 1

OWNER = 2

```
class pyhanko.pdf_utils.crypt.SecurityHandlerVersion(value)
```

Bases: `VersionEnum`

Indicates the security handler's version.

The enum constants are named more or less in accordance with the cryptographic algorithms they permit.

RC4_40 = 1

RC4_LONGER_KEYS = 2

RC4_OR_AES128 = 4

AES256 = 5

OTHER = `None`

Placeholder value for custom security handlers.

as_pdf_object() → *PdfObject*

classmethod from_number(value) → *SecurityHandlerVersion*

check_key_length(key_length: *int*) → `int`

```
class pyhanko.pdf_utils.crypt.StandardSecuritySettingsRevision(value)
```

Bases: `VersionEnum`

Indicate the standard security handler revision to emulate.

RC4_BASIC = 2

RC4_EXTENDED = 3

RC4_OR_AES128 = 4

AES256 = 6

OTHER = `None`

Placeholder value for custom security handlers.

`as_pdf_object()` → *PdfObject*

`classmethod from_number(value)` → *StandardSecuritySettingsRevision*

class `pyhanko.pdf_utils.crypt.PubKeyAdbeSubFilter(value)`

Bases: Enum

Enum describing the different subfilters that can be used for public key encryption in the PDF specification.

S3 = `'/adbe.pkcs7.s3'`

S4 = `'/adbe.pkcs7.s4'`

S5 = `'/adbe.pkcs7.s5'`

class `pyhanko.pdf_utils.crypt.CryptFilterConfiguration(crypt_filters: Optional[Dict[str, CryptFilter]] = None, default_stream_filter='/Identity', default_string_filter='/Identity', default_file_filter=None)`

Bases: object

Crypt filter store attached to a security handler.

Instances of this class are not designed to be reusable.

Parameters

- **crypt_filters** – A dictionary mapping names to their corresponding crypt filters.
- **default_stream_filter** – Name of the default crypt filter to use for streams.
- **default_string_filter** – Name of the default crypt filter to use for strings.
- **default_file_filter** – Name of the default crypt filter to use for embedded files.

Note: PyHanko currently is not aware of embedded files, so managing these is the API user's responsibility.

filters()

Enumerate all crypt filters in this configuration.

set_security_handler(handler: SecurityHandler)

Set the security handler on all crypt filters in this configuration.

Parameters

handler – A *SecurityHandler* instance.

get_for_stream()

Retrieve the default crypt filter to use with streams.

Returns

A *CryptFilter* instance.

get_for_string()

Retrieve the default crypt filter to use with strings.

Returns

A *CryptFilter* instance.

get_for_embedded_file()

Retrieve the default crypt filter to use with embedded files.

Returns

A *CryptFilter* instance.

property stream_filter_name: *NameObject*

The name of the default crypt filter to use with streams.

property string_filter_name: *NameObject*

The name of the default crypt filter to use with streams.

property embedded_file_filter_name: *NameObject*

Retrieve the name of the default crypt filter to use with embedded files.

as_pdf_object()

Serialise this crypt filter configuration to a dictionary object, including all its subordinate crypt filters (with the exception of the identity filter, if relevant).

standard_filters()

Return the “standard” filters associated with this crypt filter configuration, i.e. those registered as the defaults for strings, streams and embedded files, respectively.

These sometimes require special treatment (as per the specification).

Returns

A set with one, two or three elements.

class pyhanko.pdf_utils.crypt.CryptFilter

Bases: object

Generic abstract crypt filter class.

The superclass only handles the binding with the security handler, and offers some default implementations for serialisation routines that may be overridden in subclasses.

There is generally no requirement for crypt filters to be compatible with *any* security handler (the leaf classes in this module aren't), but the API supports mixin usage so code can be shared.

property method: *NameObject***Returns**

The method name (/CFM entry) associated with this crypt filter.

property keylen: int**Returns**

The keylength (in bytes) of the key associated with this crypt filter.

encrypt(key, plaintext: bytes, params=None) → bytes

Encrypt plaintext with the specified key.

Parameters

- **key** – The current local key, which may or may not be equal to this crypt filter's global key.
- **plaintext** – Plaintext to encrypt.
- **params** – Optional parameters private to the crypt filter, specified as a PDF dictionary. These can only be used for explicit crypt filters; the parameters are then sourced from the corresponding entry in /DecodeParms.

Returns

The resulting ciphertext.

decrypt(*key*, *ciphertext*: *bytes*, *params*=*None*) → *bytes*

Decrypt ciphertext with the specified key.

Parameters

- **key** – The current local key, which may or may not be equal to this crypt filter’s global key.
- **ciphertext** – Ciphertext to decrypt.
- **params** – Optional parameters private to the crypt filter, specified as a PDF dictionary. These can only be used for explicit crypt filters; the parameters are then sourced from the corresponding entry in `/DecodeParms`.

Returns

The resulting plaintext.

as_pdf_object() → *DictionaryObject*

Serialise this crypt filter to a PDF crypt filter dictionary.

Note: Implementations are encouraged to use a cooperative inheritance model, where subclasses first call `super().as_pdf_object()` and add the keys they need before returning the result.

This makes it easy to write crypt filter mixins that can provide functionality to multiple handlers.

Returns

A PDF crypt filter dictionary.

derive_shared_encryption_key() → *bytes*

Compute the (global) file encryption key for this crypt filter.

Returns

The key, as a *bytes* object.

Raises

misc.PdfError – Raised if the data needed to derive the key is not present (e.g. because the caller hasn’t authenticated yet).

derive_object_key(*idnum*, *generation*) → *bytes*

Derive the encryption key for a specific object, based on the shared file encryption key.

Parameters

- **idnum** – ID of the object being encrypted.
- **generation** – Generation number of the object being encrypted.

Returns

The local key to use for this object.

set_embedded_only()

property shared_key: *bytes*

Return the shared file encryption key for this crypt filter, or attempt to compute it using `derive_shared_encryption_key()` if not available.

class pyhanko.pdf_utils.crypt.**StandardCryptFilter**

Bases: [CryptFilter](#), ABC

Crypt filter for use with the standard security handler.

derive_shared_encryption_key() → bytes

Compute the (global) file encryption key for this crypt filter.

Returns

The key, as a bytes object.

Raises

[misc.PdfError](#) – Raised if the data needed to derive the key is not present (e.g. because the caller hasn't authenticated yet).

as_pdf_object()

Serialise this crypt filter to a PDF crypt filter dictionary.

Note: Implementations are encouraged to use a cooperative inheritance model, where subclasses first call `super().as_pdf_object()` and add the keys they need before returning the result.

This makes it easy to write crypt filter mixins that can provide functionality to multiple handlers.

Returns

A PDF crypt filter dictionary.

class pyhanko.pdf_utils.crypt.**PubKeyCryptFilter**(*, recipients=None, acts_as_default=False, encrypt_metadata=True, **kwargs)

Bases: [CryptFilter](#), ABC

Crypt filter for use with public key security handler. These are a little more independent than their counterparts for the standard security handlers, since different crypt filters can cater to different sets of recipients.

Parameters

- **recipients** – List of CMS objects encoding recipient information for this crypt filters.
- **acts_as_default** – Indicates whether this filter is intended to be used in /StrF or /StmF.
- **encrypt_metadata** – Whether this crypt filter should encrypt document-level metadata.

Warning: See [SecurityHandler](#) for some background on the way pyHanko interprets this value.

add_recipients(certs: List[Certificate], perms=-4, ignore_key_usage=False)

Add recipients to this crypt filter. This always adds one full CMS object to the Recipients array

Parameters

- **certs** – A list of recipient certificates.
- **perms** – The permission bits to assign to the listed recipients.
- **ignore_key_usage** – If False, the *keyEncipherment* key usage extension is required.

authenticate(*credential*) → *AuthResult*

Authenticate to this crypt filter in particular. If used in `/StmF` or `/StrF`, you don't need to worry about calling this method directly.

Parameters

credential – The *EnvelopeKeyDecrypter* to authenticate with.

Returns

An *AuthResult* object indicating the level of access obtained.

derive_shared_encryption_key() → bytes

Compute the (global) file encryption key for this crypt filter.

Returns

The key, as a bytes object.

Raises

misc.PdfError – Raised if the data needed to derive the key is not present (e.g. because the caller hasn't authenticated yet).

as_pdf_object()

Serialise this crypt filter to a PDF crypt filter dictionary.

Note: Implementations are encouraged to use a cooperative inheritance model, where subclasses first call `super().as_pdf_object()` and add the keys they need before returning the result.

This makes it easy to write crypt filter mixins that can provide functionality to multiple handlers.

Returns

A PDF crypt filter dictionary.

class pyhanko.pdf_utils.crypt.IdentityCryptFilter

Bases: *CryptFilter*

Class implementing the trivial crypt filter.

This is a singleton class, so all its instances are identical. Additionally, some of the *CryptFilter* API is nonfunctional. In particular, `as_pdf_object()` always raises an error, since the `/Identity` filter cannot be serialised.

method = `'/None'`

keylen = `0`

derive_shared_encryption_key() → bytes

Always returns an empty byte string.

derive_object_key(*idnum*, *generation*) → bytes

Always returns an empty byte string.

Parameters

- **idnum** – Ignored.
- **generation** – Ignored.

Returns

as_pdf_object()

Not implemented for this crypt filter.

Raises

[misc.PdfError](#) – Always.

encrypt(*key*, *plaintext*: *bytes*, *params*=*None*) → *bytes*

Identity function.

Parameters

- **key** – Ignored.
- **plaintext** – Returned as-is.
- **params** – Ignored.

Returns

The original plaintext.

decrypt(*key*, *ciphertext*: *bytes*, *params*=*None*) → *bytes*

Identity function.

Parameters

- **key** – Ignored.
- **ciphertext** – Returned as-is.
- **params** – Ignored.

Returns

The original ciphertext.

class `pyhanko.pdf_utils.crypt.RC4CryptFilterMixin`(***, *keylen*=5, ***kwargs*)

Bases: *[CryptFilter](#)*, *ABC*

Mixin for RC4-based crypt filters.

Parameters

keylen – Key length, in bytes. Defaults to 5.

method = `'/V2'`

keylen = `None`

encrypt(*key*, *plaintext*: *bytes*, *params*=*None*) → *bytes*

Encrypt data using RC4.

Parameters

- **key** – Local encryption key.
- **plaintext** – Plaintext to encrypt.
- **params** – Ignored.

Returns

Ciphertext.

decrypt(*key*, *ciphertext*: *bytes*, *params*=*None*) → *bytes*

Decrypt data using RC4.

Parameters

- **key** – Local encryption key.

- **ciphertext** – Ciphertext to decrypt.
- **params** – Ignored.

Returns

Plaintext.

derive_object_key(*idnum*, *generation*) → bytes

Derive the local key for the given object ID and generation number, by calling `legacy_derive_object_key()`.

Parameters

- **idnum** – ID of the object being encrypted.
- **generation** – Generation number of the object being encrypted.

Returns

The local key.

class `pyhanko.pdf_utils.crypt.AESCryptFilterMixin`(*, *keylen*, ***kwargs*)

Bases: `CryptFilter`, `ABC`

Mixin for AES-based crypt filters.

keylen = `None`

method = `None`

encrypt(*key*, *plaintext*: bytes, *params*=`None`)

Encrypt data using AES in CBC mode, with PKCS#7 padding.

Parameters

- **key** – The key to use.
- **plaintext** – The plaintext to be encrypted.
- **params** – Ignored.

Returns

The resulting ciphertext, prepended with a 16-byte initialisation vector.

decrypt(*key*, *ciphertext*: bytes, *params*=`None`) → bytes

Decrypt data using AES in CBC mode, with PKCS#7 padding.

Parameters

- **key** – The key to use.
- **ciphertext** – The ciphertext to be decrypted, prepended with a 16-byte initialisation vector.
- **params** – Ignored.

Returns

The resulting plaintext.

derive_object_key(*idnum*, *generation*) → bytes

Derive the local key for the given object ID and generation number.

If the associated handler is of version `SecurityHandlerVersion.AES256` or greater, this method simply returns the global key as-is. If not, the computation is carried out by `legacy_derive_object_key()`.

Parameters

- **idnum** – ID of the object being encrypted.
- **generation** – Generation number of the object being encrypted.

Returns

The local key.

class pyhanko.pdf_utils.crypt.**StandardAESCryptFilter**(*, keylen, **kwargs)

Bases: [StandardCryptFilter](#), [AESCryptFilterMixin](#)

AES crypt filter for the standard security handler.

class pyhanko.pdf_utils.crypt.**StandardRC4CryptFilter**(*, keylen=5, **kwargs)

Bases: [StandardCryptFilter](#), [RC4CryptFilterMixin](#)

RC4 crypt filter for the standard security handler.

class pyhanko.pdf_utils.crypt.**PubKeyAESCryptFilter**(*, recipients=None, acts_as_default=False, encrypt_metadata=True, **kwargs)

Bases: [PubKeyCryptFilter](#), [AESCryptFilterMixin](#)

AES crypt filter for public key security handlers.

class pyhanko.pdf_utils.crypt.**PubKeyRC4CryptFilter**(*, recipients=None, acts_as_default=False, encrypt_metadata=True, **kwargs)

Bases: [PubKeyCryptFilter](#), [RC4CryptFilterMixin](#)

RC4 crypt filter for public key security handlers.

class pyhanko.pdf_utils.crypt.**EnvelopeKeyDecrypter**(cert: Certificate)

Bases: object

General credential class for use with public key security handlers.

This allows the key decryption process to happen offline, e.g. on a smart card.

Parameters

cert – The recipient’s certificate.

decrypt(encrypted_key: bytes, algo_params: KeyEncryptionAlgorithm) → bytes

Invoke the actual key decryption algorithm.

Parameters

- **encrypted_key** – Payload to decrypt.
- **algo_params** – Specification of the encryption algorithm as a CMS object.

Returns

The decrypted payload.

class pyhanko.pdf_utils.crypt.**SimpleEnvelopeKeyDecrypter**(cert: Certificate, private_key: PrivateKeyInfo)

Bases: [EnvelopeKeyDecrypter](#), [SerialisableCredential](#)

Implementation of [EnvelopeKeyDecrypter](#) where the private key is an RSA key residing in memory.

Parameters

- **cert** – The recipient’s certificate.
- **private_key** – The recipient’s private key.

classmethod `get_name()` → str

Get the type name of the credential, which will be embedded into serialised values and used on deserialisation.

static `load(key_file, cert_file, key_passphrase=None)`

Load a key decrypter using key material from files on disk.

Parameters

- **key_file** – File containing the recipient’s private key.
- **cert_file** – File containing the recipient’s certificate.
- **key_passphrase** – Passphrase for the key file, if applicable.

Returns

An instance of [*SimpleEnvelopeKeyDecrypter*](#).

classmethod `load_pkcs12(pfx_file, passphrase=None)`

Load a key decrypter using key material from a PKCS#12 file on disk.

Parameters

- **pfx_file** – Path to the PKCS#12 file containing the key material.
- **passphrase** – Passphrase for the private key, if applicable.

Returns

An instance of [*SimpleEnvelopeKeyDecrypter*](#).

decrypt(*encrypted_key*: bytes, *algo_params*: *KeyEncryptionAlgorithm*) → bytes

Decrypt the payload using RSA with PKCS#1 v1.5 padding. Other schemes are not (currently) supported by this implementation.

Parameters

- **encrypted_key** – Payload to decrypt.
- **algo_params** – Specification of the encryption algorithm as a CMS object. Must use `rsaes_pkcs1v15`.

Returns

The decrypted payload.

class `pyhanko.pdf_utils.crypt.SerialisedCredential(credential_type: str, data: bytes)`

Bases: object

A credential in serialised form.

credential_type: str

The registered type name of the credential (see [*SerialisableCredential.register\(\)*](#)).

data: bytes

The credential data, as a byte string.

class `pyhanko.pdf_utils.crypt.SerialisableCredential`

Bases: ABC

Class representing a credential that can be serialised.

classmethod `get_name()` → str

Get the type name of the credential, which will be embedded into serialised values and used on deserialisation.

static register(cls: Type[SerialisableCredential])

Register a subclass into the credential serialisation registry, using the name returned by `get_name()`. Can be used as a class decorator.

Parameters

cls – The subclass.

Returns

The subclass.

static deserialize(ser_value: SerialisedCredential) → SerialisableCredential

Deserialize a `SerialisedCredential` value by looking up the proper subclass of `SerialisableCredential` and invoking its deserialisation method.

Parameters

ser_value – The value to deserialize.

Returns

The deserialised credential.

Raises

`misc.PdfReadError` – If a deserialisation error occurs.

serialize() → SerialisedCredential

Serialize a value to an annotated `SerialisedCredential` value.

Returns

A `SerialisedCredential` value.

Raises

`misc.PdfWriteError` – If a serialisation error occurs.

`pyhanko.pdf_utils.crypt.build_crypt_filter`(reg: Dict[NameObject, Callable[[DictionaryObject, bool], CryptFilter]], cfdict: DictionaryObject, acts_as_default: bool) → Optional[CryptFilter]

Interpret a crypt filter dictionary for a security handler.

Parameters

- **reg** – A registry of named crypt filters.
- **cfdict** – A crypt filter dictionary.
- **acts_as_default** – Indicates whether this filter is intended to be used in /StrF or /StmF.

Returns

An appropriate `CryptFilter` object, or None if the crypt filter uses the /None method.

Raises

`NotImplementedError` – Raised when the crypt filter's /CFM entry indicates an unknown crypt filter method.

exception `pyhanko.pdf_utils.crypt.PdfKeyNotAvailableError`(msg: str, *args)

Bases: `PdfReadError`

pyhanko.pdf_utils.embed module

Utility classes for handling embedded files in PDFs.

New in version 0.7.0.

`pyhanko.pdf_utils.embed.embed_file(pdf_writer: BasePdfFileWriter, spec: FileSpec)`

Embed a file in the document-wide embedded file registry of a PDF writer.

Parameters

- **pdf_writer** – PDF writer to house the embedded file.
- **spec** – File spec describing the embedded file.

Returns

`class pyhanko.pdf_utils.embed.EmbeddedFileObject(pdf_writer: BasePdfFileWriter, dict_data=None, stream_data=None, encoded_data=None, params: Optional[EmbeddedFileParams] = None, mime_type: Optional[str] = None)`

Bases: *StreamObject*

`classmethod from_file_data(pdf_writer: BasePdfFileWriter, data: bytes, compress=True, params: Optional[EmbeddedFileParams] = None, mime_type: Optional[str] = None) → EmbeddedFileObject`

Construct an embedded file object from file data.

This is a very thin wrapper around the constructor, with a slightly less intimidating API.

Note: This method will not register the embedded file into the document’s embedded file namespace, see *embed_file()*.

Parameters

- **pdf_writer** – PDF writer to use.
- **data** – File contents, as a `bytes` object.
- **compress** – Whether to compress the embedded file’s contents.
- **params** – Optional embedded file parameters.
- **mime_type** – Optional MIME type string.

Returns

An embedded file object.

`write_to_stream(stream, handler=None, container_ref=None)`

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

```
class pyhanko.pdf_utils.embed.EmbeddedFileParams(embed_size: bool = True, embed_checksum: bool =
                                                True, creation_date: Union[datetime.datetime,
                                                NoneType] = None, modification_date:
                                                Union[datetime.datetime, NoneType] = None)
```

Bases: object

embed_size: bool = True

If true, record the file size of the embedded file.

Note: This value is computed over the file content before PDF filters are applied. This may have performance implications in cases where the file stream contents are presented in pre-encoded form.

embed_checksum: bool = True

If true, add an MD5 checksum of the file contents.

Note: This value is computed over the file content before PDF filters are applied. This may have performance implications in cases where the file stream contents are presented in pre-encoded form.

creation_date: Optional[datetime] = None

Record the creation date of the embedded file.

modification_date: Optional[datetime] = None

Record the modification date of the embedded file.

```
class pyhanko.pdf_utils.embed.FileSpec(file_spec_string: str, file_name: Optional[str] = None,
                                       embedded_data: Optional[EmbeddedFileObject] = None,
                                       description: Optional[str] = None, af_relationship:
                                       Optional[NameObject] = None, f_related_files:
                                       Optional[List[RelatedFileSpec]] = None, uf_related_files:
                                       Optional[List[RelatedFileSpec]] = None)
```

Bases: object

Dataclass modelling an embedded file description in a PDF.

file_spec_string: str

A path-like file specification string, or URL.

Note: For backwards compatibility, this string should be encodable in PDFDocEncoding. For names that require general Unicode support, refer to [file_name](#).

file_name: Optional[str] = None

A path-like Unicode file name.

embedded_data: Optional[EmbeddedFileObject] = None

Reference to a stream object containing the file's data, as embedded in the PDF file.

description: Optional[str] = None

Textual description of the file.

af_relationship: Optional[NameObject] = None

Associated file relationship specifier.

f_related_files: `List[RelatedFileSpec] = None`

Related files with PDFDocEncoded names.

uf_related_files: `List[RelatedFileSpec] = None`

Related files with Unicode-encoded names.

as_pdf_object() \rightarrow *DictionaryObject*

Represent the file spec as a PDF dictionary.

class pyhanko.pdf_utils.embed.RelatedFileSpec(name: str, embedded_data: EmbeddedFileObject)

Bases: object

Dataclass modelling a RelatedFile construct in PDF.

name: str

Name of the related file.

Note: The encoding requirements of this field depend on whether the related file is included via the /F or /UF key.

embedded_data: *EmbeddedFileObject*

Reference to a stream object containing the file's data, as embedded in the PDF file.

classmethod fmt_related_files(lst: List[RelatedFileSpec])

pyhanko.pdf_utils.embed.wrap_encrypted_payload(plaintext_payload: bytes, *, password: Optional[str] = None, certs: Optional[List[Certificate]] = None, security_handler: Optional[SecurityHandler] = None, file_spec_string: str = 'attachment.pdf', params: Optional[EmbeddedFileParams] = None, file_name: Optional[str] = None, description='Wrapped document', include_explanation_page=True) \rightarrow PdfFileWriter

Include a PDF document as an encrypted attachment in a wrapper document.

This function sets certain flags in the wrapper document's collection dictionary to instruct compliant PDF viewers to display the attachment instead of the wrapping document. Viewers that do not fully support PDF collections will display a landing page instead, explaining how to open the attachment manually.

Using this method mitigates some weaknesses in the PDF standard's encryption provisions, and makes it harder to manipulate the encrypted attachment without knowing the encryption key.

Danger: Until PDF supports authenticated encryption mechanisms, this is a mitigation strategy, not a fool-proof defence mechanism.

Warning: While users of viewers that do not support PDF collections can still open the attached file manually, the viewer still has to support PDF files where only the attachments are encrypted.

Note: This is not quite the same as the “unencrypted wrapper document” pattern discussed in the PDF 2.0 specification. The latter is intended to support nonstandard security handlers. This function uses a standard security

handler on the wrapping document to encrypt the attachment as a binary blob. Moreover, the functionality in this function is available in PDF 1.7 viewers as well.

Parameters

- **plaintext_payload** – The plaintext payload (a binary representation of a PDF document).
- **security_handler** – The security handler to use on the wrapper document. If `None`, a security handler will be constructed based on the `password` or `certs` parameter.
- **password** – Password to encrypt the attachment with. Will be ignored if `security_handler` is provided.
- **certs** – Encrypt the file using PDF public-key encryption, targeting the keys in the provided certificates. Will be ignored if `security_handler` is provided.
- **file_spec_string** – PDFDocEncoded file spec string for the attachment.
- **params** – Embedded file parameters to use.
- **file_name** – Unicode file name for the attachment.
- **description** – Description for the attachment
- **include_explanation_page** – If `False`, do not generate an explanation page in the wrapper document. This setting could be useful if you want to customise the wrapper document's behaviour yourself.

Returns

A `PdfFileWriter` representing the wrapper document.

pyhanko.pdf_utils.extensions module

class `pyhanko.pdf_utils.extensions.DevExtensionMultivalued(value)`

Bases: `Enum`

Setting indicating how an extension is expected to behave well w.r.t. the new mechanism for multivalued extensions in ISO 32000-2:2020.

ALWAYS = 1

Always serialise this extension as a multivalued extension.

NEVER = 2

Never serialise this extension as a multivalued extension.

MAYBE = 3

Make this extension single-valued whenever possible, but allow multiple values as well, e.g. when a different but non-comparable extension with the same prefix is already present in the file.

class `pyhanko.pdf_utils.extensions.DeveloperExtension(prefix_name: NameObject, base_version: NameObject, extension_level: int, url: Optional[str] = None, extension_revision: Optional[str] = None, compare_by_level: bool = False, subsumed_by: Iterable[int] = (), subsumes: Iterable[int] = (), multivalued: DevExtensionMultivalued = DevExtensionMultivalued.MAYBE)`

Bases: `object`

PDF developer extension designation.

prefix_name: *NameObject*

Registered developer prefix.

base_version: *NameObject*

Base version on to which the extension applies.

extension_level: `int`

Extension level.

url: `Optional[str] = None`

Optional URL linking to the extension's documentation.

extension_revision: `Optional[str] = None`

Optional extra revision information. Not comparable.

compare_by_level: `bool = False`

Compare developer extensions by level number. If this value is `True` and a copy of this extension already exists in the target file with a higher level number, do not override it. If one exists with a lower level number, override it.

If this value is `False`, the decision is based on *subsumed_by* and *subsumes*.

Warning: It is generally not safe to assume that extension levels are used as a versioning system (i.e. that higher extension levels supersede lower ones), hence why the default is `False`.

subsumed_by: `Iterable[int] = ()`

List of extension levels that would subsume this one. If one of these is present in the extensions dictionary, attempting to register this extension will not override it.

Default value: empty.

Warning: This parameter is ignored if *compare_by_level* is `True`.

subsumes: `Iterable[int] = ()`

List of extensions explicitly subsumed by this one. If one of these is present in the extensions dictionary, attempting to register this extension will override it.

Default value: empty.

Warning: This parameter is ignored if *compare_by_level* is `True`.

multivalued: *DevExtensionMultivalued* = 3

Setting indicating whether this extension is expected to behave well w.r.t. the new mechanism for multivalued extensions in ISO 32000-2:2020.

as_pdf_object() → *DictionaryObject*

Format the data in this object into a PDF dictionary for registration into the */Extensions* dictionary.

Returns

A *generic.DictionaryObject*.

pyhanko.pdf_utils.filters module

Implementation of stream filters for PDF.

Taken from PyPDF2 with modifications. See [here](#) for the original license of the PyPDF2 project.

Note that not all decoders specified in the standard are supported. In particular /LZWDecode and the various JPEG-based decoders are missing.

class pyhanko.pdf_utils.filters.Decoder

Bases: object

General filter/decoder interface.

decode(data: bytes, decode_params: dict) → bytes

Decode a stream.

Parameters

- **data** – Data to decode.
- **decode_params** – Decoder parameters, sourced from the /DecoderParams entry associated with this filter.

Returns

Decoded data.

encode(data: bytes, decode_params: dict) → bytes

Encode a stream.

Parameters

- **data** – Data to encode.
- **decode_params** – Encoder parameters, sourced from the /DecoderParams entry associated with this filter.

Returns

Encoded data.

class pyhanko.pdf_utils.filters.ASCII85Decode

Bases: *Decoder*

Implementation of the base 85 encoding scheme specified in ISO 32000-1.

encode(data: bytes, decode_params=None) → bytes

Encode a stream.

Parameters

- **data** – Data to encode.
- **decode_params** – Encoder parameters, sourced from the /DecoderParams entry associated with this filter.

Returns

Encoded data.

decode(*data*, *decode_params*=None)

Decode a stream.

Parameters

- **data** – Data to decode.
- **decode_params** – Decoder parameters, sourced from the /DecoderParams entry associated with this filter.

Returns

Decoded data.

class pyhanko.pdf_utils.filters.ASCIISHexDecode

Bases: [Decoder](#)

Wrapper around `binascii.hexlify()` that implements the [Decoder](#) interface.

encode(*data*: bytes, *decode_params*=None) → bytes

Encode a stream.

Parameters

- **data** – Data to encode.
- **decode_params** – Encoder parameters, sourced from the /DecoderParams entry associated with this filter.

Returns

Encoded data.

decode(*data*, *decode_params*=None)

Decode a stream.

Parameters

- **data** – Data to decode.
- **decode_params** – Decoder parameters, sourced from the /DecoderParams entry associated with this filter.

Returns

Decoded data.

class pyhanko.pdf_utils.filters.FlateDecode

Bases: [Decoder](#)

Implementation of the /FlateDecode filter.

Warning: Currently not all predictor values are supported. This may cause problems when extracting image data from PDF files.

decode(*data*: bytes, *decode_params*)

Decode a stream.

Parameters

- **data** – Data to decode.
- **decode_params** – Decoder parameters, sourced from the /DecoderParams entry associated with this filter.

Returns

Decoded data.

encode(*data*, *decode_params*=None)

Encode a stream.

Parameters

- **data** – Data to encode.
- **decode_params** – Encoder parameters, sourced from the /DecoderParams entry associated with this filter.

Returns

Encoded data.

`pyhanko.pdf_utils.filters.get_generic_decoder(name: str) → Decoder`

Instantiate a specific stream filter decoder type by (PDF) name.

The following names are recognised:

- **/FlateDecode** or **/F1** for the decoder implementing Flate compression.
- **/ASCIIHexDecode** or **/AHx** for the decoder that converts bytes to their hexadecimal representations.
- **/ASCII85Decode** or **/A85** for the decoder that converts byte strings to a base-85 textual representation.

Warning: /Crypt is a special case because it requires access to the document's security handler.

Warning: LZW compression is currently unsupported, as are most compression methods that are used specifically for image data.

Parameters**name** – Name of the decoder to instantiate.**pyhanko.pdf_utils.font package****pyhanko.pdf_utils.font.api module****class** `pyhanko.pdf_utils.font.api.ShapeResult`(*graphics_ops*: bytes, *x_advance*: float, *y_advance*: float)

Bases: object

Result of shaping a Unicode string.

graphics_ops: bytes

PDF graphics operators to render the glyphs.

x_advance: float

Total horizontal advance in em units.

y_advance: float

Total vertical advance in em units.

```
class pyhanko.pdf_utils.font.api.FontEngine(writer: BasePdfFileWriter, base_postscript_name: str,
                                             embedded_subset: bool, obj_stream=None)
```

Bases: object

General interface for text shaping and font metrics.

property uses_complex_positioning

If True, this font engine expects the line matrix to always be equal to the text matrix when exiting and entering [shape\(\)](#). In other words, the current text position is where `0 0 Td` would move to.

If False, this method does not use any text positioning operators, and therefore uses the PDF standard's 'natural' positioning rules for text showing operators.

The default is True unless overridden.

shape(txt: str) → ShapeResult

Render a string to a format suitable for inclusion in a content stream and measure its total cursor advancement vector in em units.

Parameters

txt – String to shape.

Returns

A shaping result.

as_resource() → PdfObject

Convert a [FontEngine](#) to a PDF object suitable for embedding inside a resource dictionary.

Note: If the PDF object is an indirect reference, the caller must not attempt to dereference it. In other words, implementations can use preallocated references to delay subsetting until the last possible moment (this is even encouraged, see [prepare_write\(\)](#)).

Returns

A PDF dictionary.

prepare_write()

Called by the writer that manages this font resource before the PDF content is written to a stream.

Subsetting operations and the like should be carried out as part of this method.

```
class pyhanko.pdf_utils.font.api.FontSubsetCollection(base_postscript_name: str, subsets:
                                                       Dict[Union[str, NoneType],
                                                           ForwardRef('FontEngine')] = <factory>)
```

Bases: object

base_postscript_name: str

Base postscript name of the font.

subsets: Dict[Optional[str], FontEngine]

Dictionary mapping prefixes to subsets. None represents the full font.

add_subset() → str

```
class pyhanko.pdf_utils.font.api.FontEngineFactory
```

Bases: object

create_font_engine(writer: BasePdfFileWriter, obj_stream=None) → FontEngine

pyhanko.pdf_utils.font.basic module

```
class pyhanko.pdf_utils.font.basic.SimpleFontEngineFactory(name: str, avg_width: float, meta:
Optional[SimpleFontMeta] = None)
```

Bases: *FontEngineFactory*

```
create_font_engine(writer: BasePdfFileWriter, obj_stream=None)
```

```
static default_factory()
```

Returns

A *FontEngineFactory* instance representing the Courier standard font.

```
class pyhanko.pdf_utils.font.basic.SimpleFontEngine(writer: BasePdfFileWriter, name: str,
avg_width: float, meta:
Optional[SimpleFontMeta] = None)
```

Bases: *FontEngine*

Simplistic font engine that effectively only works with PDF standard fonts, and does not care about font metrics. Best used with monospaced fonts such as Courier.

property uses_complex_positioning

If True, this font engine expects the line matrix to always be equal to the text matrix when exiting and entering *shape()*. In other words, the current text position is where `0 0 Td` would move to.

If False, this method does not use any text positioning operators, and therefore uses the PDF standard's 'natural' positioning rules for text showing operators.

The default is True unless overridden.

shape(txt) → ShapeResult

Render a string to a format suitable for inclusion in a content stream and measure its total cursor advancement vector in em units.

Parameters

txt – String to shape.

Returns

A shaping result.

as_resource()

Convert a *FontEngine* to a PDF object suitable for embedding inside a resource dictionary.

Note: If the PDF object is an indirect reference, the caller must not attempt to dereference it. In other words, implementations can use preallocated references to delay subsetting until the last possible moment (this is even encouraged, see *prepare_write()*).

Returns

A PDF dictionary.

```
class pyhanko.pdf_utils.font.basic.SimpleFontMeta(first_char: int, last_char: int, widths: List[int],
descriptor:
pyhanko.pdf_utils.generic.DictionaryObject)
```

Bases: object

```
first_char: int
```

last_char: `int`

widths: `List[int]`

descriptor: *DictionaryObject*

`pyhanko.pdf_utils.font.basic.get_courier(pdf_writer: BasePdfFileWriter)`

Quick-and-dirty way to obtain a Courier font resource.

Parameters

pdf_writer – A PDF writer.

Returns

A resource dictionary representing the standard Courier font (or one of its metric equivalents).

pyhanko.pdf_utils.font.opentype module

Basic support for OpenType/TrueType font handling & subsetting.

This module relies on *fontTools* for OTF parsing and subsetting, and on HarfBuzz (via *uharfbuzz*) for shaping.

```
class pyhanko.pdf_utils.font.opentype.GlyphAccumulator(writer: BasePdfFileWriter, font_handle,  
                                                    font_size, features=None,  
                                                    ot_language_tag=None,  
                                                    ot_script_tag=None,  
                                                    writing_direction=None,  
                                                    bcp47_lang_code=None, obj_stream=None)
```

Bases: *FontEngine*

Utility to collect & measure glyphs from OpenType/TrueType fonts.

Parameters

- **writer** – A PDF writer.
- **font_handle** – File-like object
- **font_size** – Font size in pt units.

Note: This is only relevant for some positioning intricacies (or hacks, depending on your perspective) that may not matter for your use case.

- **features** – Features to use. If *None*, use HarfBuzz defaults.
- **ot_script_tag** – OpenType script tag to use. Will be guessed by HarfBuzz if not specified.
- **ot_language_tag** – OpenType language tag to use. Defaults to the default language system for the current script.
- **writing_direction** – Writing direction, one of ‘ltr’, ‘rtl’, ‘ttb’ or ‘btt’. Will be guessed by HarfBuzz if not specified.
- **bcp47_lang_code** – BCP 47 language code. Used to mark the text’s language in the PDF content stream, if specified.
- **obj_stream** – Try to put font-related objects into a particular object stream, if specified.

`marked_content_property_list(txt) → DictionaryObject`

shape(*txt: str, with_actual_text: bool = True*) → *ShapeResult*

Render a string to a format suitable for inclusion in a content stream and measure its total cursor advancement vector in em units.

Parameters

txt – String to shape.

Returns

A shaping result.

prepare_write()

This implementation of `prepare_write` will embed a subset of this glyph accumulator's font into the PDF writer it belongs to. Said subset will include all glyphs necessary to render the strings provided to the accumulator via `feed_string()`.

Danger: Due to the way `fontTools` handles subsetting, this is a destructive operation. The in-memory representation of the original font will be overwritten by the generated subset.

as_resource() → *IndirectObject*

Convert a *FontEngine* to a PDF object suitable for embedding inside a resource dictionary.

Note: If the PDF object is an indirect reference, the caller must not attempt to dereference it. In other words, implementations can use preallocated references to delay subsetting until the last possible moment (this is even encouraged, see `prepare_write()`).

Returns

A PDF dictionary.

```
class pyhanko.pdf_utils.font.opentype.GlyphAccumulatorFactory(font_file: str, font_size: int = 10,
                                                             ot_script_tag: Optional[str] =
                                                             None, ot_language_tag:
                                                             Optional[str] = None,
                                                             writing_direction: Optional[str] =
                                                             None, create_objstream_if_needed:
                                                             bool = True)
```

Bases: *FontEngineFactory*

Stateless callable helper class to instantiate *GlyphAccumulator* objects.

font_file: str

Path to the OTF/TTF font to load.

font_size: int = 10

Font size.

ot_script_tag: str = None

OpenType script tag to use. Will be guessed by HarfBuzz if not specified.

ot_language_tag: str = None

OpenType language tag to use. Defaults to the default language system for the current script.

writing_direction: str = None

Writing direction, one of 'ltr', 'rtl', 'ttb' or 'btt'. Will be guessed by HarfBuzz if not specified.

create_objstream_if_needed: `bool = True`

Create an object stream to hold this glyph accumulator's assets if no object stream is passed in, and the writer supports object streams.

create_font_engine(*writer*: [BasePdfFileWriter](#), *obj_stream*=None) → *GlyphAccumulator*

pyhanko.pdf_utils.generic module

Implementation of PDF object types and other generic functionality. The internals were imported from PyPDF2, with modifications.

See [here](#) for the original license of the PyPDF2 project.

class `pyhanko.pdf_utils.generic.Dereferenceable`

Bases: `object`

Represents an opaque reference to a PDF object associated with a PDF Handler (see [PdfHandler](#)).

This can either be a reference to an object with an object ID (see [Reference](#)) or a reference to the trailer of a PDF document (see [TrailerReference](#)).

get_object() → *PdfObject*

Retrieve the PDF object backing this dereferenceable.

Returns

A *PdfObject*.

get_pdf_handler()

Return the PDF handler associated with this dereferenceable.

Returns

a *PdfHandler*.

class `pyhanko.pdf_utils.generic.Reference`(*idnum*: *int*, *generation*: *int* = 0, *pdf*: *Optional[object]* = None)

Bases: [Dereferenceable](#)

A reference to an object with a certain ID and generation number, with a PDF handler attached to it.

Warning: Contrary to what one might expect, the generation number does *not* indicate the document revision in which the object was modified. In fact, nonzero generation numbers are exceedingly rare these days; in most real-world PDF files, objects are simply overridden without ever increasing the generation number.

Except in very specific circumstances, dereferencing a [Reference](#) will return the most recent version of the object with the stated object ID and generation number.

idnum: `int`

The object's ID.

generation: `int = 0`

The object's generation number (usually 0)

pdf: `object = None`

The PDF handler associated with this reference, an instance of [PdfHandler](#).

Warning: This field is ignored when hashing or comparing [Reference](#) objects, so it is the API user’s responsibility to not mix up references originating from unrelated PDF handlers.

get_object() → [PdfObject](#)

Retrieve the PDF object backing this dereferenceable.

Returns

A [PdfObject](#).

get_pdf_handler()

Return the PDF handler associated with this dereferenceable.

Returns

a [PdfHandler](#).

class pyhanko.pdf_utils.generic.TrailerReference(*reader*)

Bases: [Dereferenceable](#)

A reference to the trailer of a PDF document.

Warning: Since the trailer does not have a well-defined object ID in files with “classical” cross-reference tables (as opposed to cross-reference streams), this is not a subclass of [Reference](#).

Parameters

reader – a [PdfFileReader](#)

get_object() → [PdfObject](#)

Retrieve the PDF object backing this dereferenceable.

Returns

A [PdfObject](#).

get_pdf_handler()

Return the PDF handler associated with this dereferenceable.

Returns

a [PdfHandler](#).

class pyhanko.pdf_utils.generic.PdfObject

Bases: object

Superclass for all PDF objects.

container_ref: [Dereferenceable](#) = None

For objects read from a file, *container_ref* points to the unique addressable object containing this object.

Note: Consider the following object definition in a PDF file:

```
4 0 obj
<< /Foo (Bar) >>
```

This declares a dictionary with ID 4, but the values /Foo and (Bar) are also PDF objects (a name and a string, respectively). All of these will have *container_ref* given by a [Reference](#) with object ID 4 and generation number 0.

If an object is part of the trailer of a PDF file, *container_ref* will be a [TrailerReference](#). For newly created objects (i.e. those not read from a file), *container_ref* is always *None*.

get_container_ref() → [Dereferenceable](#)

Return a reference to the closest parent object containing this object. Raises an error if no such reference can be found.

get_object()

Resolves indirect references.

Returns

self, unless an instance of [IndirectObject](#).

write_to_stream(*stream*, *handler=None*, *container_ref: Optional[Reference] = None*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

class pyhanko.pdf_utils.generic.**IndirectObject**(*idnum*, *generation*, *pdf*)

Bases: [PdfObject](#), [Dereferenceable](#)

Thin wrapper around a [Reference](#), implementing both the [Dereferenceable](#) and [PdfObject](#) interfaces.

Warning: For many purposes, this class is functionally interchangeable with [Reference](#), with one important exception: [IndirectObject](#) instances pointing to the same reference but occurring at different locations in the file may have distinct *container_ref* values.

get_object()

Returns

The PDF object this reference points to.

get_pdf_handler()

Return the PDF handler associated with this dereferenceable.

Returns

a [PdfHandler](#).

property idnum: int

Returns

the object ID of this reference.

property generation

Returns

the generation number of this reference.

write_to_stream(*stream*, *handler=None*, *container_ref=None*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.

- **container_ref** – Local encryption key.
- **handler** – Security handler

static read_from_stream(*stream*, *container_ref*: [Dereferenceable](#))

class pyhanko.pdf_utils.generic.**NullObject**

Bases: [PdfObject](#)

PDF *null* object.

All instances are treated as equal and falsy.

write_to_stream(*stream*, *handler*=None, *container_ref*=None)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

static read_from_stream(*stream*)

class pyhanko.pdf_utils.generic.**BooleanObject**(*value*)

Bases: [PdfObject](#)

PDF boolean value.

write_to_stream(*stream*, *handler*=None, *container_ref*=None)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

static read_from_stream(*stream*)

class pyhanko.pdf_utils.generic.**FloatObject**(*value*='0', *context*=None)

Bases: [Decimal](#), [PdfObject](#)

PDF Float object.

Internally, these are treated as decimals (and therefore actually fixed-point objects, to be precise).

as_numeric()

Returns

a Python float value for this object.

write_to_stream(*stream*, *handler*=None, *container_ref*=None)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

```
class pyhanko.pdf_utils.generic.NumberObject(value)
```

Bases: int, [PdfObject](#)

PDF number object. This is the PDF type for integer values.

```
NumberPattern = re.compile(b'^+-.0-9')
```

```
ByteDot = b'.'
```

```
as_numeric()
```

Returns

a Python int value for this object.

```
write_to_stream(stream, handler=None, container_ref=None)
```

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

```
static read_from_stream(stream)
```

```
class pyhanko.pdf_utils.generic.ByteStringObject
```

Bases: bytes, [PdfObject](#)

PDF bytestring class.

```
property original_bytes
```

For compatibility with [TextStringObject.original_bytes](#)

```
write_to_stream(stream, handler=None, container_ref=None)
```

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

```
class pyhanko.pdf_utils.generic.TextStringObject
```

Bases: str, [PdfObject](#)

PDF text string object.

```
autodetected_encoding: Optional[TextStringEncoding] = None
```

Autodetected encoding when parsing the file.

```
force_output_encoding: Optional[TextStringEncoding] = None
```

Output encoding to use when serialising the string. The default is to try PDFDocEncoding first, and fall back to UTF-16BE.

```
property original_bytes
```

Retrieve the original bytes of the string as specified in the source file.

This may be necessary if this string was misidentified as a text string.

write_to_stream(*stream*, *handler=None*, *container_ref=None*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

class pyhanko.pdf_utils.generic.**NameObject**

Bases: `str`, [PdfObject](#)

PDF name object. These are valid Python strings, but names and strings are treated differently in the PDF specification, so proper care is required.

DELIMITER_PATTERN = `re.compile(b'\\s|\\[\\(\\)\\<\\[\\]\\{\\}/%]|\\x00')`

write_to_stream(*stream*, *handler=None*, *container_ref=None*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

static read_from_stream(*stream*)

class pyhanko.pdf_utils.generic.**ArrayObject**(*iterable=()*, */*)

Bases: `list`, [PdfObject](#)

PDF array object. This class extends from Python's list class, and supports its interface.

Warning: Contrary to the case of dictionary objects, PyPDF2 does not transparently dereference array entries when accessed using `__getitem__()`. For usability & consistency reasons, I decided to depart from that and dereference automatically. This makes the behaviour of [ArrayObject](#) consistent with [DictionaryObject](#).

That said, some vestiges of the old PyPDF2 behaviour may linger in the codebase. I'll fix those as I get to them.

raw_get(*index*, *decrypt*: [EncryptedObjAccess](#) = [EncryptedObjAccess.TRANSPARENT](#))

Changed in version 0.14.0: `decrypt` parameter is no longer boolean

Get a value from an array without dereferencing. In other words, if the value corresponding to the given key is of type [IndirectObject](#), the indirect reference will not be resolved.

Parameters

- **index** – Key to look up in the dictionary.
- **decrypt** – What to do when retrieving encrypted objects; see [EncryptedObjAccess](#). The default is [EncryptedObjAccess.TRANSPARENT](#).

Returns

A [PdfObject](#).

write_to_stream(*stream*, *handler=None*, *container_ref=None*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

static read_from_stream(*stream*, *container_ref*)

class pyhanko.pdf_utils.generic.**DictionaryObject**(*dict_data=None*)

Bases: dict, [PdfObject](#)

A PDF dictionary object.

Keys in a PDF dictionary are PDF names, and values are PDF objects.

When accessing a key using the standard `__getitem__()` syntax, [IndirectObject](#) references will be resolved.

raw_get(*key*, *decrypt*: [EncryptedObjAccess](#) = [EncryptedObjAccess.TRANSPARENT](#))

Changed in version 0.14.0: `decrypt` parameter is no longer boolean

Get a value from a dictionary without dereferencing. In other words, if the value corresponding to the given key is of type [IndirectObject](#), the indirect reference will not be resolved.

Parameters

- **key** – Key to look up in the dictionary.
- **decrypt** – What to do when retrieving encrypted objects; see [EncryptedObjAccess](#). The default is [EncryptedObjAccess.TRANSPARENT](#).

Returns

A [PdfObject](#).

setdefault(*key*, *value=None*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

get_and_apply(*key*, *function*: Callable[[[PdfObject](#)], Any], *, *raw=False*, *default=None*)

get_value_as_reference(*key*, *optional=False*) → [Reference](#)

write_to_stream(*stream*, *handler=None*, *container_ref=None*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

static read_from_stream(*stream*, *container_ref*: [Dereferenceable](#), *as_metadata_stream*: bool = False)

```
class pyhanko.pdf_utils.generic.StreamObject(dict_data=None, stream_data=None,
                                              encoded_data=None, handler=None)
```

Bases: *DictionaryObject*

PDF stream object.

Essentially, a PDF stream is a dictionary object with a binary blob of data attached. This data can be encoded by various filters (not all of which are currently supported, see *filters*).

A stream object can be initialised with encoded or decoded data. The former is used by *reader.PdfFileReader* to provide on-demand decoding, with *writer.BasePdfFileWriter* and its subclasses working the other way around.

Note: The *StreamObject* class manages some of its dictionary keys by itself. This is partly the case for the various */Filter* and */DecodeParms* entries, but also for the */Length* entry. The latter will be overwritten as necessary.

Parameters

- **dict_data** – The dictionary data for this stream object.
- **stream_data** – The (unencoded) stream data.
- **encoded_data** – The encoded stream data.

Warning: Ordinarily, a stream can be initialised either from decoded and from encoded data.

If both *stream_data* and *encoded_data* are provided, the caller is responsible for making sure that both are compatible given the currently relevant filter configuration.

- **handler** – A reference to the currently active *pyhanko.pdf_utils.crypt.SecurityHandler*. This is only necessary if the stream requires crypt filters.

```
add_crypt_filter(name='/Identity', params=None, handler=None)
```

```
strip_filters()
```

Ensure the stream is decoded, and remove any filters.

property data: bytes

Return the decoded stream data as bytes. If the stream hasn't been decoded yet, it will be decoded on-the-fly.

Raises

.misc.PdfStreamError – If the stream could not be decoded.

property encoded_data: bytes

Return the encoded stream data as bytes. If the stream hasn't been encoded yet, it will be encoded on-the-fly.

Raises

.misc.PdfStreamError – If the stream could not be encoded.

```
apply_filter(filter_name, params=None, allow_duplicates: Optional[bool] = True)
```

Apply a new filter to this stream. This filter will be prepended to any existing filters. This means that is is placed *last* in the encoding order, but *first* in the decoding order.

Note: Calling this method on an encoded stream will first cause the stream to be decoded using the filters already present. The cached value for the encoded stream data will be cleared.

Parameters

- **filter_name** – Name of the filter (see DECODERS)
- **params** – Parameters to the filter (will be written to /DecodeParms if not None)
- **allow_duplicates** – If None, silently ignore duplicate filters. If False, raise ValueError when attempting to add a duplicate filter. If True (default), duplicate filters are allowed.

compress()

Convenience method to add a /FlateDecode filter with default settings, if one is not already present.

Note: compression is not actually applied until the stream is written.

property is_embedded_file_stream**write_to_stream**(stream, handler=None, container_ref=None)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

pyhanko.pdf_utils.generic.**read_object**(stream, container_ref: [Dereferenceable](#), as_metadata_stream: bool = False) → [PdfObject](#)

Read a PDF object from an input stream.

Note: The *container_ref* parameter tells the API which reference to register when the returned object is modified in an incremental update. See also here [here](#) for further information.

Parameters

- **stream** – An input stream.
- **container_ref** – A reference to an object containing this one.
Note: It is perfectly possible (and common) for *container_ref* to resolve to the return value of this function.
- **as_metadata_stream** – Whether to dereference the object as an XMP metadata stream.

Returns

A [PdfObject](#).

pyhanko.pdf_utils.generic.**pdf_name**

alias of [NameObject](#)

pyhanko.pdf_utils.generic.**pdf_string**(string: Union[str, bytes, bytearray]) → Union[[ByteStringObject](#), [TextStringObject](#)]

Encode a string as a [TextStringObject](#) if possible, or a [ByteStringObject](#) otherwise.

Parameters

string – A Python string.

`pyhanko.pdf_utils.generic.pdf_date(dt: datetime) → TextStringObject`

Convert a datetime object into a PDF string. This function supports both timezone-aware and naive datetime objects.

Parameters

dt – The datetime object to convert.

Returns

A *TextStringObject* representing the datetime passed in.

class `pyhanko.pdf_utils.generic.TextStringEncoding(value)`

Bases: Enum

Encodings for PDF text strings.

PDF_DOC = None

PDFDocEncoding (one-byte character codes; PDF-specific).

UTF16BE = (b'\xfe\xff', 'utf-16be')

UTF-16BE encoding.

UTF8 = (b'\xef\xbb\xbf', 'utf-8')

UTF-8 encoding (PDF 2.0)

UTF16LE = (b'\xff\xfe', 'utf-16le')

UTF-16LE encoding.

Note: This is strictly speaking invalid in PDF 2.0, but some authoring tools output such strings anyway (presumably due to the fact that it's the default wide character encoding on Windows).

encode(string: str) → bytes

Encode a string with BOM.

Parameters

string – The string to encode.

Returns

The encoded string.

decode(string: Union[bytes, bytearray]) → str

Decode a string with BOM.

Parameters

string – The string to encode.

Returns

The encoded string.

Raises

UnicodeDecodeError – Raised if decoding fails.

class `pyhanko.pdf_utils.generic.EncryptedObjAccess(value)`

Bases: Enum

Defines what to do when an encrypted object is encountered when retrieving an object from a container.

PROXY = 0

Return the proxy object as-is, and leave further encryption/decryption handling to the caller.

TRANSPARENT = 1

Transparently decrypt the proxy's content (similarly wrapping any sub-containers in *DecryptedObjectProxy*, so this applies recursively).

Note: This is the default in most situations, since it's the least likely to get in the way of any APIs that are not explicitly aware of content encryption concerns.

RAW = 2

Return the underlying raw object as written, without attempting or deferring decryption.

class pyhanko.pdf_utils.generic.*DecryptedObjectProxy*(raw_object: PdfObject, handler)

Bases: PdfObject

Internal proxy class that allows transparent on-demand encryption of objects.

Warning: Most public-facing APIs won't leave you to deal with these *directly* (that's half the reason this class exists in the first place), and the API of this class is considered internal.

However, for reasons related to the historical PyPDF2 codebase from which pyHanko's object handling code ultimately derives, there are some Python builtins that might cause these wrapper objects to inadvertently "leak". Please [tell us about such cases](#) so we can make those types of access more convenient and robust.

Danger: The `__eq__` implementation on this class is not safe for general use, due to the fact that certain structures in PDF are exempt from encryption. Only compare proxy objects with `==` in areas of the document where these exemptions don't apply.

Parameters

- **raw_object** – A raw object, typically as-parsed from a PDF file.
- **handler** – The security handler governing this object.

raw_object: PdfObject

The underlying raw object, in its encrypted state.

property decrypted: PdfObject

The decrypted PDF object exposed as a property.

If this object is a container object, its constituent parts will be wrapped in *DecryptedObjectProxy* as well, in order to defer further decryption until the values are requested through a getter method on the container.

write_to_stream(stream, handler=None, container_ref=None)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

get_object()

Resolves indirect references.

Returns

self, unless an instance of *IndirectObject*.

property container_ref**pyhanko.pdf_utils.images module**

Utilities for embedding bitmap image data into PDF files.

The image data handling is done by [Pillow](#).

Note: Note that also here we only support a subset of what the PDF standard provides for. Most RGB and grayscale images (with or without transparency) that can be read by PIL/Pillow can be used without issue. PNG images with an indexed palette backed by one of these colour spaces can also be used.

Currently there is no support for CMYK images or (direct) support for embedding JPEG-encoded image data as such, but these features may be added later.

pyhanko.pdf_utils.images.pil_image(*img*: Image, *writer*: BasePdfFileWriter)

This function writes a PIL/Pillow Image object to a PDF file writer, as an image XObject.

Parameters

- **img** – A Pillow Image object
- **writer** – A PDF file writer

Returns

A reference to the image XObject written.

```
class pyhanko.pdf_utils.images.PdfImage(image: Union[Image, str], writer: Optional[BasePdfFileWriter]
                                         = None, resources: Optional[PdfResources] = None, name:
                                         Optional[str] = None, opacity=None, box:
                                         Optional[BoxConstraints] = None)
```

Bases: [PdfContent](#)

Wrapper class that implements the [PdfContent](#) interface for image objects.

Note: Instances of this class are reusable, in the sense that the implementation is aware of changes to the associated *writer* object. This allows the same image to be embedded into multiple files without instantiating a new *PdfImage* every time.

property image_ref: IndirectObject

Return a reference to the image XObject associated with this *PdfImage* instance. If no such reference is available, it will be created using *pil_image()*, and the result will be cached until the *writer* attribute changes (see *set_writer()*).

Returns

An indirect reference to an image XObject.

render() → bytes

Compile the content to graphics operators.

pyhanko.pdf_utils.incremental_writer module

Utility for writing incremental updates to existing PDF files.

class pyhanko.pdf_utils.incremental_writer.**IncrementalPdfFileWriter**(*input_stream*, *prev*: *Optional*[*PdfFileReader*] = *None*, *strict*=*True*)

Bases: *BasePdfFileWriter*

Class to incrementally update existing files.

This *BasePdfFileWriter* subclass encapsulates a *PdfFileReader* instance in addition to exposing an interface to add and modify PDF objects.

Incremental updates to a PDF file append modifications to the end of the file. This is critical when the original file contents are not to be modified directly (e.g. when it contains digital signatures). It has the additional advantage of providing an automatic audit trail of sorts.

Parameters

- **input_stream** – Input stream to read current revision from.
- **strict** – Ingest the source file in strict mode. The default is *True*.
- **prev** – Explicitly pass in a PDF reader. This parameter is internal API.

IO_CHUNK_SIZE = 4096

classmethod **from_reader**(*reader*: *PdfFileReader*) → *IncrementalPdfFileWriter*

Instantiate an incremental writer from a PDF file reader.

Parameters

reader – A *PdfFileReader* object with a PDF to extend.

ensure_output_version(*version*)

get_object(*ido*, *as_metadata_stream*: *bool* = *False*)

Retrieve the object associated with the provided reference from this PDF handler.

Parameters

- **ref** – An instance of *generic.Reference*.
- **as_metadata_stream** – Whether to dereference the object as an XMP metadata stream.

Returns

A PDF object.

mark_update(*obj_ref*: *Union*[*Reference*, *IndirectObject*])

Mark an object reference to be updated. This is only relevant for incremental updates, but is included as a no-op by default for interoperability reasons.

Parameters

obj_ref – An indirect object instance or a reference.

update_container(*obj*: *PdfObject*)

Mark the container of an object (as indicated by the *container_ref* attribute on *PdfObject*) for an update.

As with *mark_update()*, this only applies to incremental updates, but defaults to a no-op.

Parameters

obj – The object whose top-level container needs to be rewritten.

update_root()

Signal that the document catalog should be written to the output. Equivalent to calling `mark_update()` with `root_ref`.

set_info(*info*: *Optional*[*Union*[*IndirectObject*, *DictionaryObject*]])

Set the /Info entry of the document trailer.

Parameters

info – The new /Info dictionary, either as an indirect reference or as a *DictionaryObject*

set_custom_trailer_entry(*key*: *NameObject*, *value*: *PdfObject*)

Set a custom, unmanaged entry in the document trailer or cross-reference stream dictionary.

Warning: Calling this method to set an entry that is managed by pyHanko internally (info dictionary, document catalog, etc.) has undefined results.

Parameters

- **key** – Dictionary key to use in the trailer.
- **value** – Value to set

write(*stream*)

Write the contents of this PDF writer to a stream.

Parameters

stream – A writable output stream.

property document_meta_view: *DocumentMetadata***write_in_place()**

Write the updated file contents in-place to the same stream as the input stream. This obviously requires a stream supporting both reading and writing operations.

encrypt(*user_pwd*)

Method to handle updates to encrypted files.

This method handles decrypting of the original file, and makes sure the resulting updated file is encrypted in a compatible way. The standard mandates that updates to encrypted files be effected using the same encryption settings. In particular, incremental updates cannot remove file encryption.

Parameters

user_pwd – The original file’s user password.

Raises

PdfReadError – Raised when there is a problem decrypting the file.

encrypt_pubkey(*credential*: *EnvelopeKeyDecrypter*)

Method to handle updates to files encrypted using public-key encryption.

The same caveats as `encrypt()` apply here.

Parameters

credential – The *EnvelopeKeyDecrypter* handling the recipient’s private key.

Raises

PdfReadError – Raised when there is a problem decrypting the file.

stream_xrefs: `bool`

Boolean controlling whether or not the output file will contain its cross-references in stream format, or as a classical XRef table.

The default for new files is `True`. For incremental updates, the writer adapts to the system used in the previous iteration of the document (as mandated by the standard).

pyhanko.pdf_utils.layout module

Layout utilities (to be expanded)

exception `pyhanko.pdf_utils.layout.LayoutError(msg: str, *args)`

Bases: `ValueError`

Indicates an error in a layout computation.

exception `pyhanko.pdf_utils.layout.BoxSpecificationError(msg: Optional[str] = None)`

Bases: `LayoutError`

Raised when a box constraint is over/underspecified.

class `pyhanko.pdf_utils.layout.BoxConstraints(width=None, height=None, aspect_ratio: Optional[Fraction] = None)`

Bases: `object`

Represents a box of potentially variable width and height. Among other uses, this can be leveraged to produce a variably sized box with a fixed aspect ratio.

If width/height are not defined yet, they can be set by assigning to the `width` and `height` attributes.

property `width: int`

Returns

The width of the box.

Raises

`BoxSpecificationError` – if the box’s width could not be determined.

property `width_defined: bool`

Returns

True if the box currently has a well-defined width, False otherwise.

property `height: int`

Returns

The height of the box.

Raises

`BoxSpecificationError` – if the box’s height could not be determined.

property `height_defined: bool`

Returns

True if the box currently has a well-defined height, False otherwise.

property `aspect_ratio: Fraction`

Returns

The aspect ratio of the box.

Raises

BoxSpecificationError – if the box’s aspect ratio could not be determined.

property aspect_ratio_defined: bool

Returns

True if the box currently has a well-defined aspect ratio, False otherwise.

class pyhanko.pdf_utils.layout.**AxisAlignment**(*value*)

Bases: Enum

Class representing one-dimensional alignment along an axis.

ALIGN_MIN = 1

Align maximally towards the negative end of the axis.

ALIGN_MID = 2

Center content along the axis.

ALIGN_MAX = 3

Align maximally towards the positive end of the axis.

classmethod **from_x_align**(*align_str: str*) → *AxisAlignment*

Convert from a horizontal alignment config string.

Parameters

align_str – A string: ‘left’, ‘mid’ or ‘right’.

Returns

An *AxisAlignment* value.

Raises

ConfigurationError – on unexpected string inputs.

classmethod **from_y_align**(*align_str: str*) → *AxisAlignment*

Convert from a vertical alignment config string.

Parameters

align_str – A string: ‘bottom’, ‘mid’ or ‘top’.

Returns

An *AxisAlignment* value.

Raises

ConfigurationError – on unexpected string inputs.

property **flipped**

align(*container_len: int, inner_len: int, pre_margin, post_margin*) → int

class pyhanko.pdf_utils.layout.**Margins**(*left: int = 0, right: int = 0, top: int = 0, bottom: int = 0*)

Bases: *ConfigurableMixin*

Class describing a set of margins.

left: int = 0

right: int = 0

top: int = 0

bottom: `int = 0`

classmethod `uniform(num)`

Return a set of uniform margins.

Parameters

num – The uniform margin to apply to all four sides.

Returns

`Margins(num, num, num, num)`

static `effective(dim_name, container_len, pre, post)`

Internal helper method to compute effective margins.

effective_width(*width*)

Compute width without margins.

Parameters

width – The container width.

Returns

The width after subtracting the left and right margins.

Raises

[*LayoutError*](#) – if the container width is too short to accommodate the margins.

effective_height(*height*)

Compute height without margins.

Parameters

height – The container height.

Returns

The height after subtracting the top and bottom margins.

Raises

[*LayoutError*](#) – if the container height is too short to accommodate the margins.

classmethod `from_config(config_dict)`

Attempt to instantiate an object of the class on which it is called, by means of the configuration settings passed in.

First, we check that the keys supplied in the dictionary correspond to data fields on the current class. Then, the dictionary is processed using the `process_entries()` method. The resulting dictionary is passed to the initialiser of the current class as a kwargs dict.

Parameters

config_dict – A dictionary containing configuration values.

Returns

An instance of the class on which it is called.

Raises

[*ConfigurationError*](#) – when an unexpected configuration key is encountered or left unfilled, or when there is a problem processing one of the config values.

class `pyhanko.pdf_utils.layout.InnerScaling(value)`

Bases: `Enum`

Class representing a scaling convention.

NO_SCALING = 1

Never scale content.

STRETCH_FILL = 2

Scale content to fill the entire container.

STRETCH_TO_FIT = 3

Scale content while preserving aspect ratio until either the maximal width or maximal height is reached.

SHRINK_TO_FIT = 4

Scale content down to fit in the container, while preserving the original aspect ratio.

classmethod from_config(*config_str: str*) → *InnerScaling*

Convert from a configuration string.

Parameters

config_str – A string: ‘none’, ‘stretch-fill’, ‘stretch-to-fit’, ‘shrink-to-fit’

Returns

An *InnerScaling* value.

Raises

ConfigurationError – on unexpected string inputs.

```
class pyhanko.pdf_utils.layout.SimpleBoxLayoutRule(x_align: AxisAlignment, y_align:
                                                    AxisAlignment, margins: Margins =
                                                    Margins(left=0, right=0, top=0, bottom=0),
                                                    inner_content_scaling: InnerScaling =
                                                    InnerScaling.SHRINK_TO_FIT)
```

Bases: *ConfigurableMixin*

Class describing alignment, scaling and margin rules for a box positioned inside another box.

x_align: *AxisAlignment*

Horizontal alignment settings.

y_align: *AxisAlignment*

Vertical alignment settings.

margins: *Margins* = *Margins*(left=0, right=0, top=0, bottom=0)

Container (inner) margins. Defaults to all zeroes.

inner_content_scaling: *InnerScaling* = 4

Inner content scaling rule.

classmethod process_entries(*config_dict*)

Hook method that can modify the configuration dictionary to overwrite or tweak some of their values (e.g. to convert string parameters into more complex Python objects)

Subclasses that override this method should call `super().process_entries()`, and leave keys that they do not recognise untouched.

Parameters

config_dict – A dictionary containing configuration values.

Raises

ConfigurationError – when there is a problem processing a relevant entry.

substitute_margins(*new_margins: Margins*) → *SimpleBoxLayoutRule*

fit(*container_box*: [BoxConstraints](#), *inner_nat_width*: int, *inner_nat_height*: int) → [Positioning](#)

Position and possibly scale a box within a container, according to this layout rule.

Parameters

- **container_box** – [BoxConstraints](#) describing the container.
- **inner_nat_width** – The inner box’s natural width.
- **inner_nat_height** – The inner box’s natural height.

Returns

A [Positioning](#) describing the scaling & position of the lower left corner of the inner box.

class `pyhanko.pdf_utils.layout.Positioning`(*x_pos*: int, *y_pos*: int, *x_scale*: float, *y_scale*: float)

Bases: [ConfigurableMixin](#)

Class describing the position and scaling of an object in a container.

x_pos: int

Horizontal coordinate

y_pos: int

Vertical coordinate

x_scale: float

Horizontal scaling

y_scale: float

Vertical scaling

as_cm()

Convenience method to convert this [Positioning](#) into a PDF cm operator.

Returns

A byte string representing the cm operator corresponding to this [Positioning](#).

pyhanko.pdf_utils.metadata package

pyhanko.pdf_utils.metadata.model module

New in version 0.14.0.

This module contains the XMP data model classes and namespace registry, in addition to a simplified document meta-data model used for automated metadata management.


```
class pyhanko.pdf_utils.metadata.model.DocumentMetadata(title: ~typing.
Optional[~typing.Union[~pyhanko.pdf_utils.misc.StringWithLanguage, str]] = None, author: ~typing.
Optional[~typing.Union[~pyhanko.pdf_utils.misc.StringWithLanguage, str]] = None, subject: ~typing.
Optional[~typing.Union[~pyhanko.pdf_utils.misc.StringWithLanguage, str]] = None, keywords: ~typing.List[str] =
<factory>, creator: ~typing.
Optional[~typing.Union[~pyhanko.pdf_utils.misc.StringWithLanguage, str]] = None, created:
~typing.Optional[~typing.Union[str,
datetime.datetime]] = None,
last_modified:
~typing.Optional[~typing.Union[str,
datetime.datetime]] = 'now', xmp_extra:
~typing.
List[~pyhanko.pdf_utils.metadata.model.XmpStructure]
= <factory>, xmp_unmanaged: bool =
False)
```

Bases: object

Simple representation of document metadata. All entries are optional.

title: Optional[Union[StringWithLanguage, str]] = None

The document's title.

author: Optional[Union[StringWithLanguage, str]] = None

The document's author.

subject: Optional[Union[StringWithLanguage, str]] = None

The document's subject.

keywords: List[str]

Keywords associated with the document.

creator: Optional[Union[StringWithLanguage, str]] = None

The software that was used to author the document.

Note: This is distinct from the producer, which is typically used to indicate which PDF processor(s) interacted with the file.

created: Optional[Union[str, datetime]] = None

The time when the document was created. To set it to the current time, specify now.

last_modified: Optional[Union[str, datetime]] = 'now'

The time when the document was last modified. Defaults to the current time upon serialisation if not specified.

xmp_extra: List[XmpStructure]

Extra XMP metadata.

xmp_unmanaged: bool = False

Flag metadata as XMP-only. This means that the info dictionary will be cleared out as much as possible, and that all attributes other than `xmp_extra` will be ignored when updating XMP metadata.

Note: The last-modified date and producer entries in the info dictionary will still be updated.

Note: *DocumentMetadata* represents a data model that is much more simple than what XMP is actually capable of. You can use this flag if you need more fine-grained control.

view_over(*base*: *DocumentMetadata*)

`pyhanko.pdf_utils.metadata.model.VENDOR = 'pyHanko 0.14.0'`

pyHanko version identifier in textual form

`pyhanko.pdf_utils.metadata.model.MetaString`

A regular string, a string with a language code, or nothing at all.

alias of `Optional[Union[StringWithLanguage, str]]`

class `pyhanko.pdf_utils.metadata.model.ExpandedName`(*ns*: *str*, *local_name*: *str*)

Bases: `object`

An expanded XML name.

ns: `str`

The URI of the namespace in which the name resides.

local_name: `str`

The local part of the name.

class `pyhanko.pdf_utils.metadata.model.Qualifiers`(*quals*: *Dict*[*ExpandedName*, *XmpValue*])

Bases: `object`

XMP value qualifiers wrapper. Implements `__getitem__`. Note that `xml:lang` gets special treatment.

Parameters

quals – The qualifiers to model.

classmethod of (**lst*: *Tuple*[*ExpandedName*, *XmpValue*]) → *Qualifiers*

Construct a *Qualifiers* object from a list of name-value pairs.

Parameters

lst – A list of name-value pairs.

Returns

A *Qualifiers* object.

classmethod `lang_as_qual`(*lang*: *Optional*[*str*]) → *Qualifiers*

Construct a *Qualifiers* object that only wraps a language qualifier.

Parameters

lang – A language code.

Returns

A *Qualifiers* object.

iter_quals(*with_lang*: *bool* = *True*) → *Iterable*[*Tuple*[*ExpandedName*, *XmpValue*]]

Iterate over all qualifiers.

Parameters

with_lang – Include the language qualifier.

Returns**property lang:** `Optional[str]`

Retrieve the language qualifier, if any.

property has_non_lang_qual: `bool`

Check if there are any non-language qualifiers.

```
class pyhanko.pdf_utils.metadata.model.XmpValue(value: ~typing.Union[~pyhanko.pdf_utils.metadata.model.XmpStructure,
~pyhanko.pdf_utils.metadata.model.XmpArray,
~pyhanko.pdf_utils.metadata.model.XmpUri, str],
qualifiers: ~pyhanko.pdf_utils.metadata.model.Qualifiers =
<factory>)
```

Bases: `object`

A general XMP value, potentially with qualifiers.

value: `Union[XmpStructure, XmpArray, XmpUri, str]`

The value.

qualifiers: `Qualifiers`

Qualifiers that apply to the value.

```
class pyhanko.pdf_utils.metadata.model.XmpStructure(fields: Dict[ExpandedName, XmpValue])
```

Bases: `object`A generic XMP structure value. Implements `__getitem__` for field access.**Parameters****fields** – The structure’s fields.

```
classmethod of(*lst: Tuple[ExpandedName, XmpValue]) → XmpStructure
```

Construct an `XmpStructure` from a list of name-value pairs.**Parameters****lst** – A list of name-value pairs.**Returns**An an `XmpStructure`.

```
class pyhanko.pdf_utils.metadata.model.XmpArrayType(value)
```

Bases: `Enum`

XMP array types.

ORDERED = `'Seq'`

Ordered array.

UNORDERED = `'Bag'`

Unordered array.

ALTERNATIVE = `'Alt'`

Alternative array.

as_rdf() → `ExpandedName`

Render the type as an XML name.

```
class pyhanko.pdf_utils.metadata.model.XmpArray(array_type: XmpArrayType, entries:
                                             List[XmpValue])
```

Bases: object

An XMP array.

array_type: [XmpArrayType](#)

The type of the array.

entries: [List\[XmpValue\]](#)

The entries in the array.

classmethod ordered(*lst: Iterable[XmpValue]*) → [XmpArray](#)

Convert a list to an ordered XMP array.

Parameters

lst – An iterable of XMP values.

Returns

An ordered [XmpArray](#).

classmethod unordered(*lst: Iterable[XmpValue]*) → [XmpArray](#)

Convert a list to an unordered XMP array.

Parameters

lst – An iterable of XMP values.

Returns

An unordered [XmpArray](#).

classmethod alternative(*lst: Iterable[XmpValue]*) → [XmpArray](#)

Convert a list to an alternative XMP array.

Parameters

lst – An iterable of XMP values.

Returns

An alternative [XmpArray](#).

```
pyhanko.pdf_utils.metadata.model.NS = {'dc': 'http://purl.org/dc/elements/1.1/', 'pdf':
'http://ns.adobe.com/pdf/1.3/', 'pdfaExtension':
'http://www.aiim.org/pdfa/ns/extension/', 'pdfaProperty':
'http://www.aiim.org/pdfa/ns/property#', 'pdfaSchema':
'http://www.aiim.org/pdfa/ns/schema#', 'pdfaid': 'http://www.aiim.org/pdfa/ns/id/',
'pdfuaid': 'http://www.aiim.org/pdfua/ns/id/', 'rdf':
'http://www.w3.org/1999/02/22-rdf-syntax-ns#', 'x': 'adobe:ns:meta/', 'xml':
'http://www.w3.org/XML/1998/namespace', 'xmp': 'http://ns.adobe.com/xap/1.0/'}
```

Known namespaces and their customary prefixes.

```
pyhanko.pdf_utils.metadata.model.XML_LANG = http://www.w3.org/XML/1998/namespace/lang
```

lang in the xml namespace.

```
pyhanko.pdf_utils.metadata.model.RDF_RDF = http://www.w3.org/1999/02/22-rdf-syntax-ns#RDF
```

RDF in the rdf namespace.

```
pyhanko.pdf_utils.metadata.model.RDF_SEQ = http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq
```

Seq in the rdf namespace.

`pyhanko.pdf_utils.metadata.model.RDF_BAG = http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag`
Bag in the rdf namespace.

`pyhanko.pdf_utils.metadata.model.RDF_ALT = http://www.w3.org/1999/02/22-rdf-syntax-ns#Alt`
Alt in the rdf namespace.

`pyhanko.pdf_utils.metadata.model.RDF_LI = http://www.w3.org/1999/02/22-rdf-syntax-ns#li`
li in the rdf namespace.

`pyhanko.pdf_utils.metadata.model.RDF_VALUE =`
`http://www.w3.org/1999/02/22-rdf-syntax-ns#value`
value in the rdf namespace.

`pyhanko.pdf_utils.metadata.model.RDF_RESOURCE =`
`http://www.w3.org/1999/02/22-rdf-syntax-ns#resource`
resource in the rdf namespace.

`pyhanko.pdf_utils.metadata.model.RDF_PARSE_TYPE =`
`http://www.w3.org/1999/02/22-rdf-syntax-ns#parseType`
parseType in the rdf namespace.

`pyhanko.pdf_utils.metadata.model.RDF_ABOUT =`
`http://www.w3.org/1999/02/22-rdf-syntax-ns#about`
about in the rdf namespace.

`pyhanko.pdf_utils.metadata.model.RDF_DESCRIPTION =`
`http://www.w3.org/1999/02/22-rdf-syntax-ns#Description`
Description in the rdf namespace.

`pyhanko.pdf_utils.metadata.model.DC_TITLE = http://purl.org/dc/elements/1.1/title`
title in the dc namespace.

`pyhanko.pdf_utils.metadata.model.DC_CREATOR = http://purl.org/dc/elements/1.1/creator`
creator in the dc namespace.

`pyhanko.pdf_utils.metadata.model.DC_DESCRIPTION =`
`http://purl.org/dc/elements/1.1/description`
description in the dc namespace.

`pyhanko.pdf_utils.metadata.model.PDF_PRODUCER = http://ns.adobe.com/pdf/1.3/Producer`
Producer in the pdf namespace.

`pyhanko.pdf_utils.metadata.model.PDF_KEYWORDS = http://ns.adobe.com/pdf/1.3/keywords`
keywords in the pdf namespace.

`pyhanko.pdf_utils.metadata.model.X_XMPMETA = adobe:ns:meta/xmpmeta`
xmpmeta in the x namespace.

`pyhanko.pdf_utils.metadata.model.X_XMPTK = adobe:ns:meta/xmptk`
xmptk in the x namespace.

`pyhanko.pdf_utils.metadata.model.XMP_CREATORTOOL =`
`http://ns.adobe.com/xap/1.0/CreatorTool`
CreatorTool in the xmp namespace.

`pyhanko.pdf_utils.metadata.model.XMP_CREATEDATE = http://ns.adobe.com/xap/1.0/CreateDate`
CreateDate in the xmp namespace.

`pyhanko.pdf_utils.metadata.model.XMP_MODDATE = http://ns.adobe.com/xap/1.0/ModifyDate`
ModifyDate in the xmp namespace.

pyhanko.pdf_utils.misc module

Utility functions for PDF library. Taken from PyPDF2 with modifications and additions, see [here](#) for the original license of the PyPDF2 project.

Generally, all of these constitute internal API, except for the exception classes.

exception `pyhanko.pdf_utils.misc.PdfError(msg: str, *args)`

Bases: `Exception`

exception `pyhanko.pdf_utils.misc.PdfReadError(msg: str, *args)`

Bases: `PdfError`

exception `pyhanko.pdf_utils.misc.PdfStrictReadError(msg: str, *args)`

Bases: `PdfReadError`

exception `pyhanko.pdf_utils.misc.PdfWriteError(msg: str, *args)`

Bases: `PdfError`

exception `pyhanko.pdf_utils.misc.PdfStreamError(msg: str, *args)`

Bases: `PdfReadError`

exception `pyhanko.pdf_utils.misc.IndirectObjectExpected(msg: Optional[str] = None)`

Bases: `PdfReadError`

`pyhanko.pdf_utils.misc.get_and_apply(dictionary: dict, key, function: Callable, *, default=None)`

class `pyhanko.pdf_utils.misc.OrderedEnum(value)`

Bases: `Enum`

Ordered enum (from the Python documentation)

class `pyhanko.pdf_utils.misc.StringWithLanguage(value: str, lang_code: Optional[str] = None, country_code: Optional[str] = None)`

Bases: `object`

A string with a language attached to it.

value: `str`

lang_code: `Optional[str] = None`

country_code: `Optional[str] = None`

`pyhanko.pdf_utils.misc.is_regular_character(byte_value: int)`

`pyhanko.pdf_utils.misc.read_non_whitespace(stream, seek_back=False, allow_eof=False)`

Finds and reads the next non-whitespace character (ignores whitespace).

`pyhanko.pdf_utils.misc.read_until_whitespace(stream, maxchars=None)`

Reads non-whitespace characters and returns them. Stops upon encountering whitespace or when maxchars is reached.

`pyhanko.pdf_utils.misc.read_until_regex(stream, regex, ignore_eof=False)`

Reads until the regular expression pattern matched (ignore the match) Raise PdfStreamError on premature end-of-file. :param bool ignore_eof: If true, ignore end-of-line and return immediately :param regex: regex to match :param stream: stream to search

`pyhanko.pdf_utils.misc.skip_over_whitespace(stream, stop_after_eol=False) → bool`

Similar to readNonWhitespace, but returns a Boolean if more than one whitespace character was read.

Will return the cursor to before the first non-whitespace character encountered, or after the first end-of-line sequence if one is encountered.

`pyhanko.pdf_utils.misc.skip_over_comment(stream) → bool`

`pyhanko.pdf_utils.misc.instance_test(cls)`

`pyhanko.pdf_utils.misc.peek(itr)`

`pyhanko.pdf_utils.misc.assert_writable_and_random_access(output)`

Raise an error if the buffer in question is not writable, and return a boolean to indicate whether it supports random-access reading.

Parameters

output –

Returns

`pyhanko.pdf_utils.misc.prepare_rw_output_stream(output)`

Prepare an output stream that supports both reading and writing. Intended to be used for writing & updating signed files: when producing a signature, we render the PDF to a byte buffer with placeholder values for the signature data, or straight to the provided output stream if possible.

More precisely: this function will return the original output stream if it is writable, readable and seekable. If the output parameter is None, not readable or not seekable, this function will return a BytesIO instance instead. If the output parameter is not None and not writable, IOError will be raised.

Parameters

output – A writable file-like object, or None.

Returns

A file-like object that supports reading, writing and seeking.

`pyhanko.pdf_utils.misc.finalise_output(orig_output, returned_output)`

Several internal APIs transparently replaces non-readable/seekable buffers with BytesIO for signing operations, but we don't want to expose that to the public API user. This internal API function handles the unwrapping.

`pyhanko.pdf_utils.misc.DEFAULT_CHUNK_SIZE = 4096`

Default chunk size for stream I/O.

`pyhanko.pdf_utils.misc.chunked_write(temp_buffer: bytearray, stream, output, max_read=None)`

`pyhanko.pdf_utils.misc.chunked_digest(temp_buffer: bytearray, stream, md, max_read=None)`

`pyhanko.pdf_utils.misc.chunk_stream(temp_buffer: bytearray, stream, max_read=None)`

`class pyhanko.pdf_utils.misc.ConsList(head: object, tail: 'ConsList' = None)`

Bases: object

head: object

tail: *ConsList* = None

```
static empty() → ConsList
static sing(value) → ConsList
cons(head)

class pyhanko.pdf_utils.misc.Singleton(name, bases, dct)
    Bases: type
pyhanko.pdf_utils.misc.rd(x)
pyhanko.pdf_utils.misc.isoparse(dt_str: str) → datetime
```

pyhanko.pdf_utils.reader module

Utility to read PDF files. Contains code from the PyPDF2 project; see [here](#) for the original license.

The implementation was tweaked with the express purpose of facilitating historical inspection and auditing of PDF files with multiple revisions through incremental updates. This comes at a cost, and future iterations of this module may offer more flexibility in terms of the level of detail with which file size is scrutinised.

```
class pyhanko.pdf_utils.reader.PdfFileReader(stream, strict=True)
```

Bases: [PdfHandler](#)

Class implementing functionality to read a PDF file and cache certain data about it.

last_startxref = None

has_xref_stream = False

xrefs: [XRefCache](#)

property document_meta_view: [DocumentMetadata](#)

property input_version

property trailer_view: [DictionaryObject](#)

Returns a view of the document trailer of the document represented by this [PdfHandler](#) instance.

The view is effectively read-only, in the sense that any writes will not be reflected in the actual trailer (if the handler supports writing, that is).

Returns

A [generic.DictionaryObject](#) representing the current state of the document trailer.

property root_ref: [Reference](#)

Returns

A reference to the document catalog of this PDF handler.

property document_id: Tuple[bytes, bytes]

get_historical_root(revision: int)

Get the document catalog for a specific revision.

Parameters

revision – The revision to query, the oldest one being 0.

Returns

The value of the document catalog dictionary for that revision.

property total_revisions: int

Returns

The total number of revisions made to this file.

get_object(*ref*, *revision=None*, *never_decrypt=False*, *transparent_decrypt=True*, *as_metadata_stream=False*)

Read an object from the input stream.

Parameters

- **ref** – *Reference* to the object.
- **revision** – Revision number, to return the historical value of a reference. This always bypasses the cache. The oldest revision is numbered 0. See also *HistoricalResolver*.
- **never_decrypt** – Skip decryption step (only needed for parsing /Encrypt)
- **transparent_decrypt** – If True, all encrypted objects are transparently decrypted by default (in the sense that a user of the API in a PyPDF2 compatible way would only “see” decrypted objects). If False, this method may return a proxy object that still allows access to the “original”.

Danger: The encryption parameters are considered internal, undocumented API, and subject to change without notice.

- **as_metadata_stream** – Whether to dereference the object as an XMP metadata stream.

Returns

A *PdfObject*.

Raises

PdfReadError – Raised if there is an issue reading the object from the file.

cache_get_indirect_object(*generation*, *idnum*)

cache_indirect_object(*generation*, *idnum*, *obj*)

read()

decrypt(*password: Union[str, bytes]*) → *AuthResult*

When using an encrypted PDF file with the standard PDF encryption handler, this function will allow the file to be decrypted. It checks the given password against the document’s user password and owner password, and then stores the resulting decryption key if either password is correct.

Both legacy encryption schemes and PDF 2.0 encryption (based on AES-256) are supported.

Danger: Supplying either user or owner password will work. Cryptographically, both allow the decryption key to be computed, but processors are expected to adhere to the /P flags in the encryption dictionary when accessing a file with the user password. Currently, pyHanko does not enforce these restrictions, but it may in the future.

Danger: One should also be aware that the legacy encryption schemes used prior to PDF 2.0 are (very) weak, and we only support them for compatibility reasons. Under no circumstances should these still be used to encrypt new files.

Parameters

password – The password to match.

decrypt_pubkey(*credential*: [EnvelopeKeyDecrypter](#)) → [AuthResult](#)

Decrypt a PDF file encrypted using public-key encryption by providing a credential representing the private key of one of the recipients.

Danger: The same caveats as in [decrypt\(\)](#) w.r.t. permission handling apply to this method.

Danger: The robustness of the public key cipher being used is not the only factor in the security of public-key encryption in PDF. The standard still permits weak schemes to encrypt the actual file data and file keys. PyHanko uses sane defaults everywhere, but other software may not.

Parameters

credential – The [EnvelopeKeyDecrypter](#) handling the recipient's private key.

property encrypted

Returns

True if a document is encrypted, False otherwise.

get_historical_resolver(*revision*: int) → [HistoricalResolver](#)

Return a [PdfHandler](#) instance that provides a view on the file at a specific revision.

Parameters

revision – The revision number to use, with 0 being the oldest.

Returns

An instance of [HistoricalResolver](#).

property embedded_signatures

Returns

The signature objects embedded in this document, in signing order; see [EmbeddedPdfSignature](#).

property embedded_regular_signatures

Returns

The signature objects of type /Sig embedded in this document, in signing order; see [EmbeddedPdfSignature](#).

property embedded_timestamp_signatures

Returns

The signature objects of type /DocTimeStamp embedded in this document, in signing order; see [EmbeddedPdfSignature](#).

class `pyhanko.pdf_utils.reader.HistoricalResolver`(*reader*: [PdfFileReader](#), *revision*)

Bases: [PdfHandler](#)

[PdfHandler](#) implementation that provides a view on a particular revision of a PDF file.

Instances of [HistoricalResolver](#) should be created by calling the [get_historical_resolver\(\)](#) method on a [PdfFileReader](#) object.

Instances of this class cache the result of `get_object()` calls.

Danger: This class is documented, but is nevertheless considered internal API, and easy to misuse.

In particular, the `container_ref` attribute must *not* be relied upon for objects retrieved from a `HistoricalResolver`. Internally, it is only used to make lazy decryption work in historical revisions.

Note: Be aware that instances of this class transparently rewrite the PDF handler associated with any reference objects returned from the reader, so calling `get_object()` on an indirect reference object will cause the reference to be resolved within the selected revision.

property `document_meta_view`: `DocumentMetadata`

property `document_id`: `Tuple[bytes, bytes]`

property `trailer_view`: `DictionaryObject`

Returns a view of the document trailer of the document represented by this `PdfHandler` instance.

The view is effectively read-only, in the sense that any writes will not be reflected in the actual trailer (if the handler supports writing, that is).

Returns

A `generic.DictionaryObject` representing the current state of the document trailer.

get_object(`ref`: `Reference`, `as_metadata_stream`: `bool = False`)

Retrieve the object associated with the provided reference from this PDF handler.

Parameters

- **ref** – An instance of `generic.Reference`.
- **as_metadata_stream** – Whether to dereference the object as an XMP metadata stream.

Returns

A PDF object.

property `root_ref`: `Reference`

Returns

A reference to the document catalog of this PDF handler.

explicit_refs_in_revision()

refs_freed_in_revision()

object_streams_used()

is_ref_available(`ref`: `Reference`) → `bool`

Check if the reference in question was in scope for this revision. This call doesn't care about the specific semantics of free vs. used objects; it conservatively answers 'no' in any situation where the object ID `_could_` have been assigned by the revision in question.

Parameters

ref – A reference object (usually one written to by a newer revision)

Returns

True if the reference is unassignable, False otherwise.

collect_dependencies(*obj*: PdfObject, *since_revision*=None)

Collect all indirect references used by an object and its descendants.

Parameters

- **obj** – The object to inspect.
- **since_revision** – Optionally specify a revision number that tells the scanner to only include objects IDs that were added in that revision or later.

Warning: In particular, this means that the scanner will not recurse into older objects either.

Returns

A set of [Reference](#) objects.

`pyhanko.pdf_utils.reader.parse_catalog_version(version_str) → Optional[Tuple[int, int]]`

class `pyhanko.pdf_utils.reader.RawPdfPath(*path: Union[str, int])`

Bases: object

Class to model raw paths in a file.

This class is internal API.

access_on(*from_obj*, *dereference_last*=True) → PdfObject

access_reference_on(*from_obj*) → Reference

`pyhanko.pdf_utils.reader.process_data_at_eof(stream) → int`

Auxiliary function that reads backwards from the current position in a stream to find the EOF marker and startxref value

This is internal API.

Parameters

stream – A stream to read from

Returns

The value of the startxref pointer, if found. Otherwise a PdfReadError is raised.

pyhanko.pdf_utils.rw_common module

Utilities common to reading and writing PDF files.

class `pyhanko.pdf_utils.rw_common.PdfHandler`

Bases: object

Abstract class providing a general interface for quering objects in PDF readers and writers alike.

get_object(*ref*: Reference, *as_metadata_stream*: bool = False)

Retrieve the object associated with the provided reference from this PDF handler.

Parameters

- **ref** – An instance of [generic.Reference](#).
- **as_metadata_stream** – Whether to dereference the object as an XMP metadata stream.

Returns

A PDF object.

property trailer_view: *DictionaryObject*

Returns a view of the document trailer of the document represented by this *PdfHandler* instance.

The view is effectively read-only, in the sense that any writes will not be reflected in the actual trailer (if the handler supports writing, that is).

Returns

A *generic.DictionaryObject* representing the current state of the document trailer.

property document_meta_view: *DocumentMetadata*

property root_ref: *Reference*

Returns

A reference to the document catalog of this PDF handler.

property root: *DictionaryObject*

Returns

The document catalog of this PDF handler.

property document_id: *Tuple[bytes, bytes]*

find_page_container(*page_ix*)

Retrieve the node in the page tree containing the page with index *page_ix*, along with the necessary objects to modify it in an incremental update scenario.

Parameters

page_ix – The (zero-indexed) number of the page for which we want to retrieve the parent. A negative number counts pages from the back of the document, with index *-1* referring to the last page.

Returns

A triple with the */Pages* object (or a reference to it), the index of the target page in said */Pages* object, and a (possibly inherited) resource dictionary.

find_page_for_modification(*page_ix*)

Retrieve the page with index *page_ix* from the page tree, along with the necessary objects to modify it in an incremental update scenario.

Parameters

page_ix – The (zero-indexed) number of the page to retrieve. A negative number counts pages from the back of the document, with index *-1* referring to the last page.

Returns

A tuple with a reference to the page object and a (possibly inherited) resource dictionary.

pyhanko.pdf_utils.text module

Utilities related to text rendering & layout.

```
class pyhanko.pdf_utils.text.TextStyle(font: ~pyhanko.pdf_utils.font.api.FontEngineFactory =
                                       <factory>, font_size: int = 10, leading: ~typing.Optional[int] =
                                       None)
```

Bases: [ConfigurableMixin](#)

Container for basic test styling settings.

font: [FontEngineFactory](#)

The [FontEngineFactory](#) to be used for this text style. Defaults to Courier (as a non-embedded standard font).

font_size: int = 10

Font size to be used.

leading: int = None

Text leading. If None, the [font_size](#) parameter is used instead.

classmethod process_entries(*config_dict*)

Hook method that can modify the configuration dictionary to overwrite or tweak some of their values (e.g. to convert string parameters into more complex Python objects)

Subclasses that override this method should call `super().process_entries()`, and leave keys that they do not recognise untouched.

Parameters

config_dict – A dictionary containing configuration values.

Raises

[ConfigurationError](#) – when there is a problem processing a relevant entry.

```
class pyhanko.pdf_utils.text.TextBoxStyle(font: ~pyhanko.pdf_utils.font.api.FontEngineFactory =
                                           <factory>, font_size: int = 10, leading: ~typing.Optional[int]
                                           = None, border_width: int = 0, box_layout_rule: ~typ-
                                           ing.Optional[~pyhanko.pdf_utils.layout.SimpleBoxLayoutRule]
                                           = None, vertical_text: bool = False)
```

Bases: [TextStyle](#)

Extension of [TextStyle](#) for use in text boxes.

border_width: int = 0

Border width, if applicable.

box_layout_rule: [SimpleBoxLayoutRule](#) = None

Layout rule to nest the text within its containing box.

Warning: This only affects the position of the text object, not the alignment of the text within.

vertical_text: bool = False

Switch layout code to vertical mode instead of horizontal mode.

```
class pyhanko.pdf_utils.text.TextBox(style: TextBoxStyle, writer, resources: Optional[PdfResources] =
                                     None, box: Optional[BoxConstraints] = None, font_name='F1')
```

Bases: *PdfContent*

Implementation of a text box that implements the *PdfContent* interface.

Note: Text boxes currently don't offer automatic word wrapping.

```
put_string_line(txt)
```

```
property content_lines
```

Returns

Text content of the text box, broken up into lines.

```
property content
```

Returns

The actual text content of the text box. This is a modifiable property.

In textboxes that don't have a fixed size, setting this property can cause the text box to be resized.

```
property leading
```

Returns

The effective leading value, i.e. the *leading* attribute of the associated *TextBoxStyle*, or *font_size* if not specified.

```
render()
```

Compile the content to graphics operators.

pyhanko.pdf_utils.writer module

Utilities for writing PDF files. Contains code from the PyPDF2 project; see [here](#) for the original license.

```
class pyhanko.pdf_utils.writer.BasePdfFileWriter(root, info, document_id, obj_id_start=0,
                                                  stream_xrefs=True)
```

Bases: *PdfHandler*

Base class for PDF writers.

```
output_version = (1, 7)
```

Output version to be declared in the output file.

```
stream_xrefs: bool
```

Boolean controlling whether or not the output file will contain its cross-references in stream format, or as a classical XRef table.

The default for new files is True. For incremental updates, the writer adapts to the system used in the previous iteration of the document (as mandated by the standard).

```
get_subset_collection(base_postscript_name: str)
```

```
property document_meta: DocumentMetadata
```

property document_meta_view: [DocumentMetadata](#)

ensure_output_version(*version*)

set_info(*info*: *Optional[Union[IndirectObject, DictionaryObject]]*)

Set the /Info entry of the document trailer.

Parameters

info – The new /Info dictionary, either as an indirect reference or as a [DictionaryObject](#)

set_custom_trailer_entry(*key*: [NameObject](#), *value*: [PdfObject](#))

Set a custom, unmanaged entry in the document trailer or cross-reference stream dictionary.

Warning: Calling this method to set an entry that is managed by pyHanko internally (info dictionary, document catalog, etc.) has undefined results.

Parameters

- **key** – Dictionary key to use in the trailer.
- **value** – Value to set

property document_id: [Tuple\[bytes, bytes\]](#)

mark_update(*obj_ref*: *Union[Reference, IndirectObject]*)

Mark an object reference to be updated. This is only relevant for incremental updates, but is included as a no-op by default for interoperability reasons.

Parameters

obj_ref – An indirect object instance or a reference.

update_container(*obj*: [PdfObject](#))

Mark the container of an object (as indicated by the *container_ref* attribute on [PdfObject](#)) for an update.

As with [mark_update\(\)](#), this only applies to incremental updates, but defaults to a no-op.

Parameters

obj – The object whose top-level container needs to be rewritten.

property root_ref: [Reference](#)

Returns

A reference to the document catalog.

update_root()

Signal that the document catalog should be written to the output. Equivalent to calling [mark_update\(\)](#) with [root_ref](#).

register_extension(*ext*: [DeveloperExtension](#))

get_object(*ido*, *as_metadata_stream*: *bool = False*)

Retrieve the object associated with the provided reference from this PDF handler.

Parameters

- **ref** – An instance of [generic.Reference](#).
- **as_metadata_stream** – Whether to dereference the object as an XMP metadata stream.

Returns

A PDF object.

allocate_placeholder() → *IndirectObject*

Allocate an object reference to populate later. Calls to *get_object()* for this reference will return *NullObject* until it is populated using *add_object()*.

This method is only relevant in certain advanced contexts where an object ID needs to be known before the object it refers to can be built; chances are you'll never need it.

Returns

A *IndirectObject* instance referring to the object just allocated.

add_object(obj, obj_stream: Optional[ObjectStream] = None, idnum=None) → *IndirectObject*

Add a new object to this writer.

Parameters

- **obj** – The object to add.
- **obj_stream** – An object stream to add the object to.
- **idnum** – Manually specify the object ID of the object to be added. This is only allowed for object IDs that have previously been allocated using *allocate_placeholder()*.

Returns

A *IndirectObject* instance referring to the object just added.

prepare_object_stream(compress=True)

Prepare and return a new *ObjectStream* object.

Parameters

compress – Indicates whether the resulting object stream should be compressed.

Returns

An *ObjectStream* object.

property trailer_view: *DictionaryObject*

Returns a view of the document trailer of the document represented by this *PdfHandler* instance.

The view is effectively read-only, in the sense that any writes will not be reflected in the actual trailer (if the handler supports writing, that is).

Returns

A *generic.DictionaryObject* representing the current state of the document trailer.

write(stream)

Write the contents of this PDF writer to a stream.

Parameters

stream – A writable output stream.

register_annotation(page_ref, annot_ref)

Register an annotation to be added to a page. This convenience function takes care of calling *mark_update()* where necessary.

Parameters

- **page_ref** – Reference to the page object involved.
- **annot_ref** – Reference to the annotation object to be added.

insert_page(*new_page*, *after*=None)

Insert a page object into the tree.

Parameters

- **new_page** – Page object to insert.
- **after** – Page number (zero-indexed) after which to insert the page.

Returns

A reference to the newly inserted page.

import_object(*obj*: PdfObject, *obj_stream*: Optional[ObjectStream] = None) → PdfObject

Deep-copy an object into this writer, dealing with resolving indirect references in the process.

Danger: The table mapping indirect references in the input to indirect references in the writer is not preserved between calls. Concretely, this means that invoking `import_object()` twice on the same input reader may cause object duplication.

Parameters

- **obj** – The object to import.
- **obj_stream** – The object stream to import objects into.

Note: Stream objects and bare references will not be put into the object stream; the standard forbids this.

Returns

The object as associated with this writer. If the input object was an indirect reference, a dictionary (incl. streams) or an array, the returned value will always be a new instance.

import_page_as_xobject(*other*: PdfHandler, *page_ix*=0, *inherit_filters*=True)

Import a page content stream from some other *PdfHandler* into the current one as a form XObject.

Parameters

- **other** – A *PdfHandler*
- **page_ix** – Index of the page to copy (default: 0)
- **inherit_filters** – Inherit the content stream's filters, if present.

Returns

An *IndirectObject* referring to the page object as added to the current reader.

add_stream_to_page(*page_ix*, *stream_ref*, *resources*=None, *prepend*=False)

Append an indirect stream object to a page in a PDF as a content stream.

Parameters

- **page_ix** – Index of the page to modify. The first page has index 0.
- **stream_ref** – *IndirectObject* reference to the stream object to add.
- **resources** – Resource dictionary containing resources to add to the page's existing resource dictionary.

- **prepend** – Prepend the content stream to the list of content streams, as opposed to appending it to the end. This has the effect of causing the stream to be rendered underneath the already existing content on the page.

Returns

An *IndirectObject* reference to the page object that was modified.

add_content_to_page(*page_ix*, *pdf_content*: PdfContent, *prepend*=False)

Convenience wrapper around *add_stream_to_page()* to turn a PdfContent instance into a page content stream.

Parameters

- **page_ix** – Index of the page to modify. The first page has index 0.
- **pdf_content** – An instance of PdfContent
- **prepend** – Prepend the content stream to the list of content streams, as opposed to appending it to the end. This has the effect of causing the stream to be rendered underneath the already existing content on the page.

Returns

An *IndirectObject* reference to the page object that was modified.

merge_resources(*orig_dict*, *new_dict*) → bool

Update an existing resource dictionary object with data from another one. Returns True if the original dict object was modified directly.

The caller is responsible for avoiding name conflicts with existing resources.

class pyhanko.pdf_utils.writer.PageObject(*contents*, *media_box*, *resources*=None)

Bases: *DictionaryObject*

Subclass of *DictionaryObject* that handles some of the initialisation boilerplate for page objects.

class pyhanko.pdf_utils.writer.PdfFileWriter(*stream_xrefs*=True, *init_page_tree*=True, *info*=None)

Bases: *BasePdfFileWriter*

Class to write new PDF files.

encrypt(*owner_pass*, *user_pass*=None, ***kwargs*)

Mark this document to be encrypted with PDF 2.0 encryption (AES-256).

Caution: While pyHanko supports legacy PDF encryption as well, the API to create new documents using outdated encryption is left largely undocumented on purpose to discourage its use.

This caveat does *not* apply to incremental updates added to existing documents.

Danger: The PDF 2.0 standard mandates AES-256 in CBC mode, and also includes 12 bytes of known plaintext by design. This implies that a sufficiently knowledgeable attacker can inject arbitrary content into your encrypted files without knowledge of the password.

Adding a digital signature to the encrypted document is **not** a foolproof way to deal with this either, since most viewers will still allow the document to be opened before signatures are validated, and therefore end users are still exposed to potentially malicious content.

Until the standard supports authenticated encryption schemes, you should **never** rely on its encryption provisions if tampering is a concern.

Parameters

- **owner_pass** – The desired owner password.
- **user_pass** – The desired user password (defaults to the owner password if not specified)
- **kwargs** – Other keyword arguments to be passed to [StandardSecurityHandler.build_from_pw\(\)](#).

encrypt_pubkey(*recipients: List[Certificate]*, ***kwargs*)

Mark this document to be encrypted with PDF 2.0 public key encryption. The certificates passed in should be RSA certificates.

PyHanko defaults to AES-256 to encrypt the actual file contents. The seed used to derive the file encryption key is also encrypted using AES-256 and bundled in a CMS EnvelopedData object. The envelope key is then encrypted separately for each recipient, using their respective public keys.

Caution: The caveats for [encrypt\(\)](#) also apply here.

Parameters

- **recipients** – Certificates of the recipients that should be able to decrypt the document.
- **kwargs** – Other keyword arguments to be passed to [PubKeySecurityHandler.build_from_certs\(\)](#).

stream_xrefs: `bool`

Boolean controlling whether or not the output file will contain its cross-references in stream format, or as a classical XRef table.

The default for new files is `True`. For incremental updates, the writer adapts to the system used in the previous iteration of the document (as mandated by the standard).

object_streams: `List[ObjectStream]`

security_handler: `Optional[SecurityHandler]`

set_custom_trailer_entry(*key: NameObject*, *value: PdfObject*)

Set a custom, unmanaged entry in the document trailer or cross-reference stream dictionary.

Warning: Calling this method to set an entry that is managed by pyHanko internally (info dictionary, document catalog, etc.) has undefined results.

Parameters

- **key** – Dictionary key to use in the trailer.
- **value** – Value to set

`pyhanko.pdf_utils.writer.init_xobject_dictionary`(*command_stream: bytes*, *box_width*, *box_height*,
resources: Optional[DictionaryObject] = None) →
[StreamObject](#)

Helper function to initialise form XObject dictionaries.

Note: For utilities to handle image XObjects, see [images](#).

Parameters

- **command_stream** – The XObject’s raw appearance stream.
- **box_width** – The width of the XObject’s bounding box.
- **box_height** – The height of the XObject’s bounding box.
- **resources** – A resource dictionary to include with the form object.

Returns

A [StreamObject](#) representation of the form XObject.

`pyhanko.pdf_utils.writer.copy_into_new_writer(input_handler: PdfHandler, writer_kwargs: Optional[dict] = None) → PdfFileWriter`

Copy all objects in a given PDF handler into a new [PdfFileWriter](#). This operation will attempt to preserve the document catalog of the original `input_handler`.

Very roughly, calling this function and then immediately invoking `write()` on the resulting writer should result in an equivalent document as far as presentation is concerned. As a general rule, behaviour that is controlled from outside the document catalog (e.g. encryption) or that requires byte-for-byte equivalence with the original (e.g. digital signatures) will not survive this translation.

Parameters

- **input_handler** – [PdfHandler](#) to source objects from.
- **writer_kwargs** – Keyword arguments to pass to the writer.

Returns

New [PdfFileWriter](#) containing all objects from the input handler.

pyhanko.pdf_utils.xref module

Internal utilities to handle the processing of cross-reference data and document trailer data.

This entire module is considered internal API.

class `pyhanko.pdf_utils.xref.XRefCache(reader, xref_sections: List[XRefSection])`

Bases: `object`

Internal class to parse & store information from the xref section(s) of a PDF document.

Stores both the most recent status of all xrefs in addition to their historical values.

All members of this class are considered internal API and are subject to change without notice.

property `total_revisions`

`get_last_change(ref: Reference)`

`object_streams_used_in(revision)`

`get_introducing_revision(ref: Reference)`

`get_xref_container_info(revision) → XRefSectionMetaInfo`

get_xref_data(*revision*) → *XRefSectionData*

explicit_refs_in_revision(*revision*) → Set[*Reference*]

Look up the object refs for all objects explicitly added or overwritten in a given revision.

Parameters

revision – A revision number. The oldest revision is zero.

Returns

A set of Reference objects.

refs_freed_in_revision(*revision*) → Set[*Reference*]

Look up the object refs for all objects explicitly freed in a given revision.

Parameters

revision – A revision number. The oldest revision is zero.

Returns

A set of Reference objects.

get_startxref_for_revision(*revision*) → int

Look up the location of the XRef table/stream associated with a specific revision, as indicated by startxref or /Prev.

Parameters

revision – A revision number. The oldest revision is zero.

Returns

An integer pointer

get_historical_ref(*ref*, *revision*) → Optional[Union[int, *ObjStreamRef*]]

Look up the location of the historical value of an object.

Note: This method is not suitable for determining whether or not a particular object ID is available in a given revision, since it treats unused objects and freed objects the same way.

Parameters

- **ref** – An object reference.
- **revision** – A revision number. The oldest revision is zero.

Returns

An integer offset, an object stream reference, or **None** if the reference does not resolve in the specified revision.

property hybrid_xrefs_present: bool

Determine if a file uses hybrid references anywhere.

Returns

True if hybrid references were detected, False otherwise.

class pyhanko.pdf_utils.xref.**XRefBuilder**(*handler*: PdfHandler, *stream*, *strict*: bool, *last_startxref*: int)

Bases: object

err_limit = 10

read_xrefs()

```
class pyhanko.pdf_utils.xref.XRefType(value)
```

Bases: Enum

Different types of cross-reference entries.

FREE = 1

A freeing instruction.

STANDARD = 2

A regular top-level object.

IN_OBJ_STREAM = 3

An object that is part of an object stream.

```
class pyhanko.pdf_utils.xref.XRefEntry(xref_type: XRefType, location: Optional[Union[int,
ObjStreamRef]], idnum: int, generation: int = 0)
```

Bases: object

Value type representing a single cross-reference entry.

xref_type: [XRefType](#)

The type of cross-reference entry.

location: Optional[Union[int, [ObjStreamRef](#)]]

Location the cross-reference points to.

idnum: int

The ID of the object being referenced.

generation: int = 0

The generation number of the object being referenced.

```
class pyhanko.pdf_utils.xref.ObjStreamRef(obj_stream_id: int, ix_in_stream: int)
```

Bases: object

Identifies an object that's part of an object stream.

obj_stream_id: int

The ID number of the object stream (its generation number is presumed zero).

ix_in_stream: int

The index of the object in the stream.

```
exception pyhanko.pdf_utils.xref.ObjectHeaderReadError(msg: str, *args)
```

Bases: [PdfReadError](#)

```
class pyhanko.pdf_utils.xref.XRefSection(meta_info: XRefSectionMetaInfo, xref_data:
XRefSectionData)
```

Bases: object

Describes a cross-reference section and describes how it is serialised into the PDF file.

meta_info: [XRefSectionMetaInfo](#)

Metadata about the cross-reference section.

xref_data: [XRefSectionData](#)

A description of the actual object pointer definitions.

```
class pyhanko.pdf_utils.xref.XRefSectionData
```

Bases: object

Internal class for bookkeeping on a single cross-reference section, independently of the others.

```
try_resolve(ref: Union[Reference, IndirectObject]) → Optional[Union[int, ObjStreamRef]]
```

```
process_entries(entries: Iterator[XRefEntry], strict: bool)
```

```
process_hybrid_entries(entries: Iterator[XRefEntry], xref_meta_info: XRefSectionMetaInfo, strict: bool)
```

```
higher_generation_refs()
```

```
class pyhanko.pdf_utils.xref.XRefSectionType(value)
```

Bases: Enum

An enumeration.

```
STANDARD = 1
```

```
STREAM = 2
```

```
HYBRID_MAIN = 3
```

```
HYBRID_STREAM = 4
```

```
class pyhanko.pdf_utils.xref.XRefSectionMetaInfo(xref_section_type:
                                                    pyhanko.pdf_utils.xref.XRefSectionType, size: int,
                                                    declared_startxref: int, start_location: int,
                                                    end_location: int, stream_ref:
                                                    Union[pyhanko.pdf_utils.generic.Reference,
                                                         NoneType])
```

Bases: object

```
xref_section_type: XRefSectionType
```

The type of cross-reference section.

```
size: int
```

The highest object ID in scope for this xref section.

```
declared_startxref: int
```

Location pointed to by the startxref pointer in that revision.

```
start_location: int
```

Actual start location of the xref data. This should be equal to *declared_startxref*, but in broken files that may not be the case.

```
end_location: int
```

Location where the xref data ended.

```
stream_ref: Optional[Reference]
```

Reference to the relevant xref stream, if applicable.

```
class pyhanko.pdf_utils.xref.TrailerDictionary
```

Bases: PdfObject

The standard mandates that each trailer shall contain at least all keys used in the preceding trailer, even if unmodified. Of course, we cannot trust documents to actually follow this rule, so this class implements fallbacks.


```
non_trailer_keys = {'/DecodeParms', '/Filter', '/Index', '/Length', '/Type', '/W',
                    '/XRefStm'}
```

```
add_trailer_revision(trailer_dict: DictionaryObject)
```

```
raw_get(key, decrypt: EncryptedObjAccess = EncryptedObjAccess.TRANSPARENT, revision=None)
```

```
flatten(revision=None) → DictionaryObject
```

```
keys()
```

```
items()
```

```
write_to_stream(stream, handler=None, container_ref=None)
```

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

```
pyhanko.pdf_utils.xref.read_object_header(stream, strict)
```

```
pyhanko.pdf_utils.xref.parse_xref_stream(xref_stream: StreamObject, strict: bool = True) →
                                         Iterator[XRefEntry]
```

Parse a single cross-reference stream and yield its entries one by one.

This is internal API.

Parameters

- **xref_stream** – A StreamObject.
- **strict** – Boolean indicating whether we’re running in strict mode.

Returns

A generator object yielding *XRefEntry* objects.

```
pyhanko.pdf_utils.xref.parse_xref_table(stream) → Iterator[XRefEntry]
```

Parse a single cross-reference table and yield its entries one by one.

This is internal API.

Parameters

stream – A file-like object pointed to the start of the cross-reference table.

Returns

A generator object yielding *XRefEntry* objects.

```
pyhanko.pdf_utils.xref.write_xref_table(stream, position_dict: Dict[Tuple[int, int], int])
```

```
class pyhanko.pdf_utils.xref.ObjectStream(compress=True)
```

Bases: object

Utility class to collect objects into a PDF object stream.

Object streams are mainly useful for space efficiency reasons. They allow related objects to be grouped & compressed together in a more flexible manner.

Warning: Object streams can only be used in files with a cross-reference stream, as opposed to a classical XRef table. In particular, this means that incremental updates to files with a legacy XRef table cannot contain object streams either. See § 7.5.7 in ISO 32000-1 for further details.

Danger: Use `BasePdfFileWriter.prepare_object_stream()` to create instances of object streams. The `__init__` function is internal API.

add_object(*idnum*: int, *obj*: PdfObject)

Add an object to an object stream. Note that objects in object streams always have their generation number set to 0 by definition.

Parameters

- **idnum** – The object’s ID number.
- **obj** – The object to embed into the object stream.

Raises

TypeError – Raised if *obj* is an instance of `StreamObject` or `IndirectObject`.

as_pdf_object() → `StreamObject`

Render the object stream to a PDF stream object

Returns

An instance of `StreamObject`.

class pyhanko.pdf_utils.xref.XRefStream(*position_dict*: Dict[Tuple[int, int], Union[int, Tuple[int, int]]])

Bases: `StreamObject`

write_to_stream(*stream*, *handler*=None, *container_ref*=None)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **container_ref** – Local encryption key.
- **handler** – Security handler

pyhanko.sign package

pyhanko.sign.ades package

pyhanko.sign.ades.api module

class pyhanko.sign.ades.api.GenericCommitment(*value*)

Bases: Enum

An enumeration.

PROOF_OF_ORIGIN = 1

PROOF_OF_RECEIPT = 2

PROOF_OF_DELIVERY = 3

PROOF_OF_SENDER = 4

PROOF_OF_APPROVAL = 5

PROOF_OF_CREATION = 6

property asn1: *CommitmentTypeIndication*

```
class pyhanko.sign.adess.api.CAdESSignedAttrSpec(commitment_type:
    Optional[CommitmentTypeIndication] = None,
    timestamp_content: bool = False,
    signature_policy_identifier:
    Optional[SignaturePolicyIdentifier] = None,
    signer_attributes: Optional[SignerAttrSpec] = None)
```

Bases: object

Class that controls signed CAdES attributes on a PDF signature.

commitment_type: Optional[*CommitmentTypeIndication*] = None

Signature commitment type. Can be one of the standard values, or a custom one.

timestamp_content: bool = False

Indicate whether the signature should include a signed timestamp.

Note: This should be contrasted with *unsigned* timestamps: a signed timestamp proves that the signature was created *after* some point in time, while an *unsigned* timestamp computed over the signed content proves that the signature existed *before* said point in time.

signature_policy_identifier: Optional[*SignaturePolicyIdentifier*] = None

Signature policy identifier to embed into the signature.

Warning: Right now, pyHanko does not “understand” signature policies, so the signature policy identifier will be taken at face value and embedded without paying any heed to the actual rules of the signature policy. It is the API user’s responsibility to make sure that all relevant provisions of the signature policy are adhered to.

signer_attributes: Optional[*SignerAttrSpec*] = None

Settings for signer’s attributes, to be included in a signer-attributes-v2 attribute on the signature.

prepare_providers(message_digest, md_algorithm, timestampers: Optional[*TimeStamper*] = None)

```
class pyhanko.sign.adess.api.SignerAttrSpec(claimed_attrs: Iterable[AttCertAttribute], certified_attrs:
    Iterable[AttributeCertificateV2])
```

Bases: object

Class that controls the signer-attributes-v2 signed CAdES attribute.

These represent attributes of the signing entity, not the signature or signed content.

Note: Out of the box, only basic claimed attributes and certified attributes through V2 X.509 attribute certificates are supported.

claimed_attrs: `Iterable[AttCertAttribute]`

Attributes claimed by the signer without further justification.

certified_attrs: `Iterable[AttributeCertificateV2]`

Attribute certificates containing signer attributes.

pyhanko.sign.ades.asn1_util module

`pyhanko.sign.ades.asn1_util.as_set_of(asn1_type: Type)`

`pyhanko.sign.ades.asn1_util.register_cms_attribute(dotted_oid: str, readable_name: str, asn1_type: Type)`

pyhanko.sign.ades.cades_asn1 module

class `pyhanko.sign.ades.cades_asn1.CommitmentTypeIdentifier`(*value=None, default=None, contents=None, **kwargs*)

Bases: `ObjectIdentifier`

class `pyhanko.sign.ades.cades_asn1.CommitmentTypeQualifier`(*value=None, default=None, **kwargs*)

Bases: `Sequence`

class `pyhanko.sign.ades.cades_asn1.CommitmentTypeQualifiers`(*value=None, default=None, contents=None, spec=None, **kwargs*)

Bases: `SequenceOf`

class `pyhanko.sign.ades.cades_asn1.CommitmentTypeIndication`(*value=None, default=None, **kwargs*)

Bases: `Sequence`

class `pyhanko.sign.ades.cades_asn1.SigPolicyQualifierId`(*value=None, default=None, contents=None, **kwargs*)

Bases: `ObjectIdentifier`

class `pyhanko.sign.ades.cades_asn1.NoticeNumbers`(*value=None, default=None, contents=None, spec=None, **kwargs*)

Bases: `SequenceOf`

class `pyhanko.sign.ades.cades_asn1.NoticeReference`(*value=None, default=None, **kwargs*)

Bases: `Sequence`

class `pyhanko.sign.ades.cades_asn1.SPUserNotice`(*value=None, default=None, **kwargs*)

Bases: `Sequence`

class `pyhanko.sign.ades.cades_asn1.SPDocSpecification`(*value=None, default=None, **kwargs*)

Bases: `Sequence`

class `pyhanko.sign.ades.cades_asn1.SigPolicyQualifierInfo`(*value=None, default=None, **kwargs*)

Bases: `Sequence`

class `pyhanko.sign.ades.cades_asn1.SigPolicyQualifierInfos`(*value=None, default=None, contents=None, spec=None, **kwargs*)

Bases: `SequenceOf`

```

class pyhanko.sign.ades.cades_asn1.SignaturePolicyId(value=None, default=None, **kwargs)
    Bases: Sequence

class pyhanko.sign.ades.cades_asn1.SignaturePolicyIdentifier(name=None, value=None,
                                                             **kwargs)
    Bases: Choice

class pyhanko.sign.ades.cades_asn1.SignaturePolicyDocument(value=None, default=None, **kwargs)
    Bases: Sequence

class pyhanko.sign.ades.cades_asn1.SignaturePolicyStore(value=None, default=None, **kwargs)
    Bases: Sequence

class pyhanko.sign.ades.cades_asn1.DisplayText(name=None, value=None, **kwargs)
    Bases: Choice

class pyhanko.sign.ades.cades_asn1.SignerAttributesV2(value=None, default=None, **kwargs)
    Bases: Sequence

class pyhanko.sign.ades.cades_asn1.CertifiedAttributesV2(value=None, default=None,
                                                         contents=None, spec=None, **kwargs)
    Bases: SequenceOf

class pyhanko.sign.ades.cades_asn1.CertifiedAttributeChoices(name=None, value=None,
                                                             **kwargs)
    Bases: Choice

class pyhanko.sign.ades.cades_asn1.OtherAttrCert(value=None, default=None, **kwargs)
    Bases: Sequence

class pyhanko.sign.ades.cades_asn1.OtherAttrCertId(value=None, default=None, contents=None,
                                                    **kwargs)
    Bases: ObjectIdentifier

class pyhanko.sign.ades.cades_asn1.SignedAssertions(value=None, default=None, contents=None,
                                                    spec=None, **kwargs)
    Bases: SequenceOf

class pyhanko.sign.ades.cades_asn1.SignedAssertion(value=None, default=None, **kwargs)
    Bases: Sequence

class pyhanko.sign.ades.cades_asn1.SignedAssertionId(value=None, default=None, contents=None,
                                                      **kwargs)
    Bases: ObjectIdentifier

```

pyhanko.sign.ades.report module

Module for AdES reporting data.

Defines enums for all AdES validation statuses defined in ETSI EN 319 102-1, clause 5.1.3.

```

class pyhanko.sign.ades.report.AdESStatus(value)
    Bases: Enum

    An enumeration.

```

PASSED = 1

INDETERMINATE = 2

FAILED = 3

class pyhanko.sign.ades.report.AdESSubIndic

Bases: object

property status: *AdESStatus*

class pyhanko.sign.ades.report.AdESFailure(value)

Bases: *AdESSubIndic*, Enum

An enumeration.

FORMAT_FAILURE = 1

HASH_FAILURE = 2

SIG_CRYPTO_FAILURE = 3

REVOKED = 4

class pyhanko.sign.ades.report.AdESIndeterminate(value)

Bases: *AdESSubIndic*, Enum

An enumeration.

SIG_CONSTRAINTS_FAILURE = 1

CHAIN_CONSTRAINTS_FAILURE = 2

CERTIFICATE_CHAIN_GENERAL_FAILURE = 3

CRYPTO_CONSTRAINTS_FAILURE = 4

EXPIRED = 5

NOT_YET_VALID = 6

POLICY_PROCESSING_ERROR = 7

SIGNATURE_POLICY_NOT_AVAILABLE = 8

TIMESTAMP_ORDER_FAILURE = 9

NO_SIGNING_CERTIFICATE_FOUND = 10

NO_CERTIFICATE_CHAIN_FOUND = 11

REVOKED_NO_POE = 12

REVOKED_CA_NO_POE = 13

OUT_OF_BOUNDS_NO_POE = 14

CRYPTO_CONSTRAINTS_FAILURE_NO_POE = 15

NO_POE = 16

TRY_LATER = 17

SIGNED_DATA_NOT_FOUND = 18

GENERIC = 19

pyhanko.sign.attributes module

class pyhanko.sign.attributes.SignedAttributeProviderSpec

Bases: ABC

New in version 0.14.0.

Interface for setting up signed attributes, independently of the [Signer](#) hierarchy.

signed_attr_providers(*data_digest*: bytes, *digest_algorithm*: str) → Iterable[[CMSAttributeProvider](#)]

Lazily set up signed attribute providers.

Parameters

- **data_digest** – The digest of the data to be signed.
- **digest_algorithm** – The digest algorithm used.

class pyhanko.sign.attributes.UnsignedAttributeProviderSpec

Bases: ABC

New in version 0.14.0.

Interface for setting up unsigned attributes, independently of the [Signer](#) hierarchy.

unsigned_attr_providers(*signature*: bytes, *signed_attrs*: [CMSAttributes](#), *digest_algorithm*: str) → Iterable[[CMSAttributeProvider](#)]

Lazily set up unsigned attribute providers.

Parameters

- **signature** – The signature computed over the signed attributes.
- **signed_attrs** – Signed attributes over which the signature was taken.
- **digest_algorithm** – The digest algorithm used.

class pyhanko.sign.attributes.CMSAttributeProvider

Bases: object

Base class to provide asynchronous CMS attribute values.

attribute_type: str

Name of the CMS attribute type this provider supplies. See [cms.CMSAttributeType](#).

async build_attr_value(*dry_run=False*)

Build the attribute value asynchronously.

Parameters

- **dry_run** – True if the signer is operating in dry-run (size estimation) mode.

Returns

An attribute value appropriate for the attribute type.

async get_attribute(*dry_run=False*) → Optional[[CMSAttribute](#)]

class pyhanko.sign.attributes.**SigningTimeProvider**(*timestamp: datetime*)

Bases: [*CMSAttributeProvider*](#)

Provide a value for the signing-time attribute (i.e. an otherwise unauthenticated timestamp).

Parameters

timestamp – Datetime object to include.

attribute_type: str = 'signing_time'

Name of the CMS attribute type this provider supplies. See `cms.CMSAttributeType`.

async build_attr_value(*dry_run=False*) → Time

Build the attribute value asynchronously.

Parameters

dry_run – True if the signer is operating in dry-run (size estimation) mode.

Returns

An attribute value appropriate for the attribute type.

class pyhanko.sign.attributes.**SigningCertificateV2Provider**(*signing_cert: Certificate*)

Bases: [*CMSAttributeProvider*](#)

Provide a value for the signing-certificate-v2 attribute.

Parameters

signing_cert – Certificate containing the signer's public key.

attribute_type: str = 'signing_certificate_v2'

Name of the CMS attribute type this provider supplies. See `cms.CMSAttributeType`.

async build_attr_value(*dry_run=False*) → SigningCertificateV2

Build the attribute value asynchronously.

Parameters

dry_run – True if the signer is operating in dry-run (size estimation) mode.

Returns

An attribute value appropriate for the attribute type.

class pyhanko.sign.attributes.**AdobeRevinfoProvider**(*value: RevocationInfoArchival*)

Bases: [*CMSAttributeProvider*](#)

Yield Adobe-style revocation information for inclusion into a CMS object.

Parameters

value – A (pre-formatted) RevocationInfoArchival object.

attribute_type: str = 'adobe_revocation_info_archival'

Name of the CMS attribute type this provider supplies. See `cms.CMSAttributeType`.

async build_attr_value(*dry_run=False*) → Optional[RevocationInfoArchival]

Build the attribute value asynchronously.

Parameters

dry_run – True if the signer is operating in dry-run (size estimation) mode.

Returns

An attribute value appropriate for the attribute type.

```
class pyhanko.sign.attributes.CMSAlgorithmProtectionProvider(digest_algo: str, signature_algo: SignedDigestAlgorithm)
```

Bases: [CMSAttributeProvider](#)

attribute_type: `str` = `'cms_algorithm_protection'`

Name of the CMS attribute type this provider supplies. See `cms.CMSAttributeType`.

async build_attr_value(*dry_run=False*) → `CMSAlgorithmProtection`

Build the attribute value asynchronously.

Parameters

dry_run – True if the signer is operating in dry-run (size estimation) mode.

Returns

An attribute value appropriate for the attribute type.

```
class pyhanko.sign.attributes.TSTProvider(digest_algorithm: str, data_to_ts: bytes, timestamp: TimeStamper, attr_type: str = 'signature_time_stamp_token', prehashed=False)
```

Bases: [CMSAttributeProvider](#)

attribute_type: `str`

Name of the CMS attribute type this provider supplies. See `cms.CMSAttributeType`.

async build_attr_value(*dry_run=False*) → `ContentInfo`

Build the attribute value asynchronously.

Parameters

dry_run – True if the signer is operating in dry-run (size estimation) mode.

Returns

An attribute value appropriate for the attribute type.

pyhanko.sign.beid module

Sign PDF files using a Belgian eID card.

This module defines a very thin convenience wrapper around [pyhanko.sign.pkcs11](#) to set up a PKCS#11 session with an eID card and read the appropriate certificates on the device.

`pyhanko.sign.beid.open_beid_session`(*lib_location, slot_no=None*) → `Session`

Open a PKCS#11 session

Parameters

- **lib_location** – Path to the shared library file containing the eID PKCS#11 module. Usually, the file is named `libbeidpkcs11.so`, `libbeidpkcs11.dylib` or `beidpkcs11.dll`, depending on your operating system.
- **slot_no** – Slot number to use. If not specified, the first slot containing a token labelled BELPIC will be used.

Returns

An open PKCS#11 session object.

```
class pyhanko.sign.beid.BEIDSigner(pkcs11_session: Session, use_auth_cert: bool = False, bulk_fetch: bool = False, embed_roots=True)
```

Bases: [*PKCS11Signer*](#)

Belgian eID-specific signer implementation that automatically populates the (trustless) certificate list with the relevant certificates stored on the card. This includes the government's (self-signed) root certificate and the certificate of the appropriate intermediate CA.

pyhanko.sign.diff_analysis package

Changed in version 0.2.0: Module extracted from [*pyhanko.sign.validation*](#) and restructured into a more rule-based format.

Changed in version 0.11.0: Module refactored into sub-package.

This package defines utilities for difference analysis between revisions of the same PDF file. PyHanko uses this functionality to validate signatures on files that have been modified after signing (using PDF's incremental update feature).

In pyHanko's validation model, every incremental update is disallowed by default. For a change to be accepted, it must be cleared by at least one whitelisting rule. These rules can moreover *qualify* the modification level at which they accept the change (see [*ModificationLevel*](#)). Additionally, any rule can veto an entire revision as suspect by raising a [*SuspiciousModification*](#) exception. Whitelisting rules are encouraged to apply their vetoes liberally.

Whitelisting rules are bundled in [*DiffPolicy*](#) objects for use by the validator.

Guidelines for developing rules for use with [*StandardDiffPolicy*](#)

Caution: These APIs aren't fully stable yet, so some changes might still occur between now and the first major release.

In general, you should keep the following informal guidelines in mind when putting together custom diff rules.

- All rules are either executed completely (i.e. their generators exhausted) or aborted.
- If the diff runner aborts a rule, this always means that the entire revision is rejected. In other words, for accepted revisions, all rules will always have run to completion.
- Whitelisting rules are allowed to informally delegate some checking to other rules, provided that this is documented clearly.

Note: Example: [*CatalogModificationRule*](#) ignores `/AcroForm`, which is validated by another rule entirely.

- Rules should be entirely stateless. "Clearing" a reference by yielding it does not imply that the revision cannot be vetoed by that same rule further down the road (this is why the first point is important).

pyhanko.sign.diff_analysis.common module

Module defining common helpers for use by rules and policies.

In principle, these aren't relevant to the high-level validation API.

```
pyhanko.sign.diff_analysis.common.qualified(level:
    ~pyhanko.sign.diff_analysis.policy_api.ModificationLevel,
    rule_result:
    ~typing.Generator[~pyhanko.sign.diff_analysis.common.X,
    None, ~pyhanko.sign.diff_analysis.common.R], transform:
    ~typing.Callable[[~pyhanko.sign.diff_analysis.common.X],
    ~pyhanko.sign.diff_analysis.rules_api.ReferenceUpdate] =
    <function <lambda>>>) →
    Generator[Tuple[ModificationLevel, ReferenceUpdate],
    None, R]
```

This is a helper function for rule implementors. It attaches a fixed modification level to an existing reference update generator, respecting the original generator's return value (if relevant).

A prototypical use would be of the following form:

```
def some_generator_function():
    # do stuff
    for ref in some_list:
        # do stuff
        yield ref

    # do more stuff
    return summary_value

# ...

def some_qualified_generator_function():
    summary_value = yield from qualified(
        ModificationLevel.FORM_FILLING,
        some_generator_function()
    )
```

Provided that `some_generator_function` yields `ReferenceUpdate` objects, the yield type of the resulting generator will be tuples of the form `(level, ref)`.

Parameters

- **level** – The modification level to set.
- **rule_result** – A generator that outputs references to be whitelisted.
- **transform** – Function to apply to the reference object before appending the modification level and yielding it. Defaults to the identity.

Returns

A converted generator that outputs references qualified at the modification level specified.

```
pyhanko.sign.diff_analysis.common.safe_whitelist(old: HistoricalResolver, old_ref, new_ref) →
    Generator[Reference, None, None]
```

Checks whether an indirect reference in a PDF structure can be updated without clobbering an older object in a way that causes ramifications at the PDF syntax level.

The following are verified:

- Does the old reference point to a non-stream object?
- If the new reference is equal to the old one, does the new reference point to a non-stream object?
- If the new reference is not equal to the old one, is the new reference a newly defined object?

This is a generator for syntactical convenience and integration with internal APIs, but it will always yield at most one element.

```
pyhanko.sign.diff_analysis.common.compare_key_refs(key, old: HistoricalResolver, old_dict:
                                                    DictionaryObject, new_dict: DictionaryObject)
                                                    → Generator[Reference, None,
                                                    Tuple[Optional[PdfObject],
                                                    Optional[PdfObject]]]
```

Ensure that updating a key in a dictionary has no undesirable side effects. The following scenarios are allowed:

1. adding a key in new_dict
2. replacing a direct value in old_dict with a reference in new_dict
3. the reverse (allowed by default)
4. replacing a reference with another reference (that doesn't override anything else)

The restrictions of *safe_whitelist* apply to this function as well.

Note: this routine is only safe to use if the structure of the resulting values is also checked. Otherwise, it can lead to reference leaks if one is not careful.

```
pyhanko.sign.diff_analysis.common.compare_dicts(old_dict: PdfObject, new_dict: PdfObject, ignored:
                                                  Set[str] = frozenset({}), raise_exc=True) → bool
```

Compare entries in two dictionaries, optionally ignoring certain keys.

```
pyhanko.sign.diff_analysis.common.assert_not_stream(obj)
```

Throw *SuspiciousModification* if the argument is a stream object.

pyhanko.sign.diff_analysis.form_rules_api module

Module defining API types for use by form analysis rules.

In principle, these aren't relevant to the high-level validation API.

```
class pyhanko.sign.diff_analysis.form_rules_api.FormUpdatingRule(field_rules:
                                                                  List[FieldMDPRule],
                                                                  ignored_acroform_keys=None)
```

Bases: object

Special whitelisting rule that validates changes to the form attached to the input document.

This rule is special in two ways:

- it outputs *FormUpdate* objects instead of references;
- it delegates most of the hard work to sub-rules (instances of *FieldMDPRule*).

A *DiffPolicy* can have at most one *FormUpdatingRule*, but there is no limit on the number of *FieldMDPRule* objects attached to it.

FormUpdate objects contain a reference plus metadata about the form field it belongs to.

Parameters

- **field_rules** – A list of *FieldMDPRule* objects to validate the individual form fields.
- **ignored_acroform_keys** – Keys in the /AcroForm dictionary that may be changed. Changes are potentially subject to validation by other rules.

apply(*old*: *HistoricalResolver*, *new*: *HistoricalResolver*) → Iterable[Tuple[*ModificationLevel*, *FormUpdate*]]

Evaluate changes in the document's form between two revisions.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision to be vetted.

```
class pyhanko.sign.diff_analysis.form_rules_api.FormUpdate(updated_ref: Reference,
                                                            paths_checked:
                                                                Optional[Union[RawPdfPath,
                                                                Iterable[RawPdfPath]]] = None,
                                                            blanket_approve: bool = False,
                                                            field_name: Optional[str] = None,
                                                            valid_when_locked: bool = False,
                                                            valid_when_certifying: bool = True)
```

Bases: *ReferenceUpdate*

Container for a reference together with (optional) metadata.

Currently, this metadata consists of the relevant field's (fully qualified) name, and whether the update should be approved or not if said field is locked by the FieldMDP policy currently in force.

field_name: Optional[str] = None

The relevant field's fully qualified name, or None if there's either no obvious associated field, or if there are multiple reasonable candidates.

valid_when_locked: bool = False

Flag indicating whether the update is valid even when the field is locked. This is only relevant if *field_name* is not None.

valid_when_certifying: bool = True

Flag indicating whether the update is valid when checking against an explicit DocMDP policy. Default is True. If False, the change will only be accepted if we are evaluating changes to a document after an approval signature.

```
class pyhanko.sign.diff_analysis.form_rules_api.FieldMDPRule
```

Bases: object

Sub-rules attached to a *FormUpdatingRule*.

apply(*context*: *FieldComparisonContext*) → Iterable[Tuple[*ModificationLevel*, *FormUpdate*]]

Apply the rule to the given *FieldComparisonContext*.

Parameters

context – The context of this form revision evaluation, given as an instance of *FieldComparisonContext*.

```
class pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonSpec(field_type: str,
                                                                    old_field_ref:
                                                                        Optional[Reference],
                                                                    new_field_ref:
                                                                        Optional[Reference],
                                                                    old_canonical_path:
                                                                        Optional[RawPdfPath])
```

Bases: object

Helper object that specifies a form field name together with references to its old and new versions.

field_type: str

The (fully qualified) form field name.

old_field_ref: Optional[Reference]

A reference to the field's dictionary in the old revision, if present.

new_field_ref: Optional[Reference]

A reference to the field's dictionary in the new revision, if present.

old_canonical_path: Optional[RawPdfPath]

Path from the trailer through the AcroForm structure to this field (in the older revision). If the field is new, set to None.

property old_field: Optional[DictionaryObject]

Returns

The field's dictionary in the old revision, if present, otherwise None.

property new_field: Optional[DictionaryObject]

Returns

The field's dictionary in the new revision, if present, otherwise None.

expected_paths()

```
class pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonContext(field_specs: Dict[str,  
                                                                    FieldComparisonSpec],  
                                                                    old: HistoricalResolver,  
                                                                    new:  
                                                                    HistoricalResolver)
```

Bases: object

Context for a form diffing operation.

field_specs: Dict[str, FieldComparisonSpec]

Dictionary mapping field names to *FieldComparisonSpec* objects.

old: HistoricalResolver

The older, base revision.

new: HistoricalResolver

The newer revision.

pyhanko.sign.diff_analysis.policies module

Module defining pyHanko's standard difference policy implementation.

```
class pyhanko.sign.diff_analysis.policies.StandardDiffPolicy(global_rules:  
                                                            List[QualifiedWhitelistRule],  
                                                            form_rule:  
                                                            Optional[FormUpdatingRule],  
                                                            reject_object_freeing=True,  
                                                            ignore_orphaned_objects=True)
```

Bases: *DiffPolicy*

Run a list of rules to analyse the differences between two revisions.

Parameters

- **global_rules** – The *QualifiedWhitelistRule* objects encoding the rules to apply.
- **form_rule** – The *FormUpdatingRule* that adjudicates changes to form fields and their values.
- **reject_object_freeing** – Always fail revisions that free objects that existed prior to signing.

Note: PyHanko resolves freed references to the `null` object in PDF, and a freeing instruction in a cross-reference section is always registered as a change that needs to be approved, regardless of the value of this setting.

It is theoretically possible for a rule to permit deleting content, in which case allowing objects to be freed might be reasonable. That said, pyHanko takes the conservative default position to reject all object freeing instructions as suspect.

- **ignore_orphaned_objects** – Some PDF writers create objects that aren’t used anywhere (tsk tsk). Since those don’t affect the “actual” document content, they can usually be ignored. If `True`, newly created orphaned objects will be cleared at level *ModificationLevel.LTA_UPDATES*. Default is `True`.

apply(*old*: *HistoricalResolver*, *new*: *HistoricalResolver*, *field_mdp_spec*: *Optional*[*FieldMDPSpec*] = *None*, *doc_mdp*: *Optional*[*MDPPerm*] = *None*) → *DiffResult*

Execute the policy on a pair of revisions, with the MDP values provided. *SuspiciousModification* exceptions should be propagated.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision.
- **field_mdp_spec** – The field MDP spec that’s currently active.
- **doc_mdp** – The DocMDP spec that’s currently active.

Returns

A *DiffResult* object summarising the policy’s judgment.

review_file(*reader*: *PdfFileReader*, *base_revision*: *Union*[*int*, *HistoricalResolver*], *field_mdp_spec*: *Optional*[*FieldMDPSpec*] = *None*, *doc_mdp*: *Optional*[*MDPPerm*] = *None*) → *Union*[*DiffResult*, *SuspiciousModification*]

Implementation of *DiffPolicy.review_file()* that reviews each intermediate revision between the base revision and the current one individually.

`pyhanko.sign.diff_analysis.policies.DEFAULT_DIFF_POLICY =`
`<pyhanko.sign.diff_analysis.policies.StandardDiffPolicy object>`

Default *DiffPolicy* implementation.

This policy includes the following rules, all with the default settings. The unqualified rules in the list all have their updates qualified at level *LTA_UPDATES*.

- *CatalogModificationRule*,

- *DocInfoRule*,
- *ObjectStreamRule*,
- *XrefStreamRule*,
- *DSSCompareRule*,
- *MetadataUpdateRule*.
- *FormUpdatingRule*, with the following field rules:
 - *SigFieldCreationRule*,
 - *SigFieldModificationRule*,
 - *GenericFieldModificationRule*.

`pyhanko.sign.diff_analysis.policies.NO_CHANGES_DIFF_POLICY =`
<`pyhanko.sign.diff_analysis.policies.StandardDiffPolicy` object>

DiffPolicy implementation that does not provide any rules, and will therefore simply reject all changes.

`pyhanko.sign.diff_analysis.policy_api` module

`class pyhanko.sign.diff_analysis.policy_api.ModificationLevel(value)`

Bases: *OrderedEnum*

Records the (semantic) modification level of a document.

Compare *MDPPerm*, which records the document modification policy associated with a particular signature, as opposed to the empirical judgment indicated by this enum.

NONE = 0

The document was not modified at all (i.e. it is byte-for-byte unchanged).

LTA_UPDATES = 1

The only updates are of the type that would be allowed as part of signature long term archival (LTA) processing. That is to say, updates to the document security store or new document time stamps. For the purposes of evaluating whether a document has been modified in the sense defined in the PAdES and ISO 32000-2 standards, these updates do not count. Adding form fields is permissible at this level, but only if they are signature fields. This is necessary for proper document timestamp support.

FORM_FILLING = 2

The only updates are extra signatures and updates to form field values or their appearance streams, in addition to the previous levels.

ANNOTATIONS = 3

In addition to the previous levels, manipulating annotations is also allowed at this level.

Note: This level is currently unused by the default diff policy, and modifications to annotations other than those permitted to fill in forms are treated as suspicious.

OTHER = 4

The document has been modified in ways that aren't on the validator's whitelist. This always invalidates the corresponding signature, irrespective of cryptographical integrity or /DocMDP settings.

exception `pyhanko.sign.diff_analysis.policy_api.SuspiciousModification`

Bases: `ValueError`

Error indicating a suspicious modification

class `pyhanko.sign.diff_analysis.policy_api.DiffResult`(*modification_level*: `ModificationLevel`,
changed_form_fields: `Set[str]`)

Bases: `object`

Encodes the result of a difference analysis on two revisions.

Returned by `DiffPolicy.apply()`.

modification_level: `ModificationLevel`

The strictest modification level at which all changes pass muster.

changed_form_fields: `Set[str]`

Set containing the names of all changed form fields.

Note: For the purposes of this parameter, a change is defined as any `FormUpdate` where `FormUpdate.valid_when_locked` is `False`.

class `pyhanko.sign.diff_analysis.policy_api.DiffPolicy`

Bases: `object`

Analyse the differences between two revisions.

apply(*old*: `HistoricalResolver`, *new*: `HistoricalResolver`, *field_mdp_spec*: `Optional[FieldMDPSpec] = None`,
doc_mdp: `Optional[MDPPerm] = None`) → `DiffResult`

Execute the policy on a pair of revisions, with the MDP values provided. `SuspiciousModification` exceptions should be propagated.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision.
- **field_mdp_spec** – The field MDP spec that's currently active.
- **doc_mdp** – The DocMDP spec that's currently active.

Returns

A `DiffResult` object summarising the policy's judgment.

review_file(*reader*: `PdfFileReader`, *base_revision*: `Union[int, HistoricalResolver]`, *field_mdp_spec*:
`Optional[FieldMDPSpec] = None`, *doc_mdp*: `Optional[MDPPerm] = None`) →
`Union[DiffResult, SuspiciousModification]`

Compare the current state of a file to an earlier version, with the MDP values provided. `SuspiciousModification` exceptions should be propagated.

If there are multiple revisions between the base revision and the current one, the precise manner in which the review is conducted is left up to the implementing class. In particular, subclasses may choose to review each intermediate revision individually, or handle them all at once.

Parameters

- **reader** – PDF reader representing the current state of the file.
- **base_revision** – The older, base revision. You can choose between providing it as a revision index, or a `HistoricalResolver` instance.

- **field_mdp_spec** – The field MDP spec that’s currently active.
- **doc_mdp** – The DocMDP spec that’s currently active.

Returns

A *DiffResult* object summarising the policy’s judgment.

pyhanko.sign.diff_analysis.rules.file_structure_rules module

class pyhanko.sign.diff_analysis.rules.file_structure_rules.**CatalogModificationRule**(*ignored_keys=None*)

Bases: *QualifiedWhitelistRule*

Rule that adjudicates modifications to the document catalog.

Parameters

ignored_keys – Values in the document catalog that may change between revisions. The default ones are */AcroForm*, */DSS*, */Extensions*, */Metadata*, */MarkInfo* and */Version*.

Checking for */AcroForm*, */DSS* and */Metadata* is delegated to *FormUpdatingRule*, *DSSCompareRule* and *MetadataUpdateRule*, respectively.

apply_qualified(*old: HistoricalResolver*, *new: HistoricalResolver*) → Iterable[Tuple[*ModificationLevel*, *Reference*]]

Apply the rule to the changes between two revisions.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision to be vetted.

class pyhanko.sign.diff_analysis.rules.file_structure_rules.**ObjectStreamRule**

Bases: *WhitelistRule*

Rule that allows object streams to be added.

Note that this rule only whitelists the object streams themselves (provided they do not override any existing objects, obviously), not the objects in them.

apply(*old: HistoricalResolver*, *new: HistoricalResolver*) → Iterable[*Reference*]

Apply the rule to the changes between two revisions.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision to be vetted.

class pyhanko.sign.diff_analysis.rules.file_structure_rules.**XrefStreamRule**

Bases: *WhitelistRule*

Rule that allows new cross-reference streams to be defined.

apply(*old: HistoricalResolver*, *new: HistoricalResolver*) → Iterable[*Reference*]

Apply the rule to the changes between two revisions.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision to be vetted.

pyhanko.sign.diff_analysis.rules.form_field_rules module

```
class pyhanko.sign.diff_analysis.rules.form_field_rules.DSSCompareRule
```

Bases: [WhitelistRule](#)

Rule that allows changes to the document security store (DSS).

This rule will validate the structure of the DSS quite rigidly, and will raise [SuspiciousModification](#) whenever it encounters structural problems with the DSS. Similarly, modifications that remove structural items from the DSS also count as suspicious. However, merely removing individual OCSP responses, CRLs or certificates when they become irrelevant is permitted. This is also allowed by PAdES.

```
apply(old: HistoricalResolver, new: HistoricalResolver) → Iterable[ReferenceUpdate]
```

Apply the rule to the changes between two revisions.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision to be vetted.

```
class pyhanko.sign.diff_analysis.rules.form_field_rules.SigFieldCreationRule(approve_widget_bindings=True,
al-
low_new_visible_after_certify=False)
```

Bases: [FieldMDPRule](#)

This rule allows signature fields to be created at the root of the form hierarchy, but disallows the creation of other types of fields. It also disallows field deletion.

In addition, this rule will allow newly created signature fields to attach themselves as widget annotations to pages.

The creation of invisible signature fields is considered a modification at level [ModificationLevel.LTA_UPDATES](#), but appearance-related changes will be qualified with [ModificationLevel.FORM_FILLING](#).

Parameters

- **allow_new_visible_after_certify** – Creating new visible signature fields is disallowed after certification signatures by default; this is stricter than Acrobat. Set this parameter to True to disable this check.
- **approve_widget_bindings** – Set to False to reject new widget annotation registrations associated with approved new fields.

```
apply(context: FieldComparisonContext) → Iterable[Tuple[ModificationLevel, FormUpdate]]
```

Apply the rule to the given [FieldComparisonContext](#).

Parameters

context – The context of this form revision evaluation, given as an instance of [FieldComparisonContext](#).

```
class pyhanko.sign.diff_analysis.rules.form_field_rules.SigFieldModificationRule(always_modifiable=None,
value_update_keys=None)
```

Bases: [BaseFieldModificationRule](#)

This rule allows signature fields to be filled in, and set an appearance if desired. Deleting values from signature fields is disallowed, as is modifying signature fields that already contain a signature.

This rule will take field locks into account if the [FieldComparisonContext](#) includes a [FieldMDPSpec](#).

For (invisible) document timestamps, this is allowed at [ModificationLevel.LTA_UPDATES](#), but in all other cases the modification level will be bumped to [ModificationLevel.FORM_FILLING](#).

check_form_field(*fq_name*: str, *spec*: FieldComparisonSpec, *context*: FieldComparisonContext) → Iterable[Tuple[ModificationLevel, FormUpdate]]

Investigate updates to a particular form field. This function is called by `apply()` for every form field in the new revision.

Parameters

- **fq_name** – The fully qualified name of the form field.
- **spec** – The *FieldComparisonSpec* object describing the old state of the field in relation to the new state.
- **context** – The full *FieldComparisonContext* that is currently being evaluated.

Returns

An iterable yielding *FormUpdate* objects qualified with an appropriate *ModificationLevel*.

class pyhanko.sign.diff_analysis.rules.form_field_rules.GenericFieldModificationRule(*always_modifiable*=None, *value_update_keys*=None)

Bases: *BaseFieldModificationRule*

This rule allows non-signature form fields to be modified at `ModificationLevel.FORM_FILLING`.

This rule will take field locks into account if the *FieldComparisonContext* includes a *FieldMDPSpec*.

check_form_field(*fq_name*: str, *spec*: FieldComparisonSpec, *context*: FieldComparisonContext) → Iterable[Tuple[ModificationLevel, FormUpdate]]

Investigate updates to a particular form field. This function is called by `apply()` for every form field in the new revision.

Parameters

- **fq_name** – The fully qualified name of the form field.
- **spec** – The *FieldComparisonSpec* object describing the old state of the field in relation to the new state.
- **context** – The full *FieldComparisonContext* that is currently being evaluated.

Returns

An iterable yielding *FormUpdate* objects qualified with an appropriate *ModificationLevel*.

class pyhanko.sign.diff_analysis.rules.form_field_rules.BaseFieldModificationRule(*always_modifiable*=None, *value_update_keys*=None)

Bases: *FieldMDPRule*

Base class that implements some boilerplate to validate modifications to individual form fields.

compare_fields(*spec*: FieldComparisonSpec) → bool

Helper method to compare field dictionaries.

Parameters

spec – The current *FieldComparisonSpec*.

Returns

True if the modifications are permissible even when the field is locked, False otherwise. If keys beyond those in *value_update_keys* are changed, a *SuspiciousModification* is raised.

apply(context: *FieldComparisonContext*) → Iterable[Tuple[*ModificationLevel*, *FormUpdate*]]

Apply the rule to the given *FieldComparisonContext*.

Parameters

context – The context of this form revision evaluation, given as an instance of *FieldComparisonContext*.

check_form_field(fq_name: str, spec: *FieldComparisonSpec*, context: *FieldComparisonContext*) → Iterable[Tuple[*ModificationLevel*, *FormUpdate*]]

Investigate updates to a particular form field. This function is called by *apply()* for every form field in the new revision.

Parameters

- **fq_name** – The fully qualified name of the form field.
- **spec** – The *FieldComparisonSpec* object describing the old state of the field in relation to the new state.
- **context** – The full *FieldComparisonContext* that is currently being evaluated.

Returns

An iterable yielding *FormUpdate* objects qualified with an appropriate *ModificationLevel*.

pyhanko.sign.diff_analysis.rules.metadata_rules module

class pyhanko.sign.diff_analysis.rules.metadata_rules.**DocInfoRule**

Bases: *WhitelistRule*

Rule that allows the /Info dictionary in the trailer to be updated.

apply(old: *HistoricalResolver*, new: *HistoricalResolver*) → Iterable[*ReferenceUpdate*]

Apply the rule to the changes between two revisions.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision to be vetted.

class pyhanko.sign.diff_analysis.rules.metadata_rules.**MetadataUpdateRule**(check_xml_syntax=True, always_refuse_stream_override=False)

Bases: *WhitelistRule*

Rule to adjudicate updates to the XMP metadata stream.

The content of the metadata isn't actually validated in any significant way; this class only checks whether the XML is well-formed.

Parameters

- **check_xml_syntax** – Do a well-formedness check on the XML syntax. Default True.
- **always_refuse_stream_override** – Always refuse to override the metadata stream if its object ID existed in a prior revision, including if the new stream overrides the old metadata stream and the syntax check passes. Default False.

Note: In other situations, pyHanko will reject stream overrides on general principle, since combined with the fault-tolerance of some PDF readers, these can allow an attacker to manipulate parts of the signed content in subtle but significant ways.

In case of the metadata stream, the risk is significantly mitigated thanks to the XML syntax check on both versions of the stream, but if you're feeling extra paranoid, you can turn the default behaviour back on by setting `always_refuse_stream_override` to `True`.

static `is_well_formed_xml(metadata_ref: Reference)`

Checks whether the provided stream consists of well-formed XML data. Note that this does not perform any more advanced XML or XMP validation, the check is purely syntactic.

Parameters

metadata_ref – A reference to a (purported) metadata stream.

Raises

[*SuspiciousModification*](#) – if there are indications that the reference doesn't point to an XML stream.

apply(*old*: [HistoricalResolver](#), *new*: [HistoricalResolver](#)) → [Iterable](#)[[ReferenceUpdate](#)]

Apply the rule to the changes between two revisions.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision to be vetted.

pyhanko.sign.diff_analysis.rules_api module

Module defining common API types for use by rules and policies.

In principle, these aren't relevant to the high-level validation API.

class `pyhanko.sign.diff_analysis.rules_api.QualifiedWhitelistRule`

Bases: `object`

Abstract base class for a whitelisting rule that outputs references together with the modification level at which they're cleared.

This is intended for use by complicated whitelisting rules that need to differentiate between multiple levels.

apply_qualified(*old*: [HistoricalResolver](#), *new*: [HistoricalResolver](#)) → [Iterable](#)[[Tuple](#)[[ModificationLevel](#), [ReferenceUpdate](#)]]

Apply the rule to the changes between two revisions.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision to be vetted.

class `pyhanko.sign.diff_analysis.rules_api.WhitelistRule`

Bases: `object`

Abstract base class for a whitelisting rule that simply outputs cleared references without specifying a modification level.

These rules are more flexible than rules of type *QualifiedWhitelistRule*, since the modification level can be specified separately (see *WhitelistRule.as_qualified()*).

apply(*old*: *HistoricalResolver*, *new*: *HistoricalResolver*) → *Iterable[ReferenceUpdate]*

Apply the rule to the changes between two revisions.

Parameters

- **old** – The older, base revision.
- **new** – The newer revision to be vetted.

as_qualified(*level*: *ModificationLevel*) → *QualifiedWhitelistRule*

Construct a new *QualifiedWhitelistRule* that whitelists the object references from this rule at the level specified.

Parameters

- level** – The modification level at which the output of this rule should be cleared.

Returns

A *QualifiedWhitelistRule* backed by this rule.

```
class pyhanko.sign.diff_analysis.rules_api.ReferenceUpdate(updated_ref:
    pyhanko.pdf_utils.generic.Reference,
    paths_checked:
        Union[pyhanko.pdf_utils.reader.RawPdfPath,
        Iterable[pyhanko.pdf_utils.reader.RawPdfPath],
        NoneType] = None, blanket_approve:
        bool = False)
```

Bases: *object*

updated_ref: *Reference*

Reference that was (potentially) updated.

paths_checked: *Optional[Union[RawPdfPath, Iterable[RawPdfPath]]]* = *None*

blanket_approve: *bool* = *False*

classmethod *curry_ref*(***kwargs*)

pyhanko.sign.fields module

Utilities to deal with signature form fields and their properties in PDF files.

```
class pyhanko.sign.fields.SigFieldSpec(sig_field_name: str, on_page: int = 0, box: (<class 'int'>, <class 'int'>, <class 'int'>, <class 'int'>) = None, seed_value_dict:
    ~pyhanko.sign.fields.SigSeedValueSpec = None, field_mdp_spec:
    ~pyhanko.sign.fields.FieldMDPSpec = None,
    doc_mdp_update_value: ~pyhanko.sign.fields.MDPPerm = None,
    combine_annotation: bool = True, empty_field_appearance: bool
    = False, invis_sig_settings: ~pyhanko.sign.fields.InvisSigSettings
    = InvisSigSettings(set_print_flag=True, set_hidden_flag=False,
    box_out_of_bounds=False), readable_field_name:
    ~typing.Optional[str] = None, visible_sig_settings:
    ~pyhanko.sign.fields.VisibleSigSettings =
    VisibleSigSettings(rotate_with_page=True,
    scale_with_page_zoom=True, print_signature=True))
```

Bases: `object`

Description of a signature field to be created.

sig_field_name: `str`

Name of the signature field.

on_page: `int = 0`

Index of the page on which the signature field should be included (starting at 0). A negative number counts pages from the back of the document, with index -1 referring to the last page.

Note: This is essentially only relevant for visible signature fields, i.e. those that have a widget associated with them.

box: (`<class 'int'>`, `<class 'int'>`, `<class 'int'>`, `<class 'int'>`) = `None`

Bounding box of the signature field, if applicable.

Typically specified in `ll_x, ll_y, ur_x, ur_y` format, where `ll_*` refers to the lower left and `ur_*` to the upper right corner.

seed_value_dict: `SigSeedValueSpec = None`

Specification for the seed value dictionary, if applicable.

field_mdp_spec: `FieldMDPSpec = None`

Specification for the field lock dictionary, if applicable.

doc_mdp_update_value: `MDPPerm = None`

Value to use for the document modification policy associated with the signature in this field.

This value will be embedded into the field lock dictionary if specified, and is meaningless if `field_mdp_spec` is not specified.

Warning: DocMDP entries for approval signatures are a PDF 2.0 feature. Older PDF software will likely ignore this part of the field lock dictionary.

combine_annotation: `bool = True`

Flag controlling whether the field should be combined with its annotation dictionary; `True` by default.

empty_field_appearance: `bool = False`

Generate a neutral appearance stream for empty, visible signature fields. If `False`, an empty appearance stream will be put in.

Note: We use an empty appearance stream to satisfy the appearance requirements for widget annotations in ISO 32000-2. However, even when a nontrivial appearance stream is present on an empty signature field, many viewers will not use it to render the appearance of the empty field on-screen.

Instead, these viewers typically substitute their own native widget.

invis_sig_settings: `InvisSigSettings = InvisSigSettings(set_print_flag=True, set_hidden_flag=False, box_out_of_bounds=False)`

Advanced settings to control invisible signature field generation.

readable_field_name: `Optional[str] = None`

Human-readable field name (/TU entry).

Note: This value is commonly rendered as a tooltip in viewers, but also serves an accessibility purpose.

visible_sig_settings: `VisibleSigSettings = VisibleSigSettings(rotate_with_page=True, scale_with_page_zoom=True, print_signature=True)`

Advanced settings to control the generation of visible signature fields.

format_lock_dictionary() → `Optional[DictionaryObject]`

class `pyhanko.sign.fields.SigSeedValFlags(value)`

Bases: `Flag`

Flags for the /Ff entry in the seed value dictionary for a signature field. These mark which of the constraints are to be strictly enforced, as opposed to optional ones.

Warning: The flags `LEGAL_ATTESTATION` and `APPEARANCE_FILTER` are processed in accordance with the specification when creating a signature, but support is nevertheless limited.

- PyHanko does not support legal attestations at all, so given that the `LEGAL_ATTESTATION` requirement flag only restricts the legal attestations that can be used by the signer, pyHanko can safely ignore it when signing.

On the other hand, since the validator is not aware of legal attestations either, it cannot validate signatures that make `legal_attestations` a mandatory constraint.

- Since pyHanko does not define any named appearances, setting the `APPEARANCE_FILTER` flag and the `appearance` entry in the seed value dictionary will make pyHanko refuse to sign the document.

When validating, the situation is different: since pyHanko has no way of knowing whether the signer used the named appearance imposed by the seed value dictionary, it will simply emit a warning and continue validating the signature.

FILTER = 1

Makes the signature handler setting mandatory. PyHanko only supports /Adobe.PPKLite.

SUBFILTER = 2

See `subfilters`.

V = 4

See `sv_dict_version`.

REASONS = 8

See `reasons`.

LEGAL_ATTESTATION = 16

See `legal_attestations`.

ADD_REV_INFO = 32

See `add_rev_info`.

DIGEST_METHOD = 64

See `digest_method`.

LOCK_DOCUMENT = 128

See [lock_document](#).

APPEARANCE_FILTER = 256

See [appearance](#).

```
class pyhanko.sign.fields.SigCertConstraints(flags: ~pyhanko.sign.fields.SigCertConstraintFlags =
                                             SigCertConstraintFlags.None, subjects: ~typing.Optional[~typing.List[~asn1crypto.x509.Certificate]]
                                             = None, subject_dn:
                                             ~typing.Optional[~asn1crypto.x509.Name] = None,
                                             issuers: ~typing.Optional[~typing.List[~asn1crypto.x509.Certificate]]
                                             = None, info_url: ~typing.Optional[str] = None, url_type:
                                             ~pyhanko.pdf_utils.generic.NameObject = '/Browser',
                                             key_usage: ~typing.Optional[~typing.List[~pyhanko.sign.fields.SigCertKeyUsage]]
                                             = None)
```

Bases: `object`

This part of the seed value dictionary allows the document author to set constraints on the signer's certificate.

See Table 235 in ISO 32000-1.

flags: [SigCertConstraintFlags](#) = 0

Enforcement flags. By default, all entries are optional.

subjects: `List[Certificate]` = None

Explicit list of certificates that can be used to sign a signature field.

subject_dn: `Name` = None

Certificate subject names that can be used to sign a signature field. Subject DN entries that are not mentioned are unconstrained.

issuers: `List[Certificate]` = None

List of issuer certificates that the signer certificate can be issued by. Note that these issuers do not need to be the *direct* issuer of the signer's certificate; any descendant relationship will do.

info_url: `str` = None

Informational URL that should be opened when an appropriate certificate cannot be found (if [url_type](#) is `/Browser`, that is).

Note: PyHanko ignores this value, but we include it for compatibility.

url_type: [NameObject](#) = `'/Browser'`

Handler that should be used to open [info_url](#). `/Browser` is the only implementation-independent value.

key_usage: `List[SigCertKeyUsage]` = None

Specify the key usage extensions that should (or should not) be present on the signer's certificate.

classmethod `from_pdf_object(pdf_dict)`

Read a PDF dictionary into a [SigCertConstraints](#) object.

Parameters

`pdf_dict` – A [DictionaryObject](#).

Returns

A *SigCertConstraints* object.

as_pdf_object()

Render this *SigCertConstraints* object to a PDF dictionary.

Returns

A *DictionaryObject*.

satisfied_by(*signer: Certificate, validation_path: Optional[ValidationPath]*)

Evaluate whether a signing certificate satisfies the required constraints of this *SigCertConstraints* object.

Parameters

- **signer** – The candidate signer’s certificate.
- **validation_path** – Validation path of the signer’s certificate.

Raises

UnacceptableSignerError – Raised if the conditions are not met.

```
class pyhanko.sign.fields.SigSeedValueSpec(flags: ~pyhanko.sign.fields.SigSeedValFlags =
    SigSeedValFlags.None, reasons:
    ~typing.Optional[~typing.List[str]] = None,
    timestamp_server_url: ~typing.Optional[str] = None,
    timestamp_required: bool = False, cert:
    ~typing.Optional[~pyhanko.sign.fields.SigCertConstraints]
    = None, subfilters: ~typ-
    ing.Optional[~typing.List[~pyhanko.sign.fields.SigSeedSubFilter]]
    = None, digest_methods:
    ~typing.Optional[~typing.List[str]] = None, add_rev_info:
    ~typing.Optional[bool] = None, seed_signature_type:
    ~typing.Optional[~pyhanko.sign.fields.SeedSignatureType]
    = None, sv_dict_version: ~typ-
    ing.Optional[~typing.Union[~pyhanko.sign.fields.SeedValueDictVersion,
    int]] = None, legal_attestations:
    ~typing.Optional[~typing.List[str]] = None, lock_document:
    ~typing.Optional[~pyhanko.sign.fields.SeedLockDocument]
    = None, appearance: ~typing.Optional[str] = None)
```

Bases: object

Python representation of a PDF seed value dictionary.

flags: *SigSeedValFlags* = 0

Enforcement flags. By default, all entries are optional.

reasons: List[str] = None

Acceptable reasons for signing.

timestamp_server_url: str = None

RFC 3161 timestamp server endpoint suggestion.

timestamp_required: bool = False

Flags whether a timestamp is required. This flag is only meaningful if *timestamp_server_url* is specified.

cert: *SigCertConstraints* = None

Constraints on the signer’s certificate.

subfilters: `List[SigSeedSubFilter] = None`

Acceptable /SubFilter values.

digest_methods: `List[str] = None`

Acceptable digest methods.

add_rev_info: `bool = None`

Indicates whether revocation information should be embedded.

Warning: This flag exclusively refers to the Adobe-style revocation information embedded within the CMS object that is written to the signature field. PAdES-style revocation information that is saved to the document security store (DSS) does *not* satisfy the requirement. Additionally, the standard mandates that /SubFilter be equal to /adbe.pkcs7.detached if this flag is True.

seed_signature_type: `SeedSignatureType = None`

Specifies the type of signature that should occupy a signature field; this represents the /MDP entry in the seed value dictionary. See `SeedSignatureType` for details.

Caution: Since a certification-type signature is by definition the first signature applied to a document, compliance with this requirement cannot be cryptographically enforced.

sv_dict_version: `Union[SeedValueDictVersion, int] = None`

Specifies the compliance level required of a seed value dictionary processor. If `None`, pyHanko will compute an appropriate value.

Note: You may also specify this value directly as an integer. This covers potential future versions of the standard that pyHanko does not support out of the box.

legal_attestations: `List[str] = None`

Specifies the possible legal attestations that a certification signature occupying this signature field can supply. The corresponding flag in *flags* indicates whether this is a mandatory constraint.

Caution: Since *legal_attestations* is only relevant for certification signatures, compliance with this requirement cannot be reliably enforced. Regardless, since pyHanko's validator is also unaware of legal attestation settings, it will refuse to validate signatures where this seed value constitutes a mandatory constraint.

Additionally, since pyHanko does not support legal attestation specifications at all, it vacuously satisfies the requirements of this entry no matter what, and will therefore ignore it when signing.

lock_document: `SeedLockDocument = None`

Tell the signer whether or not the document should be locked after signing this field; see `SeedLockDocument` for details.

The corresponding flag in *flags* indicates whether this constraint is mandatory.

appearance: `str = None`

Specify a named appearance to use when generating the signature. The corresponding flag in *flags* indicates whether this constraint is mandatory.

Caution: There is no standard registry of named appearances, so these constraints are not portable, and cannot be validated.

PyHanko currently does not define any named appearances.

as_pdf_object()

Render this *SigSeedValueSpec* object to a PDF dictionary.

Returns

A *DictionaryObject*.

classmethod from_pdf_object(pdf_dict)

Read from a seed value dictionary.

Parameters

pdf_dict – A *DictionaryObject*.

Returns

A *SigSeedValueSpec* object.

build_timestamper()

Return a timestamper object based on the *timestamp_server_url* attribute of this *SigSeedValueSpec* object.

Returns

A *HTTPTimeStamper*.

class pyhanko.sign.fields.SigCertConstraintFlags(value)

Bases: *Flag*

Flags for the /Ff entry in the certificate seed value dictionary for a dictionary field. These mark which of the constraints are to be strictly enforced, as opposed to optional ones.

Warning: While this enum records values for all flags, not all corresponding constraint types have been implemented yet.

SUBJECT = 1

See *SigCertConstraints.subjects*.

ISSUER = 2

See *SigCertConstraints.issuers*.

OID = 4

Currently not supported.

SUBJECT_DN = 8

See *SigCertConstraints.subject_dn*.

RESERVED = 16

Currently not supported (reserved).

KEY_USAGE = 32

See *SigCertConstraints.key_usage*.

URL = 64

See [*SigCertConstraints.info_url*](#).

Note: As specified in the standard, this enforcement bit is supposed to be ignored by default. We include it for compatibility reasons.

UNSUPPORTED = 20

Flags for which the corresponding constraint is unsupported.

class pyhanko.sign.fields.**SigSeedSubFilter**(*value*)

Bases: Enum

Enum declaring all supported /SubFilter values.

ADOBE_PKCS7_DETACHED = '/adbe.pkcs7.detached'

PADES = '/ETSI.CAdES.detached'

ETSI_RFC3161 = '/ETSI.RFC3161'

class pyhanko.sign.fields.**SeedValueDictVersion**(*value*)

Bases: [*OrderedEnum*](#)

Specify the minimal compliance level for a seed value dictionary processor.

PDF_1_5 = 1

Require the reader to understand all keys defined in PDF 1.5.

PDF_1_7 = 2

Require the reader to understand all keys defined in PDF 1.7.

PDF_2_0 = 3

Require the reader to understand all keys defined in PDF 2.0.

class pyhanko.sign.fields.**SeedLockDocument**(*value*)

Bases: Enum

Provides a recommendation to the signer as to whether the document should be locked after signing. The corresponding flag in [*SigSeedValueSpec.flags*](#) determines whether this constraint is a required constraint.

LOCK = '/true'

Lock the document after signing.

DO_NOT_LOCK = '/false'

Lock the document after signing.

SIGNER_DISCRETION = '/auto'

Leave the decision up to the signer.

Note: This is functionally equivalent to not specifying any value.

class pyhanko.sign.fields.**SigCertKeyUsage**(*must_have: Optional[KeyUsage] = None, forbidden: Optional[KeyUsage] = None*)

Bases: object

Encodes the key usage bits that must (resp. must not) be active on the signer's certificate.

Note: See § 4.2.1.3 in [RFC 5280](#) and `KeyUsage` for more information on key usage extensions.

Note: The human-readable names of the key usage extensions are recorded in `camelCase` in [RFC 5280](#), but this class uses the naming convention of `KeyUsage` in `asn1crypto`. The conversion is done by replacing `camelCase` with `snake_case`. For example, `nonRepudiation` becomes `non_repudiation`, and `digitalSignature` turns into `digital_signature`.

Note: This class is intended to closely replicate the definition of the `KeyUsage` entry Table 235 in ISO 32000-1. In particular, it does *not* provide a mechanism to deal with extended key usage extensions (cf. § 4.2.1.12 in [RFC 5280](#)).

Parameters

- **must_have** – The `KeyUsage` object encoding the key usage extensions that must be present on the signer’s certificate.
- **forbidden** – The `KeyUsage` object encoding the key usage extensions that must *not* be present on the signer’s certificate.

`encode_to_sv_string()`

Encode the key usage requirements in the format specified in the PDF specification.

Returns

A string.

`classmethod read_from_sv_string(ku_str)`

Parse a PDF `KeyUsage` string into an instance of [SigCertKeyUsage](#). See Table 235 in ISO 32000-1.

Parameters

ku_str – A PDF `KeyUsage` string.

Returns

An instance of [SigCertKeyUsage](#).

`classmethod from_sets(must_have: Optional[Set[str]] = None, forbidden: Optional[Set[str]] = None)`

Initialise a [SigCertKeyUsage](#) object from two sets.

Parameters

- **must_have** – The key usage extensions that must be present on the signer’s certificate.
- **forbidden** – The key usage extensions that must *not* be present on the signer’s certificate.

Returns

A [SigCertKeyUsage](#) object encoding these.

`must_have_set() → Set[str]`

Return the set of key usage extensions that must be present on the signer’s certificate.

`forbidden_set() → Set[str]`

Return the set of key usage extensions that must not be present on the signer’s certificate.

class pyhanko.sign.fields.**MDPPerm**(*value*)

Bases: *OrderedEnum*

Indicates a /DocMDP level.

Cf. Table 254 in ISO 32000-1.

NO_CHANGES = 1

No changes to the document are allowed.

Warning: This does not apply to DSS updates and the addition of document time stamps.

FILL_FORMS = 2

Form filling & signing is allowed.

ANNOTATE = 3

Form filling, signing and commenting are allowed.

Warning: Validating this /DocMDP level is not currently supported, but included in the list for completeness.

class pyhanko.sign.fields.**FieldMDPAction**(*value*)

Bases: Enum

Marker for the scope of a /FieldMDP policy.

ALL = '/All'

The policy locks all form fields.

INCLUDE = '/Include'

The policy locks all fields in the list (see *FieldMDPSpec.fields*).

EXCLUDE = '/Exclude'

The policy locks all fields except those specified in the list (see *FieldMDPSpec.fields*).

class pyhanko.sign.fields.**FieldMDPSpec**(*action*: *FieldMDPAction*, *fields*: *Optional[List[str]]* = None)

Bases: object

/FieldMDP policy description.

This class models both field lock dictionaries and /FieldMDP transformation parameters.

action: *FieldMDPAction*

Indicates the scope of the policy.

fields: *Optional[List[str]]* = None

Indicates the fields subject to the policy, unless *action* is *FieldMDPAction.ALL*.

as_pdf_object() → *DictionaryObject*

Render this /FieldMDP policy description as a PDF dictionary.

Returns

A *DictionaryObject*.

as_transform_params() → *DictionaryObject*

Render this `/FieldMDP` policy description as a PDF dictionary, ready for inclusion into the `/TransformParams` entry of a `/FieldMDP` dictionary associated with a signature object.

Returns

A *DictionaryObject*.

as_sig_field_lock() → *DictionaryObject*

Render this `/FieldMDP` policy description as a PDF dictionary, ready for inclusion into the `/Lock` dictionary of a signature field.

Returns

A *DictionaryObject*.

classmethod from_pdf_object(pdf_dict) → *FieldMDPSpec*

Read a PDF dictionary into a *FieldMDPSpec* object.

Parameters

pdf_dict – A *DictionaryObject*.

Returns

A *FieldMDPSpec* object.

is_locked(field_name: str) → bool

Adjudicate whether a field should be locked by the policy described by this *FieldMDPSpec* object.

Parameters

field_name – The name of a form field.

Returns

True if the field should be locked, False otherwise.

```
class pyhanko.sign.fields.SignatureFormField(field_name, *, box=None, include_on_page=None,
                                             combine_annotation=True, invis_settings:
                                             InvisSigSettings = InvisSigSettings(set_print_flag=True,
                                             set_hidden_flag=False, box_out_of_bounds=False),
                                             visible_settings: VisibleSigSettings =
                                             VisibleSigSettings(rotate_with_page=True,
                                             scale_with_page_zoom=True, print_signature=True),
                                             annot_flags=None)
```

Bases: *DictionaryObject*

register_widget_annotation(writer: BasePdfFileWriter, sig_field_ref)

```
class pyhanko.sign.fields.InvisSigSettings(set_print_flag: bool = True, set_hidden_flag: bool = False,
                                           box_out_of_bounds: bool = False)
```

Bases: object

Invisible signature widget generation settings.

These settings exist because there is no real way of including an untagged invisible signature in a document that complies with the requirements of both PDF/A-2 (or -3) and PDF/UA-1.

Compatibility with PDF/A (the default) requires the print flag to be set. Compatibility with PDF/UA requires the hidden flag to be set (which is banned in PDF/A) or the box to be outside the crop box.

set_print_flag: bool = True

Set the print flag. Required in PDF/A.

set_hidden_flag: `bool = False`

Set the hidden flag. Required in PDF/UA.

box_out_of_bounds: `bool = False`

Put the box out of bounds (technically, this just makes the box zero-sized with large negative coordinates).

This is a hack to get around the fact that PDF/UA requires the hidden flag to be set on all in-bounds untagged annotations, and some validators consider [0, 0, 0, 0] to be an in-bounds rectangle if (0, 0) is a point that falls within the crop box.

class `pyhanko.sign.fields.VisibleSigSettings`(*rotate_with_page: bool = True, scale_with_page_zoom: bool = True, print_signature: bool = True*)

Bases: `object`

New in version 0.14.0.

Additional flags used when setting up visible signature widgets.

rotate_with_page: `bool = True`

Allow the signature widget to rotate with the page if rotation is applied (e.g. by way of the page's `/Rotate` entry). Default is `True`.

Note: If `False`, this will cause the `NoRotate` flag to be set.

scale_with_page_zoom: `bool = True`

Allow the signature widget to scale with the page's zoom level. Default is `True`.

Note: If `False`, this will cause the `NoZoom` flag to be set.

print_signature: `bool = True`

Render the signature when the document is printed. Default `True`.

`pyhanko.sign.fields.enumerate_sig_fields`(*handler: PdfHandler, filled_status=None*)

Enumerate signature fields.

Parameters

- **handler** – The [PdfHandler](#) to operate on.
- **filled_status** – Optional boolean. If `True` (resp. `False`) then all filled (resp. empty) fields are returned. If left `None` (the default), then all fields are returned.

Returns

A generator producing signature fields.

`pyhanko.sign.fields.append_signature_field`(*pdf_out: BasePdfFileWriter, sig_field_spec: SigFieldSpec*)

Append signature fields to a PDF file.

Parameters

- **pdf_out** – Incremental writer to house the objects.
- **sig_field_spec** – A [SigFieldSpec](#) object describing the signature field to add.

`pyhanko.sign.fields.ensure_sig_flags`(*writer: BasePdfFileWriter, lock_sig_flags: bool = True*)

Ensure the `SigFlags` setting is present in the `AcroForm` dictionary.

Parameters

- **writer** – A PDF writer.
- **lock_sig_flags** – Whether to flag the document as append-only.

`pyhanko.sign.fields.prepare_sig_field(sig_field_name, root, update_writer: BasePdfFileWriter, existing_fields_only=False, **kwargs)`

Returns a tuple of a boolean and a reference to a signature field. The boolean is `True` if the field was created, and `False` otherwise.

Danger: This function is internal API.

`pyhanko.sign.fields.apply_sig_field_spec_properties(pdf_out: BasePdfFileWriter, sig_field: DictionaryObject, sig_field_spec: SigFieldSpec)`

Internal function to apply field spec properties to a newly created field.

pyhanko.sign.general module

General tools related to Cryptographic Message Syntax (CMS) signatures, not necessarily to the extent implemented in the PDF specification.

CMS is defined in [RFC 5652](#). To parse CMS messages, pyHanko relies heavily on [asn1crypto](#).

`pyhanko.sign.general.simple_cms_attribute(attr_type, value)`

Convenience method to quickly construct a CMS attribute object with one value.

Parameters

- **attr_type** – The attribute type, as a string or OID.
- **value** – The value.

Returns

A `cms.CMSAttribute` object.

`pyhanko.sign.general.find_cms_attribute(attrs, name)`

Find and return CMS attribute values of a given type.

Note: This function will also check for duplicates, but not in the sense of multivalued attributes. In other words: multivalued attributes are allowed; listing the same attribute OID more than once is not.

Parameters

- **attrs** – The `cms.CMSAttributes` object.
- **name** – The attribute type as a string (as defined in `asn1crypto`).

Returns

The values associated with the requested type, if present.

Raises

- **[NonexistentAttributeError](#)** – Raised when no such type entry could be found in the `cms.CMSAttributes` object.
- **[CMSStructuralError](#)** – Raised if the given OID occurs more than once.

`pyhanko.sign.general.find_unique_cms_attribute(attrs, name)`

Find and return a unique CMS attribute value of a given type.

Parameters

- **attrs** – The `cms.CMSAttributes` object.
- **name** – The attribute type as a string (as defined in `asn1crypto`).

Returns

The value associated with the requested type, if present.

Raises

- **`NonexistentAttributeError`** – Raised when no such type entry could be found in the `cms.CMSAttributes` object.
- **`MultivaluedAttributeError`** – Raised when the attribute’s cardinality is not 1.

exception `pyhanko.sign.general.NonexistentAttributeError`

Bases: `KeyError`

exception `pyhanko.sign.general.MultivaluedAttributeError`

Bases: `ValueError`

class `pyhanko.sign.general.CertificateStore`

Bases: `CertificateCollection`, `ABC`

register(*cert: Certificate*) → bool

Register a single certificate.

Parameters

cert – Certificate to add.

Returns

True if the certificate was added, False if it already existed in this store.

register_multiple(*certs*)

Register multiple certificates.

Parameters

certs – Certificates to register.

Returns

True if at least one certificate was added, False if all certificates already existed in this store.

class `pyhanko.sign.general.SimpleCertificateStore`

Bases: `CertificateStore`

Simple trustless certificate store.

classmethod `from_certs(certs)`

register(*cert: Certificate*) → bool

Register a single certificate.

Parameters

cert – Certificate to add.

Returns

True if the certificate was added, False if it already existed in this store.

retrieve_many_by_key_identifier(*key_identifier: bytes*)

Retrieves possibly multiple certs via the corresponding key identifiers

Parameters

key_identifier – A byte string of the key identifier

Returns

A list of `asn1crypto.x509.Certificate` objects

retrieve_by_name(*name: Name*)

Retrieves a list certs via their subject name

Parameters

name – An `asn1crypto.x509.Name` object

Returns

A list of `asn1crypto.x509.Certificate` objects

retrieve_by_issuer_serial(*issuer_serial*)

Retrieve a certificate by its `issuer_serial` value.

Parameters

issuer_serial – The `issuer_serial` value of the certificate.

Returns

The certificate corresponding to the `issuer_serial` key passed in.

Returns

None or an `asn1crypto.x509.Certificate` object

exception `pyhanko.sign.general.SigningError`(*msg: str, *args*)

Bases: `ValueError`

Error encountered while signing a file.

exception `pyhanko.sign.general.UnacceptableSignerError`(*msg: str, *args*)

Bases: `SigningError`

Error raised when a signer was judged unacceptable.

class `pyhanko.sign.general.SignedDataCerts`(*signer_cert: Certificate, other_certs: List[Certificate], attribute_certs: List[AttributeCertificateV2]*)

Bases: `object`

Value type to describe certificates included in a CMS signed data payload.

signer_cert: `Certificate`

The certificate identified as the signer's certificate.

other_certs: `List[Certificate]`

Other (public-key) certificates included in the signed data object.

attribute_certs: `List[AttributeCertificateV2]`

Attribute certificates included in the signed data object.

`pyhanko.sign.general.extract_signer_info`(*signed_data: SignedData*) → `SignerInfo`

Extract the unique `SignerInfo` entry of a CMS signed data value, or throw a `ValueError`.

Parameters

signed_data – A CMS `SignedData` value.

Returns

A CMS `SignerInfo` value.

Raises

ValueError – If the number of `SignerInfo` values is not exactly one.

`pyhanko.sign.general.extract_certificate_info(signed_data: SignedData) → SignedDataCerts`

Extract and classify embedded certificates found in the `certificates` field of the signed data value.

Parameters

signed_data – A CMS `SignedData` value.

Returns

A `SignedDataCerts` object containing the embedded certificates.

`pyhanko.sign.general.load_certs_from_pemder(cert_files)`

A convenience function to load PEM/DER-encoded certificates from files.

Parameters

cert_files – An iterable of file names.

Returns

A generator producing `asn1crypto.x509.Certificate` objects.

`pyhanko.sign.general.load_cert_from_pemder(cert_file)`

A convenience function to load a single PEM/DER-encoded certificate from a file.

Parameters

cert_file – A file name.

Returns

An `asn1crypto.x509.Certificate` object.

`pyhanko.sign.general.load_private_key_from_pemder(key_file, passphrase: Optional[bytes]) → PrivateKeyInfo`

A convenience function to load PEM/DER-encoded keys from files.

Parameters

- **key_file** – File to read the key from.
- **passphrase** – Key passphrase.

Returns

A private key encoded as an unencrypted PKCS#8 `PrivateKeyInfo` object.

`pyhanko.sign.general.get_pyca_cryptography_hash(algorithm, prehashed=False)`

`pyhanko.sign.general.optimal_pss_params(cert: Certificate, digest_algorithm: str) → RSASSAPSSParams`
Figure out the optimal RSASSA-PSS parameters for a given certificate. The subject's public key must be an RSA key.

Parameters

- **cert** – An RSA X.509 certificate.
- **digest_algorithm** – The digest algorithm to use.

Returns

RSASSA-PSS parameters.

`pyhanko.sign.general.process_pss_params(params: RSASSAPSSParams, digest_algorithm, prehashed=False)`

Extract PSS padding settings and message digest from an RSASSAPSSParams value.

Internal API.

`pyhanko.sign.general.as_signing_certificate(cert: Certificate) → SigningCertificate`

Format an ASN.1 SigningCertificate object, where the certificate is identified by its SHA-1 digest.

Parameters

cert – An X.509 certificate.

Returns

A `tsp.SigningCertificate` object referring to the original certificate.

`pyhanko.sign.general.as_signing_certificate_v2(cert: Certificate, hash_algo='sha256') → SigningCertificateV2`

Format an ASN.1 SigningCertificateV2 value, where the certificate is identified by the hash algorithm specified.

Parameters

- **cert** – An X.509 certificate.
- **hash_algo** – Hash algorithm to use to digest the certificate. Default is SHA-256.

Returns

A `tsp.SigningCertificateV2` object referring to the original certificate.

`pyhanko.sign.general.match_issuer_serial(expected_issuer_serial: Union[IssuerAndSerialNumber, IssuerSerial], cert: Certificate) → bool`

Match the issuer and serial number of an X.509 certificate against some expected identifier.

Parameters

- **expected_issuer_serial** – A certificate identifier, either `cms.IssuerAndSerialNumber` or `tsp.IssuerSerial`.
- **cert** – An `x509.Certificate`.

Returns

True if there's a match, False otherwise.

`pyhanko.sign.general.check_ess_certid(cert: Certificate, certid: Union[ESSCertID, ESSCertIDv2])`

Match an ESSCertID value against a certificate.

Parameters

- **cert** – The certificate to match against.
- **certid** – The ESSCertID value.

Returns

True if the ESSCertID matches the certificate, False otherwise.

exception `pyhanko.sign.general.CMSExtractionError(failure_message)`

Bases: `ValueErrorWithMessage`

`pyhanko.sign.general.byte_range_digest(stream: IO, byte_range: Iterable[int], md_algorithm: str, chunk_size=4096) → Tuple[int, bytes]`

Internal API to compute byte range digests. Potentially dangerous if used without due caution.

Parameters

- **stream** – Stream over which to compute the digest. Must support seeking and reading.
- **byte_range** – The byte range, as a list of (offset, length) pairs, flattened.
- **md_algorithm** – The message digest algorithm to use.
- **chunk_size** – The I/O chunk size to use.

Returns

A tuple of the total digested length, and the actual digest.

exception `pyhanko.sign.general.ValueErrorWithMessage(failure_message)`

Bases: `ValueError`

Value error with a failure message attribute that can be conveniently extracted, instead of having to rely on extracting exception args generically.

exception `pyhanko.sign.general.CMSStructuralError(failure_message)`

Bases: `ValueErrorWithMessage`

Structural error in a CMS object.

pyhanko.sign.pkcs11 module

This module provides PKCS#11 integration for pyHanko, by providing a wrapper for `python-pkcs11` that can be seamlessly plugged into a `PdfSigner`.

class `pyhanko.sign.pkcs11.PKCS11Signer(pkcs11_session: Session, cert_label: Optional[str] = None, signing_cert: Optional[Certificate] = None, ca_chain=None, key_label: Optional[str] = None, prefer_pss=False, embed_roots=True, other_certs_to_pull=(), bulk_fetch=True, key_id: Optional[bytes] = None, cert_id: Optional[bytes] = None, use_raw_mechanism=False)`

Bases: `Signer`

Signer implementation for PKCS11 devices.

Parameters

- **pkcs11_session** – The PKCS11 session object to use.
- **cert_label** – The label of the certificate that will be used for signing, to be pulled from the PKCS#11 token.
- **cert_id** – ID of the certificate object that will be used for signing, to be pulled from the PKCS#11 token.
- **signing_cert** – The signer’s certificate. If the signer’s certificate is provided via this parameter, the `cert_label` and `cert_id` parameters will not be used to retrieve the signer’s certificate.
- **ca_chain** – Set of other relevant certificates (as `asn1crypto.x509.Certificate` objects).
- **key_label** – The label of the key that will be used for signing. Defaults to the value of `cert_label` if left unspecified and `key_id` is also unspecified.

Note: At least one of `key_id`, `key_label` and `cert_label` must be supplied.

- **key_id** – ID of the private key object (optional).

- **other_certs_to_pull** – List labels of other certificates to pull from the PKCS#11 device. Defaults to the empty tuple. If `None`, pull *all* certificates.
- **bulk_fetch** – Boolean indicating the fetching strategy. If `True`, fetch all certs and filter the unneeded ones. If `False`, fetch the requested certs one by one. Default value is `True`, unless `other_certs_to_pull` has one or fewer elements, in which case it is always treated as `False`.
- **use_raw_mechanism** – Use the ‘raw’ equivalent of the selected signature mechanism. This is useful when working with tokens that do not support a hash-then-sign mode of operation.

Note: This functionality is only available for ECDSA at this time. Support for other signature schemes will be added on an as-needed basis.

property `cert_registry`

property `signing_cert`

async `async_sign_raw(data: bytes, digest_algorithm: str, dry_run=False) → bytes`

Compute the raw cryptographic signature of the data provided, hashed using the digest algorithm provided.

Parameters

- **data** – Data to sign.
- **digest_algorithm** – Digest algorithm to use.

Warning: If `signature_mechanism` also specifies a digest, they should match.

- **dry_run** – Do not actually create a signature, but merely output placeholder bytes that would suffice to contain an actual signature.

Returns

Signature bytes.

async `ensure_objects_loaded()`

Async method that, when awaited, ensures that objects (relevant certificates, key handles, ...) are loaded.

This coroutine is guaranteed to be called & awaited in `sign_raw()`, but some property implementations may cause object loading to be triggered synchronously (for backwards compatibility reasons). This blocks the event loop the first time it happens.

To avoid this behaviour, asynchronous code should ideally perform `await signer.ensure_objects_loaded()` after instantiating the signer.

Note: The asynchronous context manager on `PKCS11SigningContext` takes care of that automatically.

`pyhanko.sign.pkcs11.open_pkcs11_session(lib_location: str, slot_no: Optional[int] = None, token_label: Optional[str] = None, token_criteria: Optional[TokenCriteria] = None, user_pin: Optional[Union[str, object]] = None) → Session`

Open a PKCS#11 session

Parameters

- **lib_location** – Path to the PKCS#11 module.

- **slot_no** – Slot number to use. If not specified, the first slot containing a token labelled `token_label` will be used.
- **token_label** – Deprecated since version 0.14.0: Use **token_criteria** instead.
Label of the token to use. If `None`, there is no constraint.
- **token_criteria** – Criteria that the token should match.
- **user_pin** – User PIN to use, or `PROTECTED_AUTH`. If `None`, authentication is skipped.

Note: Some PKCS#11 implementations do not require PIN when the token is opened, but will prompt for it out-of-band when signing. Whether `PROTECTED_AUTH` or `None` is used in this case depends on the implementation.

Returns

An open PKCS#11 session object.

```
class pyhanko.sign.pkcs11.PKCS11SigningContext(config: PKCS11SignatureConfig, user_pin: Optional[str] = None)
```

Bases: object

Context manager for PKCS#11 configurations.

```
pyhanko.sign.pkcs11.find_token(slots: List[Slot], slot_no: Optional[int] = None, token_criteria: Optional[TokenCriteria] = None) → Optional[Token]
```

Internal helper method to find a token.

Parameters

- **slots** – The list of slots.
- **slot_no** – Slot number to use. If not specified, the first slot containing a token satisfying the criteria will be used
- **token_criteria** – Criteria the token must satisfy.

Returns

A PKCS#11 token object, or `None` if none was found.

```
pyhanko.sign.pkcs11.select_pkcs11_signing_params(signature_mechanism: SignedDigestAlgorithm, digest_algorithm: str, use_raw_mechanism: bool) → PKCS11SignatureOperationSpec
```

Internal helper function to set up a PKCS #11 signing operation.

Parameters

- **signature_mechanism** – The signature mechanism to use (as an ASN.1 value)
- **digest_algorithm** – The digest algorithm to use
- **use_raw_mechanism** – Whether to attempt to use the raw mechanism on pre-hashed data.

Returns

pyhanko.sign.signers package

pyhanko.sign.signers.cms_embedder module

This module describes and implements the low-level *PdfCMSEmbedder* protocol for embedding CMS payloads into PDF signature objects.

```
class pyhanko.sign.signers.cms_embedder.PdfCMSEmbedder(new_field_spec: Optional[SigFieldSpec] =
                                                         None)
```

Bases: object

Low-level class that handles embedding CMS objects into PDF signature fields.

It also takes care of appearance generation and DocMDP configuration, but does not otherwise offer any of the conveniences of *PdfSigner*.

Parameters

new_field_spec – *SigFieldSpec* to use when creating new fields on-the-fly.

write_cms(*field_name*: str, *writer*: *BasePdfFileWriter*, *existing_fields_only*=False)

New in version 0.3.0.

Changed in version 0.7.0: Digest wrapped in *PreparedByteRangeDigest* in step 3; output returned in step 3 instead of step 4.

This method returns a generator coroutine that controls the process of embedding CMS data into a PDF signature field. Can be used for both timestamps and regular signatures.

Danger: This is a very low-level interface that performs virtually no error checking, and is intended to be used in situations where the construction of the CMS object to be embedded is not under the caller's control (e.g. a remote signer that produces full-fledged CMS objects).

In almost every other case, you're better off using *PdfSigner* instead, with a custom *Signer* implementation to handle the cryptographic operations if necessary.

The coroutine follows the following specific protocol.

1. First, it retrieves or creates the signature field to embed the CMS object in, and yields a reference to said field.
2. The caller should then send in a *SigObjSetup* object, which is subsequently processed by the coroutine. For convenience, the coroutine will then yield a reference to the signature dictionary (as embedded in the PDF writer).
3. Next, the caller should send a *SigIOSetup* object, describing how the resulting document should be hashed and written to the output. The coroutine will write the entire document with a placeholder region reserved for the signature and compute the document's hash and yield it to the caller. It will then yield a *prepared_digest*, output tuple, where *prepared_digest* is a *PreparedByteRangeDigest* object containing the document digest and the relevant offsets, and *output* is the output stream to which the document to be signed was written.

From this point onwards, **no objects may be changed or added** to the *IncrementalPdfFileWriter* currently in use.

4. Finally, the caller should pass in a CMS object to place inside the signature dictionary. The CMS object can be supplied as a raw bytes object, or an *asn1crypto*-style object. The coroutine's final yield is the value of the signature dictionary's */Contents* entry, given as a hexadecimal string.

Caution: It is the caller's own responsibility to ensure that enough room is available in the placeholder signature object to contain the final CMS object.

Parameters

- **field_name** – The name of the field to fill in. This should be a field of type `/Sig`.
- **writer** – An `IncrementalPdfFileWriter` containing the document to sign.
- **existing_fields_only** – If `True`, never create a new empty signature field to contain the signature. If `False`, a new field may be created if no field matching `field_name` exists.

Returns

A generator coroutine implementing the protocol described above.

```
class pyhanko.sign.signers.cms_embedder.SigMDPSetup(md_algorithm: str, certify: bool = False,
                                                    field_lock:
                                                    Union[pyhanko.sign.fields.FieldMDPSpec,
                                                         NoneType] = None, docmdp_perms:
                                                    Union[pyhanko.sign.fields.MDPPerm,
                                                         NoneType] = None)
```

Bases: object

md_algorithm: str

Message digest algorithm to write into the signature reference dictionary, if one is written at all.

Warning: It is the caller's responsibility to make sure that this value agrees with the value embedded into the CMS object, and with the algorithm used to hash the document. The low-level `PdfCMSEmbedder` API will simply take it at face value.

certify: bool = False

Sign with an author (certification) signature, as opposed to an approval signature. A document can contain at most one such signature, and it must be the first one.

field_lock: Optional[`FieldMDPSpec`] = None

Field lock information to write to the signature reference dictionary.

docmdp_perms: Optional[`MDPPerm`] = None

DocMDP permissions to write to the signature reference dictionary.

apply(`sig_obj_ref`, `writer`)

Apply the settings to a signature object.

Danger: This method is internal API.

```
class pyhanko.sign.signers.cms_embedder.SigObjSetup(sig_placeholder: PdfSignedData, mdp_setup:
                                                    Optional[SigMDPSetup] = None,
                                                    appearance_setup:
                                                    Optional[SigAppearanceSetup] = None)
```

Bases: object

Describes the signature dictionary to be embedded as the form field's value.

sig_placeholder: *PdfSignedData*

Bare-bones placeholder object, usually of type *SignatureObject* or *DocumentTimestamp*.

In particular, this determines the number of bytes to allocate for the CMS object.

mdp_setup: *Optional[SigMDPSetup]* = None

Optional DocMDP settings, see *SigMDPSetup*.

appearance_setup: *Optional[SigAppearanceSetup]* = None

Optional appearance settings, see *SigAppearanceSetup*.

```
class pyhanko.sign.signers.cms_embedder.SigAppearanceSetup(style: BaseStampStyle, timestamp:
                                                         datetime, name: Optional[str],
                                                         text_params: Optional[dict] = None)
```

Bases: object

Signature appearance configuration.

Part of the low-level *PdfCMSEmbedder* API, see *SigObjSetup*.

style: *BaseStampStyle*

Stamp style to use to generate the appearance.

timestamp: *datetime*

Timestamp to show in the signature appearance.

name: *Optional[str]*

Signer name to show in the signature appearance.

text_params: *Optional[dict]* = None

Additional text interpolation parameters to pass to the underlying stamp style.

apply(*sig_annot*, *writer*)

Apply the settings to an annotation.

Danger: This method is internal API.

```
class pyhanko.sign.signers.cms_embedder.SigIOSetup(md_algorithm: str, in_place: bool = False,
                                                    chunk_size: int = 4096, output: Optional[IO] =
                                                    None)
```

Bases: object

I/O settings for writing signed PDF documents.

Objects of this type are used in the penultimate phase of the *PdfCMSEmbedder* protocol.

md_algorithm: *str*

Message digest algorithm to use to compute the document hash. It should be supported by *pyca/cryptography*.

Warning: This is also the message digest algorithm that should appear in the corresponding *signerInfo* entry in the CMS object that ends up being embedded in the signature field.

in_place: *bool* = False

Sign the input in-place. If False, write output to a *BytesIO* object, or *output* if the latter is not None.

chunk_size: `int = 4096`

Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support `memoryview`.

output: `Optional[IO] = None`

Write the output to the specified output stream. If `None`, write to a new `BytesIO` object. Default is `None`.

pyhanko.sign.signers.csc_signer module

New in version 0.10.0.

Asynchronous *Signer* implementation for interacting with a remote signing service using the Cloud Signature Consortium (CSC) API.

This implementation is based on version 1.0.4.0 (2019-06) of the CSC API specification.

Usage notes

This module's *CSCSigner* class supplies an implementation of the *Signer* class in pyHanko. As such, it is flexible enough to be used either through pyHanko's high-level API (*sign_pdf()* et al.), or through the *interrupted signing API*.

CSCSigner overview

CSCSigner is only directly responsible for calling the `signatures/signHash` endpoint in the CSC API. Other than that, it only handles batch control. This means that the following tasks require further action on the API user's part:

- authenticating to the signing service (typically using OAuth2);
- obtaining Signature Activation Data (SAD) from the signing service;
- provisioning the certificates to embed into the document (usually those are supplied by the signing service as well).

The first two involve a degree of implementation/vendor dependence that is difficult to cater to in full generality, and the third is out of scope for *Signer* subclasses in general.

However, this module still provides a number of convenient hooks and guardrails that should allow you to fill in these blanks with relative ease. We briefly discuss these below.

Throughout, the particulars of how pyHanko should connect to a signing service are supplied in a *CSCServiceSessionInfo* object. This object contains the base CSC API URL, the CSC credential ID to use, and authentication data.

Authenticating to the signing service

While the authentication process itself is the API user's responsibility, *CSCServiceSessionInfo* includes an *oauth_token* field that will (by default) be used to populate the HTTP Authorization header for every request.

To handle OAuth-specific tasks, you might want to use a library like *OAuthLib*.

Obtaining SAD from the signing service

This is done by subclassing `CSCAuthorizationInfo` and passing an instance to the `CSCSigner`. The `CSCAuthorizationInfo` instance should call the signer's `credentials/authorize` endpoint with the proper parameters required by the service. See the documentation for `CSCAuthorizationInfo` for details and= information about helper functions.

Certificate provisioning

In pyHanko's API, `Signer` instances need to be initialised with the signer's certificate, preferably together with other relevant CA certificates. In a CSC context, these are typically retrieved from the signing service by calling the `credentials/info` endpoint.

This module offers a helper function to handle that task, see `fetch_certs_in_csc_credential()`.

```
class pyhanko.sign.signers.csc_signer.CSCSigner(session: ClientSession, auth_manager:
    CSCAuthorizationManager, sign_timeout: int = 300,
    prefer_pss: bool = False, embed_roots: bool = True,
    client_data: Optional[str] = None,
    batch_autocommit: bool = True, batch_size:
    Optional[int] = None, est_raw_signature_size=512)
```

Bases: `Signer`

Implements the `Signer` interface for a remote CSC signing service. Requests are made asynchronously, using `aiohttp`.

Parameters

- **session** – The `aiohttp` session to use when performing queries.
- **auth_manager** – A `CSCAuthorizationManager` instance capable of procuring signature activation data from the signing service.
- **sign_timeout** – Timeout for signing operations, in seconds. Defaults to 300 seconds (5 minutes).
- **prefer_pss** – When signing using an RSA key, prefer PSS padding to legacy PKCS#1 v1.5 padding. Default is `False`. This option has no effect on non-RSA signatures.
- **embed_roots** – Option that controls whether or not additional self-signed certificates should be embedded into the CMS payload. The default is `True`.
- **client_data** – CSC client data to add to any signing request(s), if applicable.
- **batch_autocommit** – Whether to automatically commit a signing transaction as soon as a batch is full. The default is `True`. If `False`, the caller has to trigger `commit()` manually.
- **batch_size** – The number of signatures to sign in one transaction. This defaults to 1 (i.e. a separate `signatures/signHash` call is made for every signature).
- **est_raw_signature_size** – Estimated raw signature size (in bytes). Defaults to 512 bytes, which, combined with other built-in safety margins, should provide a generous overestimate.

cert_registry: `CertificateStore`

Collection of certificates associated with this signer. Note that this is simply a bookkeeping tool; in particular it doesn't care about trust.

get_signature_mechanism(*digest_algorithm*)

Get the signature mechanism for this signer to use. If `signature_mechanism` is set, it will be used. Otherwise, this method will attempt to put together a default based on mechanism used in the signer's certificate.

Parameters

digest_algorithm – Digest algorithm to use as part of the signature mechanism. Only used if a signature mechanism object has to be put together on-the-fly.

Returns

A `SignedDigestAlgorithm` object.

async format_csc_signing_req(*tbs_hashes: List[str], digest_algorithm: str*) → dict

Populate the request data for a CSC signing request

Parameters

- **tbs_hashes** – Base64-encoded hashes that require signing.
- **digest_algorithm** – The digest algorithm to use.

Returns

A dict that, when encoded as a JSON object, be used as the request body for a call to `signatures/signHash`.

async async_sign_raw(*data: bytes, digest_algorithm: str, dry_run=False*) → bytes

Compute the raw cryptographic signature of the data provided, hashed using the digest algorithm provided.

Parameters

- **data** – Data to sign.
- **digest_algorithm** – Digest algorithm to use.

Warning: If `signature_mechanism` also specifies a digest, they should match.

- **dry_run** – Do not actually create a signature, but merely output placeholder bytes that would suffice to contain an actual signature.

Returns

Signature bytes.

async commit()

Commit the current batch by calling the `signatures/signHash` endpoint on the CSC service.

This coroutine does not return anything; instead, it notifies all waiting signing coroutines that their signature has been fetched.

```
class pyhanko.sign.signers.csc_signer.CSCServiceSessionInfo(service_url: str, credential_id: str,
                                                             oauth_token: Optional[str] = None,
                                                             api_ver: str = 'v1')
```

Bases: object

Information about the CSC service, together with the required authentication data.

service_url: str

Base URL of the CSC service. This is the part that precedes `/csc/<version>/...` in the API endpoint URLs.

credential_id: `str`

The identifier of the CSC credential to use when signing. The format is vendor-dependent.

oauth_token: `Optional[str] = None`

OAuth token to use when making requests to the CSC service.

api_ver: `str = 'v1'`

CSC API version.

Note: This section does not affect any of the internal logic, it only changes how the URLs are formatted.

endpoint_url(*endpoint_name*)

Complete an endpoint name to a full URL.

Parameters

endpoint_name – Name of the endpoint (e.g. `credentials/info`).

Returns

A URL.

property auth_headers

HTTP Header(s) necessary for authentication, to be passed with every request.

Note: By default, this supplies the `Authorization` header with the value of `oauth_token` as the Bearer value.

Returns

A dict of headers.

```
class pyhanko.sign.signers.csc_signer.CSCCredentialInfo(signing_cert: Certificate, chain:
    List[Certificate], supported_mechanisms:
    FrozenSet[str], max_batch_size: int,
    hash_pinning_required: bool,
    response_data: dict)
```

Bases: `object`

Information about a CSC credential, typically fetched using a `credentials/info` call. See also [`fetch_certs_in_csc_credential\(\)`](#).

signing_cert: `Certificate`

The signer's certificate.

chain: `List[Certificate]`

Other relevant CA certificates.

supported_mechanisms: `FrozenSet[str]`

Signature mechanisms supported by the credential.

max_batch_size: `int`

The maximal batch size that can be used with this credential.

hash_pinning_required: `bool`

Flag controlling whether SAD must be tied to specific hashes.

response_data: dict

The JSON response data from the server as an otherwise unparsed dict.

as_cert_store() → *CertificateStore*

Register the relevant certificates into a *CertificateStore* and return it.

Returns

A *CertificateStore*.

async pyhanko.sign.signers.csc_signer.**fetch_certs_in_csc_credential**(*session: ClientSession*,
csc_session_info: CSCServiceSessionInfo,
timeout: int = 30) →
CSCCredentialInfo

Call the credentials/info endpoint of the CSC service for a specific credential, and encode the result into a *CSCCredentialInfo* object.

Parameters

- **session** – The aiohttp session to use when performing queries.
- **csc_session_info** – General information about the CSC service and the credential.
- **timeout** – How many seconds to allow before time-out.

Returns

A *CSCCredentialInfo* object with the processed response.

class pyhanko.sign.signers.csc_signer.**CSCAuthorizationInfo**(*sad: str*, *expires_at: Optional[datetime] = None*)

Bases: object

Authorization data to make a signing request. This is the result of a call to credentials/authorize.

sad: str

Signature activation data; opaque to the client.

expires_at: Optional[datetime] = None

Expiry date of the signature activation data.

class pyhanko.sign.signers.csc_signer.**CSCAuthorizationManager**(*csc_session_info: CSCServiceSessionInfo*,
credential_info: CSCCredentialInfo)

Bases: ABC

Abstract class that handles authorisation requests for the CSC signing client.

Note: Implementations may wish to make use of the *format_csc_auth_request()* convenience method to format requests to the credentials/authorize endpoint.

Parameters

- **csc_session_info** – General information about the CSC service and the credential.
- **credential_info** – Details about the credential.

async authorize_signature(*hash_b64s: List[str]*) → *CSCAuthorizationInfo*

Request a SAD token from the signing service, either freshly or to extend the current transaction.

Depending on the lifecycle of this object, pre-fetched SAD values may be used. All authorization transaction management is left to implementing subclasses.

Parameters

hash_b64s – Base64-encoded hash values about to be signed.

Returns

Authorization data.

format_csc_auth_request(*num_signatures: int = 1, pin: Optional[str] = None, otp: Optional[str] = None, hash_b64s: Optional[List[str]] = None, description: Optional[str] = None, client_data: Optional[str] = None*) → dict

Format the parameters for a call to `credentials/authorize`.

Parameters

- **num_signatures** – The number of signatures to request authorisation for.
- **pin** – The user's PIN (if applicable).
- **otp** – The current value of an OTP token, provided by the user (if applicable).
- **hash_b64s** – An explicit list of base64-encoded hashes to be tied to the SAD. Is optional if the service's SCAL value is 1, i.e. when *hash_pinning_required* is false.
- **description** – A free-form description of the authorisation request (optional).
- **client_data** – Custom vendor-specific data (if applicable).

Returns

A dict that, when encoded as a JSON object, be used as the request body for a call to `credentials/authorize`.

static parse_csc_auth_response(*response_data: dict*) → *CSCAuthorizationInfo*

Parse the response from a `credentials/authorize` call into a *CSCAuthorizationInfo* object.

Parameters

response_data – The decoded response JSON.

Returns

A *CSCAuthorizationInfo* object.

property auth_headers

HTTP Header(s) necessary for authentication, to be passed with every request. By default, this delegates to *CSCServiceSessionInfo.auth_headers*.

Returns

A dict of headers.

```
class pyhanko.sign.signers.csc_signer.PrefetchedSADAuthorizationManager(csc_session_info:
CSCServiceSessionInfo, credential_info:
CSCCredentialInfo, csc_auth_info:
CSCAuthorizationInfo)
```

Bases: *CSCAuthorizationManager*

Simplistic *CSCAuthorizationManager* for use with pre-fetched signature activation data.

This class is effectively only useful for CSC services that do not require SAD to be pinned to specific document hashes. It allows you to use a SAD that was fetched before starting the signing process, for a one-shot signature.

This can simplify resource management in cases where obtaining a SAD is time-consuming, but the caller still wants the conveniences of pyHanko's high-level API without having to keep too many pyHanko objects in memory while waiting for a `credentials/authorize` call to go through.

Legitimate uses are limited, but the implementation is trivial, so we provide it here.

Parameters

- **csc_session_info** – General information about the CSC service and the credential.
- **credential_info** – Details about the credential.
- **csc_auth_info** – The pre-fetched signature activation data.

async authorize_signature(*hash_b64s: List[str]*) → *CSCAuthorizationInfo*

Return the prefetched SAD, or raise an error if called twice.

Parameters

hash_b64s – List of hashes to be signed; ignored.

Returns

The prefetched authorisation data.

pyhanko.sign.signers.constants module

This module defines constants & defaults used by pyHanko when creating digital signatures.

`pyhanko.sign.signers.constants.DEFAULT_MD = 'sha256'`

Default message digest algorithm used when computing digests for use in signatures.

`pyhanko.sign.signers.constants.DEFAULT_SIG_SUBFILTER =
SigSeedSubFilter.ADOBE_PKCS7_DETACHED`

Default SubFilter to use for a PDF signature.

`pyhanko.sign.signers.constants.DEFAULT_SIGNER_KEY_USAGE = {'non_repudiation'}`

Default key usage bits required for the signer's certificate.

`pyhanko.sign.signers.constants.SIG_DETAILS_DEFAULT_TEMPLATE = 'Digitally signed by
%(signer)s.\nTimestamp: %(ts)s.'`

Default template string for signature appearances.

`pyhanko.sign.signers.constants.DEFAULT_SIGNING_STAMP_STYLE =
TextStampStyle(border_width=3, background=<pyhanko.pdf_utils.content.RawContent object>,
background_layout=SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MID: 2>,
y_align=<AxisAlignment.ALIGN_MID: 2>, margins=Margins(left=5, right=5, top=5, bottom=5),
inner_content_scaling=<InnerScaling.SHRIK_TO_FIT: 4>), background_opacity=0.6,
text_box_style=TextBoxStyle(font=<pyhanko.pdf_utils.font.basic.SimpleFontEngineFactory
object>, font_size=10, leading=None, border_width=0, box_layout_rule=None,
vertical_text=False), inner_content_layout=None, stamp_text='Digitally signed by
%(signer)s.\nTimestamp: %(ts)s.', timestamp_format='%Y-%m-%d %H:%M:%S %Z')`

Default stamp style used for visible signatures.

pyhanko.sign.signers.functions module

This module defines pyHanko's high-level API entry points.

`pyhanko.sign.signers.functions.sign_pdf(pdf_out: BasePdfFileWriter, signature_meta: PdfSignatureMetadata, signer: Signer, timestamp: Optional[TimeStamper] = None, new_field_spec: Optional[SigFieldSpec] = None, existing_fields_only=False, bytes_reserved=None, in_place=False, output=None)`

Thin convenience wrapper around `PdfSigner.sign_pdf()`.

Parameters

- **pdf_out** – An *IncrementalPdfFileWriter*.
- **bytes_reserved** – Bytes to reserve for the CMS object in the PDF file. If not specified, make an estimate based on a dummy signature.
- **signature_meta** – The specification of the signature to add.
- **signer** – *Signer* object to use to produce the signature object.
- **timestamp** – *TimeStamper* object to use to produce any time stamp tokens that might be required.
- **in_place** – Sign the input in-place. If False, write output to a BytesIO object.
- **existing_fields_only** – If True, never create a new empty signature field to contain the signature. If False, a new field may be created if no field matching *field_name* exists.
- **new_field_spec** – If a new field is to be created, this parameter allows the caller to specify the field's properties in the form of a *SigFieldSpec*. This parameter is only meaningful if *existing_fields_only* is False.
- **output** – Write the output to the specified output stream. If None, write to a new BytesIO object. Default is None.

Returns

The output stream containing the signed output.

`async pyhanko.sign.signers.functions.async_sign_pdf(pdf_out: BasePdfFileWriter, signature_meta: PdfSignatureMetadata, signer: Signer, timestamp: Optional[TimeStamper] = None, new_field_spec: Optional[SigFieldSpec] = None, existing_fields_only=False, bytes_reserved=None, in_place=False, output=None)`

Thin convenience wrapper around `PdfSigner.async_sign_pdf()`.

Parameters

- **pdf_out** – An *IncrementalPdfFileWriter*.
- **bytes_reserved** – Bytes to reserve for the CMS object in the PDF file. If not specified, make an estimate based on a dummy signature.
- **signature_meta** – The specification of the signature to add.
- **signer** – *Signer* object to use to produce the signature object.
- **timestamp** – *TimeStamper* object to use to produce any time stamp tokens that might be required.

- **in_place** – Sign the input in-place. If False, write output to a BytesIO object.
- **existing_fields_only** – If True, never create a new empty signature field to contain the signature. If False, a new field may be created if no field matching *field_name* exists.
- **new_field_spec** – If a new field is to be created, this parameter allows the caller to specify the field’s properties in the form of a *SigFieldSpec*. This parameter is only meaningful if *existing_fields_only* is False.
- **output** – Write the output to the specified output stream. If None, write to a new BytesIO object. Default is None.

Returns

The output stream containing the signed output.

```
pyhanko.sign.signers.functions.embed_payload_with_cms(pdf_writer: BasePdfFileWriter,  
                                                    file_spec_string: str, payload:  
                                                    EmbeddedFileObject, cms_obj: ContentInfo,  
                                                    extension='.sig', file_name: Optional[str] =  
                                                    None, file_spec_kwargs=None,  
                                                    cms_file_spec_kwargs=None)
```

Embed some data as an embedded file stream into a PDF, and associate it with a CMS object.

The resulting CMS object will also be turned into an embedded file, and associated with the original payload through a related file relationship.

This can be used to bundle (non-PDF) detached signatures with PDF attachments, for example.

New in version 0.7.0.

Parameters

- **pdf_writer** – The PDF writer to use.
- **file_spec_string** – See *file_spec_string* in *FileSpec*.
- **payload** – Payload object.
- **cms_obj** – CMS object pertaining to the payload.
- **extension** – File extension to use for the CMS attachment.
- **file_name** – See *file_name* in *FileSpec*.
- **file_spec_kwargs** – Extra arguments to pass to the *FileSpec* constructor for the main attachment specification.
- **cms_file_spec_kwargs** – Extra arguments to pass to the *FileSpec* constructor for the CMS attachment specification.

pyhanko.sign.signers.pdf_byterange module

This module contains the low-level building blocks for dealing with bookkeeping around /ByteRange digests in PDF files.

```
class pyhanko.sign.signers.pdf_byterange.PreparedByteRangeDigest(document_digest: bytes,  
                                                                reserved_region_start: int,  
                                                                reserved_region_end: int)
```

Bases: object

New in version 0.7.0.

Changed in version 0.14.0: Removed `md_algorithm` attribute since it was unused.

Bookkeeping class that contains the digest of a document that is about to be signed (or otherwise authenticated) based on said digest. It also keeps track of the region in the output stream that is omitted in the byte range.

Instances of this class can easily be serialised, which allows for interrupting the signing process partway through.

document_digest: `bytes`

Digest of the document, computed over the appropriate `/ByteRange`.

reserved_region_start: `int`

Start of the reserved region in the output stream that is not part of the `/ByteRange`.

reserved_region_end: `int`

End of the reserved region in the output stream that is not part of the `/ByteRange`.

fill_with_cms(*output: IO, cms_data: Union[bytes, ContentInfo]*)

Write a DER-encoded CMS object to the reserved region indicated by `reserved_region_start` and `reserved_region_end` in the output stream.

Parameters

- **output** – Output stream to use. Must be writable and seekable.
- **cms_data** – CMS object to write. Can be provided as an `asn1crypto.cms.ContentInfo` object, or as raw DER-encoded bytes.

Returns

A bytes object containing the contents that were written, plus any additional padding.

fill_reserved_region(*output: IO, content_bytes: bytes*)

Write hex-encoded contents to the reserved region indicated by `reserved_region_start` and `reserved_region_end` in the output stream.

Parameters

- **output** – Output stream to use. Must be writable and seekable.
- **content_bytes** – Content bytes. These will be padded, hexadecimally encoded and written to the appropriate location in output stream.

Returns

A bytes object containing the contents that were written, plus any additional padding.

class `pyhanko.sign.signers.pdf_byterange.PdfByteRangeDigest`(*data_key='/Contents', *, bytes_reserved=None*)

Bases: `DictionaryObject`

General class to model a PDF Dictionary that has a `/ByteRange` entry and a another data entry (named `/Contents` by default) that will contain a value based on a digest computed over said `/ByteRange`. The `/ByteRange` will cover the entire file, except for the value of the data entry itself.

Danger: This is internal API.

Parameters

- **data_key** – Name of the data key, which is `/Contents` by default.
- **bytes_reserved** – Number of bytes to reserve for the contents placeholder. If `None`, a generous default is applied, but you should try to estimate the size as accurately as possible.

fill(writer: [BasePdfFileWriter](#), md_algorithm, in_place=False, output=None, chunk_size=4096)

Generator coroutine that handles the document hash computation and the actual filling of the placeholder data.

Danger: This is internal API; you should use [PdfSigner](#) wherever possible. If you *really* need fine-grained control, use [PdfCMSEmbedder](#) instead.

```
class pyhanko.sign.signers.pdf_byterange.PdfSignedData(obj_type, subfilter: SigSeedSubFilter =  
    SigSeedSubFilter.ADOBE\_PKCS7\_DETACHED,  
    timestamp: Optional[datetime] = None,  
    bytes_reserved=None)
```

Bases: [PdfByteRangeDigest](#)

Generic class to model signature dictionaries in a PDF file. See also [SignatureObject](#) and [DocumentTimestamp](#).

Parameters

- **obj_type** – The type of signature object.
- **subfilter** – See [SigSeedSubFilter](#).
- **timestamp** – The timestamp to embed into the /M entry.
- **bytes_reserved** – The number of bytes to reserve for the signature. Defaults to 16 KiB.

Warning: Since the CMS object is written to the output file as a hexadecimal string, you should request **twice** the (estimated) number of bytes in the DER-encoded version of the CMS object.

```
class pyhanko.sign.signers.pdf_byterange.SignatureObject(timestamp: Optional[datetime] = None,  
    subfilter: SigSeedSubFilter = SigSeedSubFilter.ADOBE\_PKCS7\_DETACHED,  
    name=None, location=None,  
    reason=None, bytes_reserved=None)
```

Bases: [PdfSignedData](#)

Class modelling a (placeholder for) a regular PDF signature.

Parameters

- **timestamp** – The (optional) timestamp to embed into the /M entry.
- **subfilter** – See [SigSeedSubFilter](#).
- **bytes_reserved** – The number of bytes to reserve for the signature. Defaults to 16 KiB.

Warning: Since the CMS object is written to the output file as a hexadecimal string, you should request **twice** the (estimated) number of bytes in the DER-encoded version of the CMS object.

- **name** – Signer name. You probably want to leave this blank, viewers should default to the signer's subject name.

- **location** – Optional signing location.
- **reason** – Optional signing reason. May be restricted by seed values.

class pyhanko.sign.signers.pdf_byterange.**DocumentTimestamp**(bytes_reserved=None)

Bases: *PdfSignedData*

Class modelling a (placeholder for) a regular PDF signature.

Parameters

bytes_reserved – The number of bytes to reserve for the signature. Defaults to 16 KiB.

Warning: Since the CMS object is written to the output file as a hexadecimal string, you should request **twice** the (estimated) number of bytes in the DER-encoded version of the CMS object.

pyhanko.sign.signers.pdf_cms module

This module defines utility classes to format CMS objects for use in PDF signatures.

class pyhanko.sign.signers.pdf_cms.**Signer**(prefer_pss=False, embed_roots=True)

Bases: object

Abstract signer object that is agnostic as to where the cryptographic operations actually happen.

As of now, pyHanko provides two implementations:

- *SimpleSigner* implements the easy case where all the key material can be loaded into memory.
- *PKCS11Signer* implements a signer that is capable of interfacing with a PKCS11 device (see also *BEIDSigner*).

Parameters

- **prefer_pss** – When signing using an RSA key, prefer PSS padding to legacy PKCS#1 v1.5 padding. Default is `False`. This option has no effect on non-RSA signatures.
- **embed_roots** – New in version 0.9.0.

Option that controls whether or not additional self-signed certificates should be embedded into the CMS payload. The default is `True`.

Note: The signer's certificate is always embedded, even if it is self-signed.

Note: Trust roots are configured by the validator, so embedding them typically does nothing in a typical validation process. Therefore they can be safely omitted in most cases. Nonetheless, embedding the roots can be useful for documentation purposes.

Warning: To be precise, if this flag is `False`, a certificate will be dropped if (a) it is not the signer's, (b) it is self-issued and (c) its subject and authority key identifiers match (or either is missing). In other words, we never validate the actual self-signature. This heuristic is sufficiently accurate for most applications.

signing_cert: `Optional[Certificate] = None`

Changed in version 0.14.0: Made optional (see note)

The certificate that will be used to create the signature.

Note: This is an optional field only to a limited extent. Subclasses may require it to be present, and not setting it at the beginning of the signing process implies that certain high-level convenience features will no longer work or be limited in function (e.g., automatic hash selection, appearance generation, revocation information collection, ...).

However, making `signing_cert` optional enables certain signing workflows where the certificate of the signer is not known until the signature has actually been produced. This is most relevant in certain types of remote signing scenarios.

cert_registry: `CertificateStore`

Collection of certificates associated with this signer. Note that this is simply a bookkeeping tool; in particular it doesn't care about trust.

signature_mechanism: `SignedDigestAlgorithm = None`

The (cryptographic) signature mechanism to use.

attribute_certs: `Iterable[AttributeCertificateV2] = ()`

Attribute certificates to include with the signature.

Note: Only v2 attribute certificates are supported.

get_signature_mechanism(*digest_algorithm*)

Get the signature mechanism for this signer to use. If `signature_mechanism` is set, it will be used. Otherwise, this method will attempt to put together a default based on mechanism used in the signer's certificate.

Parameters

digest_algorithm – Digest algorithm to use as part of the signature mechanism. Only used if a signature mechanism object has to be put together on-the-fly.

Returns

A `SignedDigestAlgorithm` object.

property subject_name: `Optional[str]`

Returns

The subject's common name as a string, extracted from `signing_cert`, or `None` if no signer's certificate is available

static format_revinfo(*ocsp_responses: Optional[list] = None, crls: Optional[list] = None*)

Format Adobe-style revocation information for inclusion into a CMS object.

Parameters

- **ocsp_responses** – A list of OCSP responses to include.
- **crls** – A list of CRLs to include.

signer_info(*digest_algorithm: str, signed_attrs, signature*)

Format the `SignerInfo` entry for a CMS signature.

Parameters

- **digest_algorithm** – Digest algorithm to use.
- **signed_attrs** – Signed attributes (see [signed_attrs\(\)](#)).
- **signature** – The raw signature to embed (see [sign_raw\(\)](#)).

Returns

An `asn1crypto.cms.SignerInfo` object.

async [async_sign_raw](#)(*data: bytes, digest_algorithm: str, dry_run=False*) → bytes

Compute the raw cryptographic signature of the data provided, hashed using the digest algorithm provided.

Parameters

- **data** – Data to sign.
- **digest_algorithm** – Digest algorithm to use.

Warning: If [signature_mechanism](#) also specifies a digest, they should match.

- **dry_run** – Do not actually create a signature, but merely output placeholder bytes that would suffice to contain an actual signature.

Returns

Signature bytes.

async [unsigned_attrs](#)(*digest_algorithm: str, signature: bytes, signed_attrs: CMSAttributes, timestamp=None, dry_run=False*) → Optional[CMSAttributes]

Changed in version 0.9.0: Made asynchronous [_\(breaking change\)_](#)

Changed in version 0.14.0: Added `signed_attrs` parameter [_\(breaking change\)_](#)

Compute the unsigned attributes to embed into the CMS object. This function is called after signing the hash of the signed attributes (see [signed_attrs\(\)](#)).

By default, this method only handles timestamp requests, but other functionality may be added by subclasses

If this method returns `None`, no unsigned attributes will be embedded.

Parameters

- **digest_algorithm** – Digest algorithm used to hash the signed attributes.
- **signed_attrs** – Signed attributes of the signature.
- **signature** – Signature of the signed attribute hash.
- **timestamp** – Timestamp supplier to use.
- **dry_run** – Flag indicating “dry run” mode. If `True`, only the approximate size of the output matters, so cryptographic operations can be replaced by placeholders.

Returns

The unsigned attributes to add, or `None`.

async [signed_attrs](#)(*data_digest: bytes, digest_algorithm: str, attr_settings: Optional[PdfCMSSignedAttributes] = None, content_type='data', use_pades=False, timestamp=None, dry_run=False, is_pdf_sig=True*)

Changed in version 0.4.0: Added positional `digest_algorithm` parameter [_\(breaking change\)_](#).

Changed in version 0.5.0: Added `dry_run`, `timestamp` and `cares_meta` parameters.

Changed in version 0.9.0: Made asynchronous, grouped some parameters under `attr_settings` (breaking change)_

Format the signed attributes for a CMS signature.

Parameters

- **data_digest** – Raw digest of the data to be signed.
- **digest_algorithm** – New in version 0.4.0.
Name of the digest algorithm used to compute the digest.
- **use_pades** – Respect PAdES requirements.
- **dry_run** – New in version 0.5.0.
Flag indicating “dry run” mode. If `True`, only the approximate size of the output matters, so cryptographic operations can be replaced by placeholders.
- **attr_settings** – *PdfCMSSignedAttributes* object describing the attributes to be added.
- **timestamp** – New in version 0.5.0.
Timestamp to use when creating timestamp tokens.
- **content_type** – CMS content type of the encapsulated data. Default is *data*.

Danger: This parameter is internal API, and non-default values must not be used to produce PDF signatures.

- **is_pdf_sig** – Whether the signature being generated is for use in a PDF document.

Danger: This parameter is internal API.

Returns

An `asn1crypto.cms.CMSAttributes` object.

async async_sign(*data_digest: bytes, digest_algorithm: str, dry_run=False, use_pades=False, timestamp=None, signed_attr_settings: Optional[PdfCMSSignedAttributes] = None, is_pdf_sig=True, encap_content_info=None*) → `ContentInfo`

New in version 0.9.0.

Produce a detached CMS signature from a raw data digest.

Parameters

- **data_digest** – Digest of the actual content being signed.
- **digest_algorithm** – Digest algorithm to use. This should be the same digest method as the one used to hash the (external) content.
- **dry_run** – If `True`, the actual signing step will be replaced with a placeholder.
In a PDF signing context, this is necessary to estimate the size of the signature container before computing the actual digest of the document.
- **signed_attr_settings** – *PdfCMSSignedAttributes* object describing the attributes to be added.

- **use_pades** – Respect PAdES requirements.
- **timestamp** – `TimeStamp` used to obtain a trusted timestamp token that can be embedded into the signature container.

Note: If `dry_run` is true, the `timestamp`’s `dummy_response()` method will be called to obtain a placeholder token. Note that with a standard `HTTPTimeStamp`, this might still hit the timestamping server (in order to produce a realistic size estimate), but the dummy response will be cached.

- **is_pdf_sig** – Whether the signature being generated is for use in a PDF document.

Danger: This parameter is internal API.

- **encap_content_info** – Data to encapsulate in the CMS object.

Danger: This parameter is internal API, and must not be used to produce PDF signatures.

Returns

An `ContentInfo` object.

async `async_sign_prescribed_attributes`(*digest_algorithm: str, signed_attrs: CMSAttributes, cms_version='v1', dry_run=False, timestamp=None, encap_content_info=None*) → `ContentInfo`

New in version 0.9.0.

Start the CMS signing process with the prescribed set of signed attributes.

Parameters

- **digest_algorithm** – Digest algorithm to use. This should be the same digest method as the one used to hash the (external) content.
- **signed_attrs** – CMS attributes to sign.
- **dry_run** – If True, the actual signing step will be replaced with a placeholder.
In a PDF signing context, this is necessary to estimate the size of the signature container before computing the actual digest of the document.
- **timestamp** – `TimeStamp` used to obtain a trusted timestamp token that can be embedded into the signature container.

Note: If `dry_run` is true, the `timestamp`’s `dummy_response()` method will be called to obtain a placeholder token. Note that with a standard `HTTPTimeStamp`, this might still hit the timestamping server (in order to produce a realistic size estimate), but the dummy response will be cached.

- **cms_version** – CMS version to use.
- **encap_content_info** – Data to encapsulate in the CMS object.

Danger: This parameter is internal API, and must not be used to produce PDF signatures.

Returns

An `ContentInfo` object.

```
async async_sign_general_data(input_data: Union[IO, bytes, ContentInfo, EncapsulatedContentInfo],
                              digest_algorithm: str, detached=True, use_cades=False,
                              timestamp=None, chunk_size=4096, signed_attr_settings:
                              Optional[PdfCMSSignedAttributes] = None, max_read=None) →
                              ContentInfo
```

New in version 0.9.0.

Produce a CMS signature for an arbitrary data stream (not necessarily PDF data).

Parameters

- **input_data** – The input data to sign. This can be either a `bytes` object a file-type object, a `cms.ContentInfo` object or a `cms.EncapsulatedContentInfo` object.

Warning: `asn1crypto` mandates `cms.ContentInfo` for CMS v1 signatures. In practical terms, this means that you need to use `cms.ContentInfo` if the content type is data, and `cms.EncapsulatedContentInfo` otherwise.

Warning: We currently only support CMS v1, v3 and v4 signatures. This is only a concern if you need certificates or CRLs of type ‘other’, in which case you can change the version yourself (this will not invalidate any signatures). You’ll also need to do this if you need support for version 1 attribute certificates, or if you want to sign with `subjectKeyIdentifier` in the `sid` field.

- **digest_algorithm** – The name of the digest algorithm to use.
- **detached** – If `True`, create a CMS detached signature (i.e. an object where the encapsulated content is not embedded in the signature object itself). This is the default. If `False`, the content to be signed will be embedded as encapsulated content.
- **signed_attr_settings** – `PdfCMSSignedAttributes` object describing the attributes to be added.
- **use_cades** – Construct a CADES-style CMS object.
- **timestamp** – `PdfTimeStamp` to use to create a signature timestamp

Note: If you want to create a *content* timestamp (as opposed to a *signature* timestamp), see `CAdESSignedAttrSpec`.

- **chunk_size** – Chunk size to use when consuming input data.
- **max_read** – Maximal number of bytes to read from the input stream.

Returns

A CMS `ContentInfo` object of type `signedData`.

sign(*data_digest*: bytes, *digest_algorithm*: str, *timestamp*: Optional[datetime] = None, *dry_run*=False, *revocation_info*=None, *use_pades*=False, *timestamper*=None, *cares_signed_attr_meta*: Optional[CAdESSignedAttrSpec] = None, *encap_content_info*=None) → ContentInfo

Deprecated since version 0.9.0: Use [async_sign\(\)](#) instead. The implementation of this method will invoke [async_sign\(\)](#) using `asyncio.run()`.

Produce a detached CMS signature from a raw data digest.

Parameters

- **data_digest** – Digest of the actual content being signed.
- **digest_algorithm** – Digest algorithm to use. This should be the same digest method as the one used to hash the (external) content.
- **timestamp** – Signing time to embed into the signed attributes (will be ignored if *use_pades* is True).

Note: This timestamp value is to be interpreted as an unfounded assertion by the signer, which may or may not be good enough for your purposes.

- **dry_run** – If True, the actual signing step will be replaced with a placeholder.
In a PDF signing context, this is necessary to estimate the size of the signature container before computing the actual digest of the document.
- **revocation_info** – Revocation information to embed; this should be the output of a call to [Signer.format_revinfo\(\)](#) (ignored when *use_pades* is True).
- **use_pades** – Respect PAdES requirements.
- **timestamper** – TimeStamper used to obtain a trusted timestamp token that can be embedded into the signature container.

Note: If *dry_run* is true, the *timestamper*’s `dummy_response()` method will be called to obtain a placeholder token. Note that with a standard `HTTPTimeStamper`, this might still hit the timestamping server (in order to produce a realistic size estimate), but the dummy response will be cached.

- **cares_signed_attr_meta** – New in version 0.5.0.
Specification for CAdES-specific signed attributes.
- **encap_content_info** – Data to encapsulate in the CMS object.

Danger: This parameter is internal API, and must not be used to produce PDF signatures.

Returns

An ContentInfo object.

sign_prescribed_attributes(*digest_algorithm*: str, *signed_attrs*: CMSAttributes, *cms_version*='v1', *dry_run*=False, *timestamper*=None, *encap_content_info*=None) → ContentInfo

Deprecated since version 0.9.0: Use [async_sign_prescribed_attributes\(\)](#) instead. The implementation of this method will invoke [async_sign_prescribed_attributes\(\)](#) using `asyncio.run()`.

Start the CMS signing process with the prescribed set of signed attributes.

Parameters

- **digest_algorithm** – Digest algorithm to use. This should be the same digest method as the one used to hash the (external) content.
- **signed_attrs** – CMS attributes to sign.
- **dry_run** – If True, the actual signing step will be replaced with a placeholder.
In a PDF signing context, this is necessary to estimate the size of the signature container before computing the actual digest of the document.
- **timestamper** – `TimeStamper` used to obtain a trusted timestamp token that can be embedded into the signature container.

Note: If `dry_run` is true, the timestamper’s `dummy_response()` method will be called to obtain a placeholder token. Note that with a standard `HTTPTimeStamper`, this might still hit the timestamping server (in order to produce a realistic size estimate), but the dummy response will be cached.

- **cms_version** – CMS version to use.
- **encap_content_info** – Data to encapsulate in the CMS object.

Danger: This parameter is internal API, and must not be used to produce PDF signatures.

Returns

An `ContentInfo` object.

sign_general_data(*input_data: Union[IO, bytes, ContentInfo, EncapsulatedContentInfo], digest_algorithm: str, detached=True, timestamp: Optional[datetime] = None, use_cades=False, timestamper=None, cades_signed_attr_meta: Optional[CAdESSignedAttrSpec] = None, chunk_size=4096, max_read=None*) → `ContentInfo`

New in version 0.7.0.

Deprecated since version 0.9.0: Use `async_sign_general_data()` instead. The implementation of this method will invoke `async_sign_general_data()` using `asyncio.run()`.

Produce a CMS signature for an arbitrary data stream (not necessarily PDF data).

Parameters

- **input_data** – The input data to sign. This can be either a bytes object a file-type object, a `cms.ContentInfo` object or a `cms.EncapsulatedContentInfo` object.

Warning: `asn1crypto` mandates `cms.ContentInfo` for CMS v1 signatures. In practical terms, this means that you need to use `cms.ContentInfo` if the content type is data, and `cms.EncapsulatedContentInfo` otherwise.

Warning: We currently only support CMS v1, v3 and v4 signatures. This is only a concern if you need certificates or CRLs of type ‘other’, in which case you can change the version yourself (this will not invalidate any signatures). You’ll also need to do this if you need support for version 1 attribute certificates, or if you want to sign with `subjectKeyIdentifier` in the `sid` field.

- **digest_algorithm** – The name of the digest algorithm to use.
- **detached** – If `True`, create a CMS detached signature (i.e. an object where the encapsulated content is not embedded in the signature object itself). This is the default. If `False`, the content to be signed will be embedded as encapsulated content.
- **timestamp** – Signing time to embed into the signed attributes (will be ignored if `use_cades` is `True`).

Note: This timestamp value is to be interpreted as an unfounded assertion by the signer, which may or may not be good enough for your purposes.

- **use_cades** – Construct a CAdES-style CMS object.
- **timestamp** – [PdfTimeStamper](#) to use to create a signature timestamp

Note: If you want to create a *content* timestamp (as opposed to a *signature* timestamp), see [CAdESSignedAttrSpec](#).

- **cares_signed_attr_meta** – Specification for CAdES-specific signed attributes.
- **chunk_size** – Chunk size to use when consuming input data.
- **max_read** – Maximal number of bytes to read from the input stream.

Returns

A CMS `ContentInfo` object of type `signedData`.

```
class pyhanko.sign.signers.pdf_cms.SimpleSigner(signing_cert: Certificate, signing_key:
    PrivateKeyInfo, cert_registry: CertificateStore,
    signature_mechanism:
    Optional[SignedDigestAlgorithm] = None,
    prefer_pss=False, embed_roots=True,
    attribute_certs=None)
```

Bases: [Signer](#)

Simple signer implementation where the key material is available in local memory.

signing_key: `PrivateKeyInfo`

Private key associated with the certificate in `signing_cert`.

async async_sign_raw(*data: bytes, digest_algorithm: str, dry_run=False*) → bytes

Compute the raw cryptographic signature of the data provided, hashed using the digest algorithm provided.

Parameters

- **data** – Data to sign.
- **digest_algorithm** – Digest algorithm to use.

Warning: If `signature_mechanism` also specifies a digest, they should match.

- **dry_run** – Do not actually create a signature, but merely output placeholder bytes that would suffice to contain an actual signature.

Returns

Signature bytes.

sign_raw(*data: bytes, digest_algorithm: str*) → bytes

Synchronous raw signature implementation.

Parameters

- **data** – Data to be signed.
- **digest_algorithm** – Digest algorithm to use.

Returns

Raw signature encoded according to the conventions of the signing algorithm used.

classmethod load_pkcs12(*pfx_file, ca_chain_files=None, other_certs=None, passphrase=None, signature_mechanism=None, prefer_pss=False*)

Load certificates and key material from a PKCS#12 archive (usually .pfx or .p12 files).

Parameters

- **pfx_file** – Path to the PKCS#12 archive.
- **ca_chain_files** – Path to (PEM/DER) files containing other relevant certificates not included in the PKCS#12 file.
- **other_certs** – Other relevant certificates, specified as a list of `asn1crypto.x509`. Certificate objects.
- **passphrase** – Passphrase to decrypt the PKCS#12 archive, if required.
- **signature_mechanism** – Override the signature mechanism to use.
- **prefer_pss** – Prefer PSS signature mechanism over RSA PKCS#1 v1.5 if there's a choice.

Returns

A [*SimpleSigner*](#) object initialised with key material loaded from the PKCS#12 file provided.

classmethod load(*key_file, cert_file, ca_chain_files=None, key_passphrase=None, other_certs=None, signature_mechanism=None, prefer_pss=False*)

Load certificates and key material from PEM/DER files.

Parameters

- **key_file** – File containing the signer's private key.
- **cert_file** – File containing the signer's certificate.
- **ca_chain_files** – File containing other relevant certificates.
- **key_passphrase** – Passphrase to decrypt the private key (if required).
- **other_certs** – Other relevant certificates, specified as a list of `asn1crypto.x509`. Certificate objects.
- **signature_mechanism** – Override the signature mechanism to use.
- **prefer_pss** – Prefer PSS signature mechanism over RSA PKCS#1 v1.5 if there's a choice.

Returns

A *SimpleSigner* object initialised with key material loaded from the files provided.

```
class pyhanko.sign.signers.pdf_cms.ExternalSigner(signing_cert: Optional[Certificate], cert_registry:
Optional[CertificateStore], signature_value:
Optional[Union[bytes, int]] = None,
signature_mechanism:
Optional[SignedDigestAlgorithm] = None,
prefer_pss: bool = False, embed_roots: bool =
True)
```

Bases: *Signer*

Class to help formatting CMS objects for use with remote signing. It embeds a fixed signature value into the CMS, set at initialisation.

Intended for use with *Interrupted signing*.

Parameters

- **signing_cert** – The signer’s certificate.
- **cert_registry** – The certificate registry to use in CMS generation.
- **signature_value** – The value of the signature as a byte string, a placeholder length, or None.
- **signature_mechanism** – The signature mechanism used by the external signing service.
- **prefer_pss** – Switch to prefer PSS when producing RSA signatures, as opposed to RSA with PKCS#1 v1.5 padding.
- **embed_roots** – Whether to embed relevant root certificates into the CMS payload.

cert_registry: *CertificateStore*

Collection of certificates associated with this signer. Note that this is simply a bookkeeping tool; in particular it doesn’t care about trust.

async async_sign_raw(*data*: bytes, *digest_algorithm*: str, *dry_run*=False) → bytes

Return a fixed signature value.

signed_attr_prov_spec: Optional[*SignedAttributeProviderSpec*]

unsigned_attr_prov_spec: Optional[*UnsignedAttributeProviderSpec*]

```
class pyhanko.sign.signers.pdf_cms.PdfCMSSignedAttributes(signing_time: Optional[datetime] =
None, ades_signed_attrs:
Optional[CAAdESSignedAttrSpec] =
None, adobe_revinfo_attr:
Optional[RevocationInfoArchival] =
None)
```

Bases: CMSSignedAttributes

New in version 0.7.0.

Changed in version 0.14.0: Split off some fields into CMSSignedAttributes.

Serialisable container class describing input for various signed attributes in a CMS object for a PDF signature.

adobe_revinfo_attr: Optional[RevocationInfoArchival] = None

Adobe-style signed revocation info attribute.

```
async pyhanko.sign.signers.pdf_cms.format_attributes(attr_provs: List[CMSAttributeProvider],
                                                    other_attrs: Iterable[CMSAttributes] = (),
                                                    dry_run: bool = False) → CMSAttributes
```

Format CMS attributes obtained from attribute providers.

Parameters

- **attr_provs** – List of attribute providers.
- **other_attrs** – Other (predetermined) attributes to include.
- **dry_run** – Whether to invoke the attribute providers in dry-run mode or not.

Returns

A `cms.CMSAttributes` value.

```
async pyhanko.sign.signers.pdf_cms.format_signed_attributes(data_digest: bytes, attr_provs:
                                                            List[CMSAttributeProvider],
                                                            content_type='data', dry_run=False)
                                                            → CMSAttributes
```

Format signed attributes for a CMS `SignerInfo` value.

Parameters

- **data_digest** – The byte string to put in the `messageDigest` attribute.
- **attr_provs** – List of attribute providers to source attributes from.
- **content_type** – The content type of the data being signed (default is `data`).
- **dry_run** – Whether to invoke the attribute providers in dry-run mode or not.

Returns

A `cms.CMSAttributes` value representing the signed attributes.

```
pyhanko.sign.signers.pdf_cms.asyncify_signer(signer_cls)
```

Decorator to turn a legacy *Signer* subclass into one that works with the new async API.

```
pyhanko.sign.signers.pdf_cms.select_suitable_signing_md(key: PublicKeyInfo) → str
```

Choose a reasonable default signing message digest given the properties of (the public part of) a key.

The fallback value is `constants.DEFAULT_MD`.

Parameters

key – A `keys.PublicKeyInfo` object.

Returns

The name of a message digest algorithm.

pyhanko.sign.signers.pdf_signer module

This module implements support for PDF-specific signing functionality.

```

class pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata(field_name: ~typing.Optional[str] =
    None, md_algorithm:
    ~typing.Optional[str] = None, location:
    ~typing.Optional[str] = None, reason:
    ~typing.Optional[str] = None, name:
    ~typing.Optional[str] = None, certify:
    bool = False, subfilter: ~typing.
    Optional[~pyhanko.sign.fields.SigSeedSubFilter]
    = None, embed_validation_info: bool =
    False, use_pades_lta: bool = False,
    timestamp_field_name:
    ~typing.Optional[str] = None,
    validation_context: ~typing.
    Optional[~pyhanko_certvalidator.context.ValidationC
    = None, docmdp_permissions:
    ~pyhanko.sign.fields.MDPPerm =
    MDPPerm.FILL_FORMS,
    signer_key_usage: ~typing.Set[str] =
    <factory>, cades_signed_attr_spec:
    ~typing.
    Optional[~pyhanko.sign.ades.api.CAdESSignedAttrSp
    = None, dss_settings: ~py-
    hanko.sign.signers.pdf_signer.DSSContentSettings
    =
    DSSContentSettings(include_vri=True,
    skip_if_unneeded=True, place-
    ment=<SigDSSPlacementPreference.TOGETHER_WITH_
    3>, next_ts_settings=None),
    tight_size_estimates: bool = False,
    ac_validation_context: ~typing.
    Optional[~pyhanko_certvalidator.context.ValidationC
    = None)

```

Bases: object

Specification for a PDF signature.

field_name: Optional[str] = None

The name of the form field to contain the signature. If there is only one available signature field, the name may be inferred.

md_algorithm: Optional[str] = None

The name of the digest algorithm to use. It should be supported by *pyca/cryptography*.

If None, *select_suitable_signing_md()* will be invoked to generate a suitable default, unless a seed value dictionary happens to be available.

location: Optional[str] = None

Location of signing.

reason: Optional[str] = None

Reason for signing (textual).

name: Optional[str] = None

Name of the signer. This value is usually not necessary to set, since it should appear on the signer's certificate, but there are cases where it might be useful to specify it here (e.g. in situations where signing is delegated to a trusted third party).

certify: `bool = False`

Sign with an author (certification) signature, as opposed to an approval signature. A document can contain at most one such signature, and it must be the first one.

subfilter: `Optional[SigSeedSubFilter] = None`

Signature subfilter to use.

This should be one of `ADOBE_PKCS7_DETACHED` or `PADES`. If not specified, the value may be inferred from the signature field's seed value dictionary. Failing that, `ADOBE_PKCS7_DETACHED` is used as the default value.

embed_validation_info: `bool = False`

Flag indicating whether validation info (OCSP responses and/or CRLs) should be embedded or not. This is necessary to be able to validate signatures long after they have been made. This flag requires `validation_context` to be set.

The precise manner in which the validation info is embedded depends on the (effective) value of `subfilter`:

- With `ADOBE_PKCS7_DETACHED`, the validation information will be embedded inside the CMS object containing the signature.
- With `PADES`, the validation information will be embedded into the document security store (DSS).

use_pades_lta: `bool = False`

If `True`, the signer will append an additional document timestamp after writing the signature's validation information to the document security store (DSS). This flag is only meaningful if `subfilter` is `PADES`.

The PAdES B-LTA profile solves the long-term validation problem by adding a timestamp chain to the document after the regular signatures, which is updated with new timestamps at regular intervals. This provides an audit trail that ensures the long-term integrity of the validation information in the DSS, since OCSP responses and CRLs also have a finite lifetime.

See also `PdfTimeStamper.update_archival_timestamp_chain()`.

timestamp_field_name: `Optional[str] = None`

Name of the timestamp field created when `use_pades_lta` is `True`. If not specified, a unique name will be generated using `uuid`.

validation_context: `Optional[ValidationContext] = None`

The validation context to use when validating signatures. If provided, the signer's certificate and any timestamp certificates will be validated before signing.

This parameter is mandatory when `embed_validation_info` is `True`.

docmdp_permissions: `MDPPerm = 2`

Indicates the document modification policy that will be in force after this signature is created. Only relevant for certification signatures or signatures that apply locking.

Warning: For non-certification signatures, this is only explicitly allowed since PDF 2.0 (ISO 32000-2), so older software may not respect this setting on approval signatures.

signer_key_usage: `Set[str]`

Key usage extensions required for the signer's certificate. Defaults to `non_repudiation` only, but sometimes `digital_signature` or a combination of both may be more appropriate. See `x509.KeyUsage` for a complete list.

Only relevant if a validation context is also provided.

caades_signed_attr_spec: `Optional[CADES_SignedAttrSpec] = None`

New in version 0.5.0.

Specification for CADES-specific attributes.

dss_settings: `DSSContentSettings = DSSContentSettings(include_vri=True, skip_if_unneeded=True, placement=<SigDSSPlacementPreference.TOGETHER_WITH_NEXT_TS: 3>, next_ts_settings=None)`

New in version 0.8.0.

DSS output settings. See [DSSContentSettings](#).

tight_size_estimates: `bool = False`

New in version 0.8.0.

When estimating the size of a signature container, do not add safety margins.

Note: This should be OK if the entire CMS object is produced by pyHanko, and the signing scheme produces signatures of a fixed size. However, if the signature container includes unsigned attributes such as signature timestamps, the size of the signature is never entirely predictable.

ac_validation_context: `Optional[ValidationContext] = None`

New in version 0.11.0.

Validation context for attribute certificates

```
class pyhanko.sign.signers.pdf_signer.DSSContentSettings(include_vri: bool = True,
                                                         skip_if_unneeded: bool = True,
                                                         placement: SigDSSPlacementPreference
                                                         = SigDSSPlacementPreference
                                                         .TOGETHER_WITH_NEXT_TS,
                                                         next_ts_settings:
                                                         Optional[TimestampDSSContentSettings]
                                                         = None)
```

Bases: [GeneralDSSContentSettings](#)

New in version 0.8.0.

Settings for a DSS update with validation information for a signature.

placement: `SigDSSPlacementPreference = 3`

Preference for where to perform a DSS update with validation information for a specific signature. See [SigDSSPlacementPreference](#).

The default is [SigDSSPlacementPreference](#).TOGETHER_WITH_NEXT_TS.

next_ts_settings: `Optional[TimestampDSSContentSettings] = None`

Explicit settings for DSS updates pertaining to a document timestamp added as part of the same signing workflow, if applicable.

If None, a default will be generated based on the values of this settings object.

Note: When consuming [DSSContentSettings](#) objects, you should call [get_settings_for_ts\(\)](#) instead of relying on the value of this field.

get_settings_for_ts() → *TimestampDSSContentSettings*

Retrieve DSS update settings for document timestamps that are part of our signing workflow, if there are any.

assert_viable()

Check settings for consistency, and raise *SigningError* otherwise.

```
class pyhanko.sign.signers.pdf_signer.TimestampDSSContentSettings(include_vri: bool = True,
                                                                    skip_if_unneeded: bool =
                                                                    True, update_before_ts: bool
                                                                    = False)
```

Bases: *GeneralDSSContentSettings*

New in version 0.8.0.

Settings for a DSS update with validation information for a document timestamp.

Note: In most workflows, adding a document timestamp doesn't trigger any DSS updates beyond VRI additions, because the same TSA is used for signature timestamps and for document timestamps.

update_before_ts: `bool = False`

Perform DSS update before creating the timestamp, instead of after.

Warning: This setting can only be used if `include_vri` is `False`.

assert_viable()

Check settings for consistency, and raise *SigningError* otherwise.

```
class pyhanko.sign.signers.pdf_signer.GeneralDSSContentSettings(include_vri: bool = True,
                                                                    skip_if_unneeded: bool = True)
```

Bases: `object`

New in version 0.8.0.

Settings that govern DSS creation and updating in general.

include_vri: `bool = True`

Flag to control whether to create and update entries in the VRI dictionary. The default is to always update the VRI dictionary.

Note: The VRI dictionary is a relic of the past that is effectively deprecated in the current PAdES standards, and most modern validators don't rely on it being there.

That said, there's no real harm in creating these entries, other than that it occasionally forces DSS updates where none would otherwise be necessary, and that it prevents the DSS from being updated prior to signing (as opposed to after signing).

skip_if_unneeded: `bool = True`

Do not perform a write if updating the DSS would not add any new information.

Note: This setting is only used if the DSS update would happen in its own revision.

```
class pyhanko.sign.signers.pdf_signer.SigDSSPlacementPreference(value)
```

Bases: Enum

New in version 0.8.0.

Preference for where to perform a DSS update with validation information for a specific signature.

TOGETHER_WITH_SIGNATURE = 1

Update the DSS in the revision that contains the signature. Doing so can be useful to create a PAdES-B-LT signature in a single revision. Such signatures can be processed by a validator that isn't capable of incremental update analysis.

Warning: This setting can only be used if `include_vri` is `False`.

SEPARATE_REVISION = 2

Always perform the DSS update in a separate revision, after the signature, but before any timestamps are added.

Note: This is the old default behaviour.

TOGETHER_WITH_NEXT_TS = 3

If the signing workflow includes a document timestamp after the signature, update the DSS in the same revision as the timestamp. In the absence of document timestamps, this is equivalent to [SEPARATE_REVISION](#).

Warning: This option controls the addition of validation info for the signature and its associated signature timestamp, not the validation info for the document timestamp itself. See [DSSContentSettings.next_ts_settings](#).

In most practical situations, the distinction is only relevant in interrupted signing workflows (see [Interrupted signing](#)), where the lifecycle of the validation context is out of pyHanko's hands.

```
class pyhanko.sign.signers.pdf_signer.PdfTimeStamper(timestamper: TimeStamper, field_name:
Optional[str] = None, invis_settings:
InvisSigSettings =
InvisSigSettings(set_print_flag=True,
set_hidden_flag=False,
box_out_of_bounds=False),
readable_field_name: str = 'Timestamp')
```

Bases: object

Class to encapsulate the process of appending document timestamps to PDF files.

property field_name: str

Retrieve or generate the field name for the signature field to contain the document timestamp.

Returns

The field name, as a (Python) string.

```
timestamp_pdf(pdf_out: IncrementalPdfFileWriter, md_algorithm, validation_context=None,
bytes_reserved=None, validation_paths=None, timestamp: Optional[TimeStamper] =
None, *, in_place=False, output=None, dss_settings: TimestampDSSContentSettings =
TimestampDSSContentSettings(include_vri=True, skip_if_unneeded=True,
update_before_ts=False), chunk_size=4096, tight_size_estimates: bool = False)
```

Changed in version 0.9.0: Wrapper around `async_timestamp_pdf()`.

Timestamp the contents of `pdf_out`. Note that `pdf_out` should not be written to after this operation.

Parameters

- **pdf_out** – An *IncrementalPdfFileWriter*.
- **md_algorithm** – The hash algorithm to use when computing message digests.
- **validation_context** – The `pyhanko_certvalidator.ValidationContext` against which the TSA response should be validated. This validation context will also be used to update the DSS.
- **bytes_reserved** – Bytes to reserve for the CMS object in the PDF file. If not specified, make an estimate based on a dummy signature.

Warning: Since the CMS object is written to the output file as a hexadecimal string, you should request **twice** the (estimated) number of bytes in the DER-encoded version of the CMS object.

- **validation_paths** – If the validation path(s) for the TSA’s certificate are already known, you can pass them using this parameter to avoid having to run the validation logic again.
- **timestamper** – Override the default *TimeStamper* associated with this *PdfTimeStamper*.
- **output** – Write the output to the specified output stream. If `None`, write to a new `BytesIO` object. Default is `None`.
- **in_place** – Sign the original input stream in-place. This parameter overrides `output`.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support `memoryview`.
- **dss_settings** – DSS output settings. See *TimestampDSSContentSettings*.
- **tight_size_estimates** – When estimating the size of a document timestamp container, do not add safety margins.

Note: External TSAs cannot be relied upon to always produce the exact same output length, which makes this option risky to use.

Returns

The output stream containing the signed output.

```
async async_timestamp_pdf(pdf_out: IncrementalPdfFileWriter, md_algorithm,
                           validation_context=None, bytes_reserved=None, validation_paths=None,
                           timestamper: Optional[TimeStamper] = None, *, in_place=False,
                           output=None, dss_settings: TimestampDSSContentSettings =
                               TimestampDSSContentSettings(include_vri=True, skip_if_unneeded=True,
                               update_before_ts=False), chunk_size=4096, tight_size_estimates: bool =
                               False, embed_roots: bool = True)
```

New in version 0.9.0.

Timestamp the contents of `pdf_out`. Note that `pdf_out` should not be written to after this operation.

Parameters

- **pdf_out** – An *IncrementalPdfFileWriter*.
- **md_algorithm** – The hash algorithm to use when computing message digests.
- **validation_context** – The `pyhanko_certvalidator.ValidationContext` against which the TSA response should be validated. This validation context will also be used to update the DSS.
- **bytes_reserved** – Bytes to reserve for the CMS object in the PDF file. If not specified, make an estimate based on a dummy signature.

Warning: Since the CMS object is written to the output file as a hexadecimal string, you should request **twice** the (estimated) number of bytes in the DER-encoded version of the CMS object.

- **validation_paths** – If the validation path(s) for the TSA’s certificate are already known, you can pass them using this parameter to avoid having to run the validation logic again.
- **timestamp** – Override the default *TimeStamper* associated with this *PdfTimeStamper*.
- **output** – Write the output to the specified output stream. If `None`, write to a new `BytesIO` object. Default is `None`.
- **in_place** – Sign the original input stream in-place. This parameter overrides `output`.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support `memoryview`.
- **dss_settings** – DSS output settings. See *TimestampDSSContentSettings*.
- **tight_size_estimates** – When estimating the size of a document timestamp container, do not add safety margins.

Note: External TSAs cannot be relied upon to always produce the exact same output length, which makes this option risky to use.

- **embed_roots** – Option that controls whether the root certificate of each validation path should be embedded into the DSS. The default is `True`.

Note: Trust roots are configured by the validator, so embedding them typically does nothing in a typical validation process. Therefore they can be safely omitted in most cases. Nonetheless, embedding the roots can be useful for documentation purposes.

Returns

The output stream containing the signed output.

update_archival_timestamp_chain(*reader*: *PdfFileReader*, *validation_context*, *in_place*=`True`, *output*=`None`, *chunk_size*=4096, *default_md_algorithm*='sha256')

Changed in version 0.9.0: Wrapper around `async_update_archival_timestamp_chain()`.

Validate the last timestamp in the timestamp chain on a PDF file, and write an updated version to an output stream.

Parameters

- **reader** – A *PdfReader* encapsulating the input file.

- **validation_context** – `pyhanko_certvalidator.ValidationContext` object to validate the last timestamp.
- **output** – Write the output to the specified output stream. If `None`, write to a new `BytesIO` object. Default is `None`.
- **in_place** – Sign the original input stream in-place. This parameter overrides `output`.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support `memoryview`.
- **default_md_algorithm** – Message digest to use if there are no preceding timestamps in the file.

Returns

The output stream containing the signed output.

```
async async_update_archival_timestamp_chain(reader: PdfFileReader, validation_context,
                                             in_place=True, output=None, chunk_size=4096,
                                             default_md_algorithm='sha256', embed_roots: bool
                                             = True)
```

New in version 0.9.0.

Validate the last timestamp in the timestamp chain on a PDF file, and write an updated version to an output stream.

Parameters

- **reader** – A `PdfReader` encapsulating the input file.
- **validation_context** – `pyhanko_certvalidator.ValidationContext` object to validate the last timestamp.
- **output** – Write the output to the specified output stream. If `None`, write to a new `BytesIO` object. Default is `None`.
- **in_place** – Sign the original input stream in-place. This parameter overrides `output`.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support `memoryview`.
- **default_md_algorithm** – Message digest to use if there are no preceding timestamps in the file.
- **embed_roots** – Option that controls whether the root certificate of each validation path should be embedded into the DSS. The default is `True`.

Note: Trust roots are configured by the validator, so embedding them typically does nothing in a typical validation process. Therefore they can be safely omitted in most cases. Nonetheless, embedding the roots can be useful for documentation purposes.

Returns

The output stream containing the signed output.

```
class pyhanko.sign.signers.pdf_signer.PdfSigner(signature_meta: PdfSignatureMetadata, signer:
                                                  Signer, *, timestamp: Optional[TimeStamper] =
                                                  None, stamp_style: Optional[BaseStampStyle] =
                                                  None, new_field_spec: Optional[SigFieldSpec] =
                                                  None)
```

Bases: `object`

Class to handle PDF signatures in general.

Parameters

- **signature_meta** – The specification of the signature to add.
- **signer** – *Signer* object to use to produce the signature object.
- **timestamper** – *TimeStamper* object to use to produce any time stamp tokens that might be required.
- **stamp_style** – Stamp style specification to determine the visible style of the signature, typically an object of type *TextStampStyle* or *QRStampStyle*. Defaults to constants. `DEFAULT_SIGNING_STAMP_STYLE`.
- **new_field_spec** – If a new field is to be created, this parameter allows the caller to specify the field’s properties in the form of a *SigFieldSpec*. This parameter is only meaningful if `existing_fields_only` is `False`.

property default_md_for_signer: `Optional[str]`

Name of the default message digest algorithm for this signer, if there is one. This method will try the `md_algorithm` attribute on the signer’s `signature_meta`, or try to retrieve the digest algorithm associated with the underlying *Signer*.

Returns

The name of the message digest algorithm, or `None`.

init_signing_session(*pdf_out*: *BasePdfFileWriter*, *existing_fields_only*=`False`) → *PdfSigningSession*

Initialise a signing session with this *PdfSigner* for a specified PDF file writer.

This step in the signing process handles all field-level operations prior to signing: it creates the target form field if necessary, and makes sure the seed value dictionary gets processed.

See also *digest_doc_for_signing()* and *sign_pdf()*.

Parameters

- **pdf_out** – The writer containing the PDF file to be signed.
- **existing_fields_only** – If `True`, never create a new empty signature field to contain the signature. If `False`, a new field may be created if no field matching *field_name* exists.

Returns

A *PdfSigningSession* object modelling the signing session in its post-setup stage.

digest_doc_for_signing(*pdf_out*: *BasePdfFileWriter*, *existing_fields_only*=`False`, *bytes_reserved*=`None`, **, appearance_text_params*=`None`, *in_place*=`False`, *output*=`None`, *chunk_size*=4096) → `Tuple[PreparedByteRangeDigest, PdfTBSDocument, IO]`

Deprecated since version 0.9.0: Use *async_digest_doc_for_signing()* instead.

Set up all stages of the signing process up to and including the point where the signature placeholder is allocated, and the document’s `/ByteRange` digest is computed.

See *sign_pdf()* for a less granular, more high-level approach.

Note: This method is useful in remote signing scenarios, where you might want to free up resources while waiting for the remote signer to respond. The *PreparedByteRangeDigest* object returned allows you to keep track of the required state to fill the signature container at some later point in time.

Parameters

- **pdf_out** – A PDF file writer (usually an *IncrementalPdfFileWriter*) containing the data to sign.
- **existing_fields_only** – If True, never create a new empty signature field to contain the signature. If False, a new field may be created if no field matching *field_name* exists.
- **bytes_reserved** – Bytes to reserve for the CMS object in the PDF file. If not specified, make an estimate based on a dummy signature.

Warning: Since the CMS object is written to the output file as a hexadecimal string, you should request **twice** the (estimated) number of bytes in the DER-encoded version of the CMS object.

- **appearance_text_params** – Dictionary with text parameters that will be passed to the signature appearance constructor (if applicable).
- **output** – Write the output to the specified output stream. If None, write to a new BytesIO object. Default is None.
- **in_place** – Sign the original input stream in-place. This parameter overrides output.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support *memoryview*.

Returns

A tuple containing a *PreparedByteRangeDigest* object, a *PdfTBSDocument* object and an output handle to which the document in its current state has been written.

```
async async_digest_doc_for_signing(pdf_out: BasePdfFileWriter, existing_fields_only=False,
                                   bytes_reserved=None, *, appearance_text_params=None,
                                   in_place=False, output=None, chunk_size=4096) →
                                   Tuple[PreparedByteRangeDigest, PdfTBSDocument, IO]
```

New in version 0.9.0.

Set up all stages of the signing process up to and including the point where the signature placeholder is allocated, and the document's /ByteRange digest is computed.

See *sign_pdf()* for a less granular, more high-level approach.

Note: This method is useful in remote signing scenarios, where you might want to free up resources while waiting for the remote signer to respond. The *PreparedByteRangeDigest* object returned allows you to keep track of the required state to fill the signature container at some later point in time.

Parameters

- **pdf_out** – A PDF file writer (usually an *IncrementalPdfFileWriter*) containing the data to sign.
- **existing_fields_only** – If True, never create a new empty signature field to contain the signature. If False, a new field may be created if no field matching *field_name* exists.
- **bytes_reserved** – Bytes to reserve for the CMS object in the PDF file. If not specified, make an estimate based on a dummy signature.

Warning: Since the CMS object is written to the output file as a hexadecimal string, you should request **twice** the (estimated) number of bytes in the DER-encoded version of the CMS object.

- **appearance_text_params** – Dictionary with text parameters that will be passed to the signature appearance constructor (if applicable).
- **output** – Write the output to the specified output stream. If `None`, write to a new `BytesIO` object. Default is `None`.
- **in_place** – Sign the original input stream in-place. This parameter overrides `output`.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support `memoryview`.

Returns

A tuple containing a [PreparedByteRangeDigest](#) object, a [PdfTBSDocument](#) object and an output handle to which the document in its current state has been written.

sign_pdf(pdf_out: [BasePdfFileWriter](#), existing_fields_only=False, bytes_reserved=None, *, appearance_text_params=None, in_place=False, output=None, chunk_size=4096)

Changed in version 0.9.0: Wrapper around [async_sign_pdf\(\)](#).

Sign a PDF file using the provided output writer.

Parameters

- **pdf_out** – A PDF file writer (usually an [IncrementalPdfFileWriter](#)) containing the data to sign.
- **existing_fields_only** – If `True`, never create a new empty signature field to contain the signature. If `False`, a new field may be created if no field matching [field_name](#) exists.
- **bytes_reserved** – Bytes to reserve for the CMS object in the PDF file. If not specified, make an estimate based on a dummy signature.
- **appearance_text_params** – Dictionary with text parameters that will be passed to the signature appearance constructor (if applicable).
- **output** – Write the output to the specified output stream. If `None`, write to a new `BytesIO` object. Default is `None`.
- **in_place** – Sign the original input stream in-place. This parameter overrides `output`.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support `memoryview`.

Returns

The output stream containing the signed data.

async async_sign_pdf(pdf_out: [BasePdfFileWriter](#), existing_fields_only=False, bytes_reserved=None, *, appearance_text_params=None, in_place=False, output=None, chunk_size=4096)

New in version 0.9.0.

Sign a PDF file using the provided output writer.

Parameters

- **pdf_out** – A PDF file writer (usually an [IncrementalPdfFileWriter](#)) containing the data to sign.

- **existing_fields_only** – If True, never create a new empty signature field to contain the signature. If False, a new field may be created if no field matching *field_name* exists.
- **bytes_reserved** – Bytes to reserve for the CMS object in the PDF file. If not specified, make an estimate based on a dummy signature.
- **appearance_text_params** – Dictionary with text parameters that will be passed to the signature appearance constructor (if applicable).
- **output** – Write the output to the specified output stream. If None, write to a new BytesIO object. Default is None.
- **in_place** – Sign the original input stream in-place. This parameter overrides output.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support memoryview.

Returns

The output stream containing the signed data.

```
class pyhanko.sign.signers.pdf_signer.PdfSigningSession(pdf_signer: PdfSigner, pdf_out:
    BasePdfFileWriter, cms_writer, sig_field,
    md_algorithm: str, timestamper:
    TimeStamper, subfilter: SigSeedSubFilter,
    system_time: Optional[datetime] = None,
    sv_spec: Optional[SigSeedValueSpec] =
    None)
```

Bases: object

New in version 0.7.0.

Class modelling a PDF signing session in its initial state.

The `__init__` method is internal API, get an instance using *PdfSigner.init_signing_session()*.

```
async perform_presign_validation(pdf_out: Optional[BasePdfFileWriter] = None) →
    Optional[PreSignValidationStatus]
```

Perform certificate validation checks for the signer's certificate, including any necessary revocation checks.

This function will also attempt to validate & collect revocation information for the relevant TSA (by requesting a dummy timestamp).

Parameters

pdf_out – Current PDF writer. Technically optional; only used to look for the end of the timestamp chain in the previous revision when producing a PAdES-LTA signature in a document that is already signed (to ensure that the timestamp chain is uninterrupted).

Returns

A *PreSignValidationStatus* object, or None if there is no validation context available.

```
async estimate_signature_container_size(validation_info: PreSignValidationStatus, tight=False)
```

```
prepare_tbs_document(validation_info: PreSignValidationStatus, bytes_reserved,
    appearance_text_params=None) → PdfTBSDocument
```

Set up the signature appearance (if necessary) and signature dictionary in the PDF file, to put the document in its final pre-signing state.

Parameters

- **validation_info** – Validation information collected prior to signing.
- **bytes_reserved** – Bytes to reserve for the signature container.

- **appearance_text_params** – Optional text parameters for the signature appearance content.

Returns

A *PdfTBSDocument* describing the document in its final pre-signing state.

```
class pyhanko.sign.signers.pdf_signer.PdfTBSDocument(cms_writer: str, signer: Signer, md_algorithm: str,
use_pades: bool, timestamp: Optional[TimeStamper] = None,
post_sign_instructions: Optional[PostSignInstructions] = None,
validation_context: Optional[ValidationContext] = None)
```

Bases: object

New in version 0.7.0.

A PDF document in its final pre-signing state.

The `__init__` method is internal API, get an instance using *PdfSigningSession.prepare_tbs_document()*. Alternatively, use *resume_signing()* or *finish_signing()* to continue a previously interrupted signing process without instantiating a new *PdfTBSDocument* object.

```
digest_tbs_document(*, output: Optional[IO] = None, in_place: bool = False, chunk_size=4096) →
    Tuple[PreparedByteRangeDigest, IO]
```

Write the document to an output stream and compute the digest, while keeping track of the (future) location of the signature contents in the output stream.

The digest can then be passed to the next part of the signing pipeline.

Warning: This method can only be called once.

Parameters

- **output** – Write the output to the specified output stream. If *None*, write to a new *BytesIO* object. Default is *None*.
- **in_place** – Sign the original input stream in-place. This parameter overrides *output*.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support *memoryview*.

Returns

A tuple containing a *PreparedByteRangeDigest* and the output stream to which the output was written.

```
async perform_signature(document_digest: bytes, pdf_cms_signed_attrs: PdfCMSSignedAttributes) →
    PdfPostSignatureDocument
```

Perform the relevant cryptographic signing operations on the document digest, and write the resulting CMS object to the appropriate location in the output stream.

Warning: This method can only be called once, and must be invoked after *digest_tbs_document()*.

Parameters

- **document_digest** – Digest of the document, as computed over the relevant */ByteRange*.

- **pdf_cms_signed_attrs** – Description of the signed attributes to include.

Returns

A *PdfPostSignatureDocument* object.

```
classmethod resume_signing(output: IO, prepared_digest: PreparedByteRangeDigest, signature_cms:  
                           Union[bytes, ContentInfo], post_sign_instr:  
                           Optional[PostSignInstructions] = None, validation_context:  
                           Optional[ValidationContext] = None) → PdfPostSignatureDocument
```

Resume signing after obtaining a CMS object from an external source.

This is a class method; it doesn't require a *PdfTBSDocument* instance. Contrast with *perform_signature()*.

Parameters

- **output** – Output stream housing the document in its final pre-signing state. This stream must at least be writable and seekable, and also readable if post-signature processing is required.
- **prepared_digest** – The prepared digest returned by a prior call to *digest_tbs_document()*.
- **signature_cms** – CMS object to embed in the signature dictionary.
- **post_sign_instr** – Instructions for post-signing processing (DSS updates and document timestamps).
- **validation_context** – Validation context to use in post-signing operations. This is mainly intended for TSA certificate validation, but it can also contain additional validation data to embed in the DSS.

Returns

A *PdfPostSignatureDocument*.

```
classmethod finish_signing(output: IO, prepared_digest: PreparedByteRangeDigest, signature_cms:  
                           Union[bytes, ContentInfo], post_sign_instr:  
                           Optional[PostSignInstructions] = None, validation_context:  
                           Optional[ValidationContext] = None, chunk_size=4096)
```

Finish signing after obtaining a CMS object from an external source, and perform any required post-signature processing.

This is a class method; it doesn't require a *PdfTBSDocument* instance. Contrast with *perform_signature()*.

Parameters

- **output** – Output stream housing the document in its final pre-signing state.
- **prepared_digest** – The prepared digest returned by a prior call to *digest_tbs_document()*.
- **signature_cms** – CMS object to embed in the signature dictionary.
- **post_sign_instr** – Instructions for post-signing processing (DSS updates and document timestamps).
- **validation_context** – Validation context to use in post-signing operations. This is mainly intended for TSA certificate validation, but it can also contain additional validation data to embed in the DSS.

- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support `memoryview`.

async classmethod async_finish_signing(*output: IO, prepared_digest: PreparedByteRangeDigest, signature_cms: Union[bytes, ContentInfo], post_sign_instr: Optional[PostSignInstructions] = None, validation_context: Optional[ValidationContext] = None, chunk_size=4096*)

Finish signing after obtaining a CMS object from an external source, and perform any required post-signature processing.

This is a class method; it doesn't require a `PdfTBSDocument` instance. Contrast with `perform_signature()`.

Parameters

- **output** – Output stream housing the document in its final pre-signing state.
- **prepared_digest** – The prepared digest returned by a prior call to `digest_tbs_document()`.
- **signature_cms** – CMS object to embed in the signature dictionary.
- **post_sign_instr** – Instructions for post-signing processing (DSS updates and document timestamps).
- **validation_context** – Validation context to use in post-signing operations. This is mainly intended for TSA certificate validation, but it can also contain additional validation data to embed in the DSS.
- **chunk_size** – Size of the internal buffer (in bytes) used to feed data to the message digest function if the input stream does not support `memoryview`.

```
class pyhanko.sign.signers.pdf_signer.PdfPostSignatureDocument(sig_contents: bytes,
                                                                post_sign_instr:
                                                                Optional[PostSignInstructions] =
                                                                None, validation_context:
                                                                Optional[ValidationContext] =
                                                                None)
```

Bases: `object`

New in version 0.7.0.

Represents the final phase of the PDF signing process

async post_signature_processing(*output: IO, chunk_size=4096*)

Handle DSS updates and LTA timestamps, if applicable.

Parameters

- **output** – I/O buffer containing the signed document. Must support reading, writing and seeking.
- **chunk_size** – Chunk size to use for I/O operations that do not support the buffer protocol.

```
class pyhanko.sign.signers.pdf_signer.PreSignValidationStatus(signer_path: ValidationPath,  
                                                             validation_paths:  
                                                             List[ValidationPath],  
                                                             ts_validation_paths:  
                                                             Optional[List[ValidationPath]] =  
                                                             None, adobe_revinfo_attr:  
                                                             Optional[RevocationInfoArchival]  
                                                             = None, ocsp_to_embed:  
                                                             Optional[List[OCSPResponse]] =  
                                                             None, crls_to_embed:  
                                                             Optional[List[CertificateList]] =  
                                                             None, ac_validation_paths:  
                                                             Optional[List[ValidationPath]] =  
                                                             None)
```

Bases: object

New in version 0.7.0.

Container for validation data collected prior to creating a signature, e.g. for later inclusion in a document's DSS, or as a signed attribute on the signature.

signer_path: ValidationPath

Validation path for the signer's certificate.

validation_paths: List[ValidationPath]

List of other relevant validation paths.

ts_validation_paths: Optional[List[ValidationPath]] = None

List of validation paths relevant for embedded timestamps.

adobe_revinfo_attr: Optional[RevocationInfoArchival] = None

Preformatted revocation info attribute to include, if requested by the settings.

ocsp_to_embed: List[OCSPResponse] = None

List of OCSP responses collected so far.

crls_to_embed: List[CertificateList] = None

List of CRLS collected so far.

ac_validation_paths: Optional[List[ValidationPath]] = None

List of validation paths relevant for embedded attribute certificates.

```

class pyhanko.sign.signers.pdf_signer.PostSignInstructions(validation_info: ~py-
    hanko.sign.signers.pdf_signer.PreSignValidationStatus,
    timestamp: ~typing.Optional[~pyhanko.sign.timestamps.api.TimeStamper]
    = None, timestamp_md_algorithm: ~typing.Optional[str] = None,
    timestamp_field_name: ~typing.Optional[str] = None,
    dss_settings: ~pyhanko.sign.signers.pdf_signer.DSSContentSettings
    = DSSContentSettings(include_vri=True, skip_if_unneeded=True, place-
    ment=<SigDSSPlacementPreference.TOGETHER_WITH_NEXT_TS>, next_ts_settings=None),
    tight_size_estimates: bool = False, embed_roots: bool = True,
    file_credential: ~typing.Optional[~pyhanko.pdf_utils.crypt.cred_ser.Serialised]
    = None)

```

Bases: `object`

New in version 0.7.0.

Container class housing instructions for incremental updates to the document after the signature has been put in place. Necessary for PAdES-LT and PAdES-LTA workflows.

validation_info: `PreSignValidationStatus`

Validation information to embed in the DSS (if not already present).

timestamp: `Optional[TimeStamper] = None`

Timestamp to use for produce document timestamps. If `None`, no timestamp will be added.

timestamp_md_algorithm: `Optional[str] = None`

Digest algorithm to use when producing timestamps. Defaults to `DEFAULT_MD`.

timestamp_field_name: `Optional[str] = None`

Name of the timestamp field to use. If not specified, a field name will be generated.

dss_settings: `DSSContentSettings = DSSContentSettings(include_vri=True, skip_if_unneeded=True, placement=<SigDSSPlacementPreference.TOGETHER_WITH_NEXT_TS: 3>, next_ts_settings=None)`

New in version 0.8.0.

Settings to fine-tune DSS generation.

tight_size_estimates: `bool = False`

New in version 0.8.0.

When estimating the size of a document timestamp container, do not add safety margins.

Note: External TSAs cannot be relied upon to always produce the exact same output length, which makes this option risky to use.

embed_roots: `bool = True`

New in version 0.9.0.

Option that controls whether the root certificate of each validation path should be embedded into the DSS. The default is `True`.

Note: Trust roots are configured by the validator, so embedding them typically does nothing in a typical validation process. Therefore they can be safely omitted in most cases. Nonetheless, embedding the roots can be useful for documentation purposes.

Note: This setting is not part of `DSSContentSettings` because its value is taken from the corresponding property on the `Signer` involved, not from the initial configuration.

file_credential: `Optional[SerialisedCredential] = None`

New in version 0.13.0.

Serialised file credential, to update encrypted files.

pyhanko.sign.timestamps package

pyhanko.sign.timestamps.api module

Module to handle the timestamping functionality in pyHanko.

Many PDF signature profiles require trusted timestamp tokens. The tools in this module allow pyHanko to obtain such tokens from [RFC 3161](#)-compliant time stamping authorities.

class `pyhanko.sign.timestamps.api.TimeStamper(include_nonce=True)`

Bases: `object`

Changed in version 0.9.0: Made API more asyncio-friendly _(breaking change)_

Class to make [RFC 3161](#) timestamp requests.

request_cms(*message_digest*, *md_algorithm*)

Format the body of an [RFC 3161](#) request as a CMS object. Subclasses with more specific needs may want to override this.

Parameters

- **message_digest** – Message digest to which the timestamp will apply.
- **md_algorithm** – Message digest algorithm to use.

Note: As per [RFC 8933](#), `md_algorithm` should also be the algorithm used to compute `message_digest`.

Returns

An `asn1crypto.tsp.TimestampReq` object.

async validation_paths(*validation_context*)

Produce validation paths for the certificates gathered by this `TimeStamper`.

This is internal API.

Parameters

validation_context – The validation context to apply.

Returns

An asynchronous generator of validation paths.

async async_dummy_response(*md_algorithm*) → ContentInfo

Return a dummy response for use in CMS object size estimation.

For every new *md_algorithm* passed in, this method will call the `timestamp()` method exactly once, with a dummy digest. The resulting object will be cached and reused for future invocations of `dummy_response()` with the same *md_algorithm* value.

Parameters

md_algorithm – Message digest algorithm to use.

Returns

A timestamp token, encoded as an `asn1crypto.cms.ContentInfo` object.

async async_request_tsa_response(*req: TimeStampReq*) → TimeStampResp

Submit the specified timestamp request to the server.

Parameters

req – Request body to submit.

Returns

A timestamp response from the server.

Raises

IOError – Raised in case of an I/O issue in the communication with the timestamping server.

async async_timestamp(*message_digest, md_algorithm*) → ContentInfo

Request a timestamp for the given message digest.

Parameters

- **message_digest** – Message digest to which the timestamp will apply.
- **md_algorithm** – Message digest algorithm to use.

Note: As per [RFC 8933](#), *md_algorithm* should also be the algorithm used to compute *message_digest*.

Returns

A timestamp token, encoded as an `asn1crypto.cms.ContentInfo` object.

Raises

- **IOError** – Raised in case of an I/O issue in the communication with the timestamping server.
- **TimestampRequestError** – Raised if the timestamp server did not return a success response, or if the server's response is invalid.

pyhanko.sign.timestamps.aiohttp_client module

```
class pyhanko.sign.timestamps.aiohttp_client.AIOHttpTimeStamper(url, session:
                                                                    Union[ClientSession,
                                                                    LazySession], https=False,
                                                                    timeout=5, headers=None, auth:
                                                                    Optional[BasicAuth] = None)
```

Bases: *TimeStamper*

async **async_request_headers()** → dict

Format the HTTP request headers. Subclasses can override this to perform their own header generation logic.

Returns

Header dictionary.

async **get_session()** → ClientSession

async **async_timestamp(message_digest, md_algorithm)** → ContentInfo

Request a timestamp for the given message digest.

Parameters

- **message_digest** – Message digest to which the timestamp will apply.
- **md_algorithm** – Message digest algorithm to use.

Note: As per [RFC 8933](#), `md_algorithm` should also be the algorithm used to compute `message_digest`.

Returns

A timestamp token, encoded as an `asn1crypto.cms.ContentInfo` object.

Raises

- **IOError** – Raised in case of an I/O issue in the communication with the timestamping server.
- **TimestampRequestError** – Raised if the timestamp server did not return a success response, or if the server's response is invalid.

async **async_request_tsa_response(req: TimeStampReq)** → TimeStampResp

Submit the specified timestamp request to the server.

Parameters

req – Request body to submit.

Returns

A timestamp response from the server.

Raises

IOError – Raised in case of an I/O issue in the communication with the timestamping server.

pyhanko.sign.timestamps.requests_client module

```
class pyhanko.sign.timestamps.requests_client.HTTPTimeStamper(url, https=False, timeout=5,
                                                             auth=None, headers=None)
```

Bases: *TimeStamper*

Standard HTTP-based timestamp client.

request_headers() → dict

Format the HTTP request headers.

Returns

Header dictionary.

async async_request_tsa_response(req: *TimeStampReq*) → *TimeStampResp*

Submit the specified timestamp request to the server.

Parameters

req – Request body to submit.

Returns

A timestamp response from the server.

Raises

IOError – Raised in case of an I/O issue in the communication with the timestamping server.

pyhanko.sign.timestamps.dummy_client module

```
class pyhanko.sign.timestamps.dummy_client.DummyTimeStamper(tsa_cert: Certificate, tsa_key:
                                                             PrivateKeyInfo, certs_to_embed:
                                                             Optional[CertificateStore] = None,
                                                             fixed_dt: Optional[datetime] = None,
                                                             include_nonce=True,
                                                             override_md=None)
```

Bases: *TimeStamper*

Timestamping client that acts as its own TSA. It accepts all requests and signs them using the certificate provided. Used for testing purposes.

request_tsa_response(req: *TimeStampReq*) → *TimeStampResp*

async async_request_tsa_response(req: *TimeStampReq*) → *TimeStampResp*

Submit the specified timestamp request to the server.

Parameters

req – Request body to submit.

Returns

A timestamp response from the server.

Raises

IOError – Raised in case of an I/O issue in the communication with the timestamping server.

pyhanko.sign.timestamps.common_utils module

exception pyhanko.sign.timestamps.common_utils.TimestampRequestError

Bases: OSError

Raised when an error occurs while requesting a timestamp.

pyhanko.sign.timestamps.common_utils.get_nonce()

pyhanko.sign.timestamps.common_utils.extract_ts_certs(*ts_token*, *store*: CertificateStore)

pyhanko.sign.timestamps.common_utils.dummy_digest(*md_algorithm*: str) → bytes

pyhanko.sign.timestamps.common_utils.handle_tsp_response(*response*: TimestampResp, *nonce*: Optional[bytes]) → ContentInfo

pyhanko.sign.timestamps.common_utils.set_tsp_headers(*headers*: dict)

pyhanko.sign.validation module

pyhanko.sign.validation.dss module

class pyhanko.sign.validation.dss.VRI(*certs*: set = <factory>, *ocsps*: set = <factory>, *crls*: set = <factory>)

Bases: object

VRI dictionary as defined in PAdES / ISO 32000-2. These dictionaries collect data that may be relevant for the validation of a specific signature.

Note: The data are stored as PDF indirect objects, not asn1crypto values. In particular, values are tied to a specific PDF handler.

certs: set

Relevant certificates.

ocsps: set

Relevant OCSF responses.

crls: set

Relevant CRLs.

as_pdf_object() → DictionaryObject

Returns

A PDF dictionary representing this VRI entry.

class pyhanko.sign.validation.dss.DocumentSecurityStore(*writer*: BasePdfFileWriter, *certs*=None, *ocsps*=None, *crls*=None, *vri_entries*=None, *backing_pdf_object*=None)

Bases: object

Representation of a DSS in Python.

property modified

static `sig_content_identifier(contents)` → *NameObject*

Hash the contents of a signature object to get the corresponding VRI identifier.

This is internal API.

Parameters

contents – Signature contents.

Returns

A name object to put into the DSS.

register_vri(*identifier*, *, *certs*=(), *ocsp*s=(), *crl*s=())

Register validation information for a set of signing certificates associated with a particular signature.

Parameters

- **identifier** – Identifier of the signature object (see *sig_content_identifier*). If *None*, only embed the data into the DSS without associating it with any VRI.
- **certs** – Certificates to add.
- **ocsp**s – OCSP responses to add.
- **crl**s – CRLs to add.

as_pdf_object()

Convert the *DocumentSecurityStore* object to a python dictionary. This method also handles DSS updates.

Returns

A PDF object representing this DSS.

load_certs()

Return a generator that parses and yields all certificates in the DSS.

Returns

A generator yielding *Certificate* objects.

as_validation_context(*validation_context_kwargs*, *include_revinfo*=*True*) → *ValidationContext*

Construct a validation context from the data in this DSS.

Parameters

- **validation_context_kwargs** – Extra kwargs to pass to the `__init__` function.
- **include_revinfo** – If *False*, revocation info is skipped.

Returns

A validation context preloaded with information from this DSS.

classmethod `read_dss(handler: PdfHandler)` → *DocumentSecurityStore*

Read a DSS record from a file and add the data to a validation context.

Parameters

handler – PDF handler from which to read the DSS.

Returns

A *DocumentSecurityStore* object describing the current state of the DSS.

classmethod `supply_dss_in_writer(pdf_out: BasePdfFileWriter, sig_contents, *, certs=None, ocsp=None, crls=None, paths=None, validation_context=None, embed_roots: bool = True)` → *DocumentSecurityStore*

Add or update a DSS, and optionally associate the new information with a VRI entry tied to a signature object.

You can either specify the CMS objects to include directly, or pass them in as output from *pyhanko_certvalidator*.

Parameters

- **pdf_out** – PDF writer to write to.
- **sig_contents** – Contents of the new signature (used to compute the VRI hash), as as a hexadecimal string, including any padding. If *None*, the information will not be added to any VRI dictionary.
- **certs** – Certificates to include in the VRI entry.
- **ocsps** – OCSP responses to include in the VRI entry.
- **crls** – CRLs to include in the VRI entry.
- **paths** – Validation paths that have been established, and need to be added to the DSS.
- **validation_context** – Validation context from which to draw OCSP responses and CRLs.
- **embed_roots** – New in version 0.9.0.

Option that controls whether the root certificate of each validation path should be embedded into the DSS. The default is *True*.

Note: Trust roots are configured by the validator, so embedding them typically does nothing in a typical validation process. Therefore they can be safely omitted in most cases. Nonetheless, embedding the roots can be useful for documentation purposes.

Warning: This only applies to paths, not the *certs* parameter.

Returns

a *DocumentSecurityStore* object containing both the new and existing contents of the DSS (if any).

classmethod **add_dss**(*output_stream*, *sig_contents*, *, *certs*=*None*, *ocsps*=*None*, *crls*=*None*, *paths*=*None*, *validation_context*=*None*, *force_write*: *bool* = *False*, *embed_roots*: *bool* = *True*, *file_credential*: *Optional*[*SerialisedCredential*] = *None*)

Wrapper around *supply_dss_in_writer()*.

The result is applied to the output stream as an incremental update.

Parameters

- **output_stream** – Output stream to write to.
- **sig_contents** – Contents of the new signature (used to compute the VRI hash), as as a hexadecimal string, including any padding. If *None*, the information will not be added to any VRI dictionary.
- **certs** – Certificates to include in the VRI entry.
- **ocsps** – OCSP responses to include in the VRI entry.

- **crls** – CRLs to include in the VRI entry.
- **paths** – Validation paths that have been established, and need to be added to the DSS.
- **force_write** – Force a write even if the DSS doesn't have any new content.
- **validation_context** – Validation context from which to draw OCSP responses and CRLs.
- **embed_roots** – New in version 0.9.0.

Option that controls whether the root certificate of each validation path should be embedded into the DSS. The default is `True`.

Note: Trust roots are configured by the validator, so embedding them typically does nothing in a typical validation process. Therefore they can be safely omitted in most cases. Nonetheless, embedding the roots can be useful for documentation purposes.

Warning: This only applies to paths, not the `certs` parameter.

- **file_credential** – New in version 0.13.0.

Serialised file credential, to update encrypted files.

```

async pyhanko.sign.validation.dss.async_add_validation_info(embedded_sig:
    EmbeddedPdfSignature,
    validation_context:
    ValidationContext,
    skip_timestamp=False,
    add_vri_entry=True, in_place=False,
    output=None, force_write=False,
    chunk_size=4096, embed_roots: bool
    = True)

```

Add validation info (CRLs, OCSP responses, extra certificates) for a signature to the DSS of a document in an incremental update. This is a wrapper around `collect_validation_info()`.

Parameters

- **embedded_sig** – The signature for which the revocation information needs to be collected.
- **validation_context** – The validation context to use.
- **skip_timestamp** – If `True`, do not attempt to validate the timestamp attached to the signature, if one is present.
- **add_vri_entry** – Add a /VRI entry for this signature to the document security store. Default is `True`.
- **output** – Write the output to the specified output stream. If `None`, write to a new `BytesIO` object. Default is `None`.
- **in_place** – Sign the original input stream in-place. This parameter overrides `output`.
- **chunk_size** – Chunk size parameter to use when copying output to a new stream (irrelevant if `in_place` is `True`).
- **force_write** – Force a new revision to be written, even if not necessary (i.e. when all data in the validation context is already present in the DSS).

- **embed_roots** – Option that controls whether the root certificate of each validation path should be embedded into the DSS. The default is True.

Note: Trust roots are configured by the validator, so embedding them typically does nothing in a typical validation process. Therefore they can be safely omitted in most cases. Nonetheless, embedding the roots can be useful for documentation purposes.

Returns

The (file-like) output object to which the result was written.

```
async pyhanko.sign.validation.dss.collect_validation_info(embedded_sig: EmbeddedPdfSignature,
                                                         validation_context: ValidationContext,
                                                         skip_timestamp=False)
```

Query revocation info for a PDF signature using a validation context, and store the results in a validation context.

This works by validating the signer's certificate against the provided validation context, which causes revocation info to be cached for later retrieval.

Warning: This function does *not* actually validate the signature, but merely checks the signer certificate's chain of trust.

Parameters

- **embedded_sig** – Embedded PDF signature to operate on.
- **validation_context** – Validation context to use.
- **skip_timestamp** – If the signature has a time stamp token attached to it, also collect revocation information for the timestamp.

Returns

A list of validation paths.

pyhanko.sign.validation.errors module

```
exception pyhanko.sign.validation.errors.SignatureValidationError(failure_message,
                                                                  ades_subindication:
                                                                  Optional[AdESSubIndic] =
                                                                  None)
```

Bases: [ValueErrorWithMessage](#)

Error validating a signature.

property ades_status: [Optional\[AdESStatus\]](#)

```
exception pyhanko.sign.validation.errors.WeakHashAlgorithmError(*args, **kwargs)
```

Bases: [SignatureValidationError](#)

```
exception pyhanko.sign.validation.errors.ValidationInfoReadingError(failure_message)
```

Bases: [ValueErrorWithMessage](#)

Error reading validation info.

```
exception pyhanko.sign.validation.errors.NoDSSFoundError
```

Bases: [ValidationInfoReadingError](#)

exception `pyhanko.sign.validation.errors.SigSeedValueValidationError`(*failure_message*,
ades_subindication:
Optional[`AdESSubIndic`]
= `None`)

Bases: `SignatureValidationError`

Error validating a signature's seed value constraints.

exception `pyhanko.sign.validation.errors.CMSAlgorithmProtectionError`(*failure_message*)

Bases: `ValueErrorWithMessage`

Error related to CMS algorithm protection checks.

pyhanko.sign.validation.generic_cms module

`pyhanko.sign.validation.generic_cms.validate_sig_integrity`(*signer_info*: `SignerInfo`, *cert*:
`Certificate`, *expected_content_type*: `str`,
actual_digest: `bytes`,
weak_hash_algorithms=`frozenset({'md2',
'md5', 'sha1'})`) → `Tuple[bool, bool]`

Validate the integrity of a signature for a particular `signerInfo` object inside a CMS signed data container.

Warning: This function does not do any trust checks, and is considered “dangerous” API because it is easy to misuse.

Parameters

- **signer_info** – A `cms.SignerInfo` object.
- **cert** – The signer's certificate.

Note: This function will not attempt to extract certificates from the signed data.

- **expected_content_type** – The expected value for the content type attribute (as a Python string, see `cms.ContentType`).
- **actual_digest** – The actual digest to be matched to the message digest attribute.
- **weak_hash_algorithms** – List, tuple or set of weak hashing algorithms.

Returns

A tuple of two booleans. The first indicates whether the provided digest matches the value in the signed attributes. The second indicates whether the signature of the digest is valid.

```
async pyhanko.sign.validation.generic_cms.async_validate_cms_signature(signed_data:
    ~asn1crypto.cms.SignedData,
    status_cls: ~typing.Type[~pyhanko.sign.validation.generic_
    = <class 'py-
    hanko.sign.validation.status.SignatureStat
    raw_digest:
    ~typing.Optional[bytes]
    = None,
    validation_context:
    ~typ-
    ing.Optional[~pyhanko.certvalidator.cont
    = None, status_kwargs:
    ~typing.Optional[dict]
    = None,
    key_usage_settings:
    ~typ-
    ing.Optional[~pyhanko.sign.validation.set
    = None, en-
    cap_data_invalid=False)
```

Validate a CMS signature (i.e. a `SignedData` object).

Parameters

- **signed_data** – The `asn1crypto.cms.SignedData` object to validate.
- **status_cls** – Status class to use for the validation result.
- **raw_digest** – Raw digest, computed from context.
- **validation_context** – Validation context to validate the signer’s certificate.
- **status_kwargs** – Other keyword arguments to pass to the `status_class` when reporting validation results.
- **key_usage_settings** – A [KeyUsageConstraints](#) object specifying which key usages must or must not be present in the signer’s certificate.
- **encap_data_invalid** – If `True`, the encapsulated data inside the CMS is invalid, but the remaining validation logic still has to be run (e.g. a timestamp token, which requires validation of the embedded message imprint).

This option is considered internal API, the semantics of which may change without notice in the future.

Returns

A [SignatureStatus](#) object (or an instance of a proper subclass)

```
async pyhanko.sign.validation.generic_cms.collect_timing_info(signer_info: SignerInfo,
    ts_validation_context:
    ValidationContext, raw_digest:
    bytes)
```

Collect and validate timing information in a `SignerInfo` value. This includes the `signingTime` attribute, content timestamp information and signature timestamp information.

Parameters

- **signer_info** – A `SignerInfo` value.
- **ts_validation_context** – The timestamp validation context to validate against.

- **raw_digest** – The raw external message digest bytes (only relevant for the validation of the content timestamp token, if there is one)

```

async pyhanko.sign.validation.generic_cms.validate_tst_signed_data(tst_signed_data:
                                                                    SignedData,
                                                                    validation_context:
                                                                    ValidationContext,
                                                                    expected_tst_imprint: bytes)

```

Validate the SignedData of a time stamp token.

Parameters

- **tst_signed_data** – The SignedData value to validate; must encapsulate a TSTInfo value.
- **validation_context** – The validation context to validate against.
- **expected_tst_imprint** – The expected message imprint value that should be contained in the encapsulated TSTInfo.

Returns

Keyword arguments for a TimeStampSignatureStatus.

```

async pyhanko.sign.validation.generic_cms.async_validate_detached_cms(input_data: Union[bytes,
                                                                                          IO, ContentInfo, Encap-
                                                                                          sulatedContentInfo],
                                                                    signed_data:
                                                                    SignedData,
                                                                    signer_validation_context:
                                                                    Optional[ValidationContext]
                                                                    = None,
                                                                    ts_validation_context:
                                                                    Optional[ValidationContext]
                                                                    = None,
                                                                    ac_validation_context:
                                                                    Optional[ValidationContext]
                                                                    = None,
                                                                    key_usage_settings: Op-
                                                                    tional[KeyUsageConstraints]
                                                                    = None,
                                                                    chunk_size=4096,
                                                                    max_read=None) →
                                                                    StandardCMSSigna-
                                                                    tureStatus

```

Validate a detached CMS signature.

Parameters

- **input_data** – The input data to sign. This can be either a bytes object, a file-like object or a cms.ContentInfo / cms.EncapsulatedContentInfo object.

If a CMS content info object is passed in, the *content* field will be extracted.

- **signed_data** – The cms.SignedData object containing the signature to verify.

- **signer_validation_context** – Validation context to use to verify the signer certificate’s trust.
- **ts_validation_context** – Validation context to use to verify the TSA certificate’s trust, if a timestamp token is present. By default, the same validation context as that of the signer is used.
- **ac_validation_context** – Validation context to use to validate attribute certificates. If not supplied, no AC validation will be performed.

Note: [RFC 5755](#) requires attribute authority trust roots to be specified explicitly; hence why there’s no default.

- **key_usage_settings** – Key usage parameters for the signer.
- **chunk_size** – Chunk size to use when consuming input data.
- **max_read** – Maximal number of bytes to read from the input stream.

Returns

A description of the signature’s status.

```
async pyhanko.sign.validation.generic_cms.cms_basic_validation(signed_data:
    ~asn1crypto.cms.SignedData,
    status_cls: ~typing.Type[~pyhanko.sign.validation.generic_cms.Status],
    = <class 'pyhanko.sign.validation.status.SignatureStatus'>,
    raw_digest: ~typing.Optional[bytes] = None,
    validation_context: ~typing.Optional[~pyhanko.certvalidator.context.Validator] = None,
    status_kwargs: ~typing.Optional[dict] = None,
    key_usage_settings: ~typing.Optional[~pyhanko.sign.validation.settings.KeyUsageSettings] = None,
    encap_data_invalid=False)
```

Perform basic validation of CMS and PKCS#7 signatures in isolation (i.e. integrity and trust checks).

Internal API.

```
pyhanko.sign.validation.generic_cms.compute_signature_tst_digest(signer_info: SignerInfo) →
    Optional[bytes]
```

Compute the digest of the signature according to the message imprint algorithm information in a signature timestamp token.

Internal API.

Parameters

signer_info – A SignerInfo value.

Returns

The computed digest, or None if there is no signature timestamp.

```
pyhanko.sign.validation.generic_cms.extract_tst_data(signer_info, signed=False) →
    Optional[SignedData]
```

Extract signed data associated with a timestamp token.

Internal API.

Parameters

- **signer_info** – A `SignerInfo` value.
- **signed** – If `True`, look for a content timestamp (among the signed attributes), else look for a signature timestamp (among the unsigned attributes).

Returns

The `SignedData` value found, or `None`.

`pyhanko.sign.validation.generic_cms.extract_self_reported_ts(signer_info: SignerInfo) → Optional[datetime]`

Extract self-reported timestamp (from the `signingTime` attribute)

Internal API.

Parameters

signer_info – A `SignerInfo` value.

Returns

The value of the `signingTime` attribute as a `datetime`, or `None`.

`pyhanko.sign.validation.generic_cms.extract_certs_for_validation(signed_data: SignedData) → SignedDataCerts`

Extract certificates from a CMS signed data object for validation purposes, identifying the signer's certificate in accordance with ETSI EN 319 102-1, 5.2.3.4.

Parameters

signed_data – The CMS payload.

Returns

The extracted certificates.

`async pyhanko.sign.validation.generic_cms.collect_signer_attr_status(sd_attr_certificates: Iterable[AttributeCertificateV2], signer_cert: Certificate, validation_context: Optional[ValidationContext], sd_signed_attrs: CMSAttributes)`

`pyhanko.sign.validation.generic_cms.validate_algorithm_protection(attrs: CMSAttributes, claimed_digest_algorithm_obj: DigestAlgorithm, claimed_signature_algorithm_obj: SignedDigestAlgorithm)`

Internal API to validate the CMS algorithm protection attribute defined in [RFC 6211](#), if present.

Parameters

- **attrs** – A CMS attribute list.
- **claimed_digest_algorithm_obj** – The claimed (i.e. unprotected) digest algorithm value.
- **claimed_signature_algorithm_obj** – The claimed (i.e. unprotected) signature algorithm value.

Raises

- **errors.CMSStructuralError** – if multiple CMS protection attributes are present
- **errors.CMSAlgorithmProtectionError** – if a mismatch is detected

pyhanko.sign.validation.ltv module

class pyhanko.sign.validation.ltv.**RevocationInfoValidationType**(*value*)

Bases: Enum

Indicates a validation profile to use when validating revocation info.

ADOBE_STYLE = 'adobe'

Retrieve validation information from the CMS object, using Adobe's revocation info archival attribute.

PADES_LT = 'pades'

Retrieve validation information from the DSS, and require the signature's embedded timestamp to still be valid.

PADES_LTA = 'pades-lta'

Retrieve validation information from the DSS, but read & validate the chain of document timestamps leading up to the signature to establish the integrity of the validation information at the time of signing.

classmethod **as_tuple**()

pyhanko.sign.validation.ltv.**apply_adobe_revocation_info**(*signer_info: SignerInfo*,
validation_context_kwargs=None) →
ValidationContext

Read Adobe-style revocation information from a CMS object, and load it into a validation context.

Parameters

- **signer_info** – Signer info CMS object.
- **validation_context_kwargs** – Extra kwargs to pass to the `__init__` function.

Returns

A validation context preloaded with the relevant revocation information.

pyhanko.sign.validation.ltv.**retrieve_adobe_revocation_info**(*signer_info: SignerInfo*)

Retrieve Adobe-style revocation information from a `SignerInfo` value, if present.

Internal API.

Parameters

signer_info – A `SignerInfo` value.

Returns

A tuple of two (potentially empty) lists, containing OCSP responses and CRLs, respectively.

pyhanko.sign.validation.ltv.**get_timestamp_chain**(*reader: PdfFileReader*) →
Iterator[*EmbeddedPdfSignature*]

Get the document timestamp chain of the associated reader, ordered from new to old.

Parameters

reader – A *PdfFileReader*.

Returns

An iterable of *EmbeddedPdfSignature* objects representing document timestamps.

```

async pyhanko.sign.validation.ltv.async_validate_pdf_ltv_signature(embedded_sig:
    EmbeddedPdfSignature,
    validation_type: Revocation-
    InfoValidationType,
    validation_context_kwargs=None,
    bootstrap_validation_context:
    Optional[ValidationContext]
    = None,
    ac_validation_context_kwargs=None,
    force_revinfo=False,
    diff_policy:
    Optional[DiffPolicy] =
    None, key_usage_settings:
    Optional[KeyUsageConstraints]
    = None, skip_diff: bool =
    False) → PdfSignatureStatus

```

New in version 0.9.0.

Validate a PDF LTV signature according to a particular profile.

Parameters

- **embedded_sig** – Embedded signature to evaluate.
- **validation_type** – Validation profile to use.
- **validation_context_kwargs** – Keyword args to instantiate `pyhanko_certvalidator.ValidationContext` objects needed over the course of the validation.
- **ac_validation_context_kwargs** – Keyword arguments for the validation context to use to validate attribute certificates. If not supplied, no AC validation will be performed.

Note: [RFC 5755](#) requires attribute authority trust roots to be specified explicitly; hence why there's no default.

- **bootstrap_validation_context** – Validation context used to validate the current timestamp.
- **force_revinfo** – Require all certificates encountered to have some form of live revocation checking provisions.
- **diff_policy** – Policy to evaluate potential incremental updates that were appended to the signed revision of the document. Defaults to `DEFAULT_DIFF_POLICY`.
- **key_usage_settings** – A `KeyUsageConstraints` object specifying which key usages must or must not be present in the signer's certificate.
- **skip_diff** – If True, skip the difference analysis step entirely.

Returns

The status of the signature.

```

async pyhanko.sign.validation.ltv.establish_timestamp_trust(tst_signed_data: SignedData,
    validation_context:
    ValidationContext,
    expected_tst_imprint: bytes)

```

Wrapper around `validate_tst_signed_data()` for use when analysing timestamps for the purpose of establishing a timestamp chain. Its main purpose is throwing/logging an error if validation fails, since that amounts to lack of trust in the purported validation time.

This is internal API.

Parameters

- **tst_signed_data** – The SignedData value to validate; must encapsulate a TSTInfo value.
- **validation_context** – The validation context to apply to the timestamp.
- **expected_tst_imprint** – The expected message imprint for the TSTInfo value.

Returns

A *TimestampSignatureStatus* if validation is successful.

Raises

SignatureValidationError if validation fails.

pyhanko.sign.validation.pdf_embedded module

```
class pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature(reader: PdfFileReader, sig_field: DictionaryObject, fq_name: str)
```

Bases: object

Class modelling a signature embedded in a PDF document.

sig_object: *DictionaryObject*

The signature dictionary.

sig_field: *DictionaryObject*

The field dictionary of the form field containing the signature.

signed_data: SignedData

CMS signed data in the signature.

property embedded_attr_certs: List[AttributeCertificateV2]

Embedded attribute certificates.

property other_embedded_certs: List[Certificate]

Embedded X.509 certificates, excluding than that of the signer.

property signer_cert: Certificate

Certificate of the signer.

property sig_object_type: *NameObject*

Returns the type of the embedded signature object. For ordinary signatures, this will be `/Sig`. In the case of a document timestamp, `/DocTimeStamp` is returned.

Returns

A PDF name object describing the type of signature.

property field_name

Returns

Name of the signature field.

property self_reported_timestamp: `Optional[datetime]`

Returns

The signing time as reported by the signer, if embedded in the signature's signed attributes.

property attached_timestamp_data: `Optional[SignedData]`

Returns

The signed data component of the timestamp token embedded in this signature, if present.

compute_integrity_info(*diff_policy=None, skip_diff=False*)

Compute the various integrity indicators of this signature.

Parameters

- **diff_policy** – Policy to evaluate potential incremental updates that were appended to the signed revision of the document. Defaults to `DEFAULT_DIFF_POLICY`.
- **skip_diff** – If True, skip the difference analysis step entirely.

summarise_integrity_info() → dict

Compile the integrity information for this signature into a dictionary that can later be passed to [PdfSignatureStatus](#) as kwargs.

This method is only available after calling [EmbeddedPdfSignature.compute_integrity_info\(\)](#).

property seed_value_spec: `Optional[SigSeedValueSpec]`

property docmdp_level: `Optional[MDPPerm]`

Returns

The document modification policy required by this signature or its Lock dictionary.

Warning: This does not take into account the DocMDP requirements of earlier signatures (if present).

The specification forbids signing with a more lenient DocMDP than the one currently in force, so this should not happen in a compliant document. That being said, any potential violations will still invalidate the earlier signature with the stricter DocMDP policy.

property fieldmdp: `Optional[FieldMDPSpec]`

Returns

Read the field locking policy of this signature, if applicable. See also [FieldMDPSpec](#).

compute_digest() → bytes

Compute the /ByteRange digest of this signature. The result will be cached.

Returns

The digest value.

compute_tst_digest() → Optional[bytes]

Compute the digest of the signature needed to validate its timestamp token (if present).

Warning: This computation is only relevant for timestamp tokens embedded inside a regular signature. If the signature in question is a document timestamp (where the entire signature object is a timestamp token), this method does not apply.

Returns

The digest value, or `None` if there is no timestamp token.

evaluate_signature_coverage() → *SignatureCoverageLevel*

Internal method used to evaluate the coverage level of a signature.

Returns

The coverage level of the signature.

evaluate_modifications(diff_policy: DiffPolicy) → Union[*DiffResult*, *SuspiciousModification*]

Internal method used to evaluate the modification level of a signature.

class pyhanko.sign.validation.pdf_embedded.**DocMDPInfo**(permission, author_sig)

Bases: tuple

Encodes certification information for a signed document, consisting of a reference to the author signature, together with the associated DocMDP policy.

property author_sig

Alias for field number 1

property permission

Alias for field number 0

pyhanko.sign.validation.pdf_embedded.**read_certification_data**(reader: PdfFileReader) → Optional[*DocMDPInfo*]

Read the certification information for a PDF document, if present.

Parameters

reader – Reader representing the input document.

Returns

A *DocMDPInfo* object containing the relevant data, or `None`.

async pyhanko.sign.validation.pdf_embedded.**async_validate_pdf_signature**(embedded_sig: EmbeddedPdfSignature, signer_validation_context: Optional[ValidationContext] = None, ts_validation_context: Optional[ValidationContext] = None, ac_validation_context: Optional[ValidationContext] = None, diff_policy: Optional[DiffPolicy] = None, key_usage_settings: Optional[KeyUsageConstraints] = None, skip_diff: bool = False) → PdfSignatureStatus

New in version 0.9.0.

Validate a PDF signature.

Parameters

- **embedded_sig** – Embedded signature to evaluate.
- **signer_validation_context** – Validation context to use to validate the signature’s chain of trust.
- **ts_validation_context** – Validation context to use to validate the timestamp’s chain of trust (defaults to `signer_validation_context`).
- **ac_validation_context** – Validation context to use to validate attribute certificates. If not supplied, no AC validation will be performed.

Note: [RFC 5755](#) requires attribute authority trust roots to be specified explicitly; hence why there’s no default.

- **diff_policy** – Policy to evaluate potential incremental updates that were appended to the signed revision of the document. Defaults to `DEFAULT_DIFF_POLICY`.
- **key_usage_settings** – A [KeyUsageConstraints](#) object specifying which key usages must or must not be present in the signer’s certificate.
- **skip_diff** – If True, skip the difference analysis step entirely.

Returns

The status of the PDF signature in question.

async `pyhanko.sign.validation.pdf_embedded.async_validate_pdf_timestamp`(*embedded_sig*: [EmbeddedPdfSignature](#), *validation_context*: *Optional*[[ValidationContext](#)] = *None*, *diff_policy*: *Optional*[[DiffPolicy](#)] = *None*, *skip_diff*: *bool* = *False*) → [DocumentTimestampStatus](#)

New in version 0.9.0.

Validate a PDF document timestamp.

Parameters

- **embedded_sig** – Embedded signature to evaluate.
- **validation_context** – Validation context to use to validate the timestamp’s chain of trust.
- **diff_policy** – Policy to evaluate potential incremental updates that were appended to the signed revision of the document. Defaults to `DEFAULT_DIFF_POLICY`.
- **skip_diff** – If True, skip the difference analysis step entirely.

Returns

The status of the PDF timestamp in question.

`pyhanko.sign.validation.pdf_embedded.report_seed_value_validation`(*embedded_sig*: [EmbeddedPdfSignature](#),
validation_path: [ValidationPath](#),
timestamp_found: *bool*)

Internal API function to enforce seed value constraints (if present) and report on the result(s).

Parameters

- **embedded_sig** – The embedded signature.
- **validation_path** – The validation path for the signer’s certificate.
- **timestamp_found** – Flag indicating whether a valid timestamp was found or not.

Returns

A `status_kwargs` dict.

`pyhanko.sign.validation.pdf_embedded.extract_contents`(*sig_object*: [DictionaryObject](#)) → bytes

Internal function to extract the (DER-encoded) signature bytes from a PDF signature dictionary.

Parameters

sig_object – A signature dictionary.

Returns

The extracted contents as a byte string.

pyhanko.sign.validation.settings module

class `pyhanko.sign.validation.settings.KeyUsageConstraints`(*key_usage*: *Optional[Set[str]] = None*,
key_usage_forbidden:
Optional[Set[str]] = None,
extd_key_usage: *Optional[Set[str]] = None*,
explicit_extd_key_usage_required:
bool = True, *match_all_key_usages*:
bool = False)

Bases: [ConfigurableMixin](#)

Convenience class to pass around key usage requirements and validate them. Intended to be flexible enough to handle both PKIX and ISO 32000 certificate seed value constraint semantics.

Changed in version 0.6.0: Bring extended key usage semantics in line with [RFC 5280](#) (PKIX).

key_usage: `Set[str] = None`

All or some (depending on `match_all_key_usage`) of these key usage extensions must be present in the signer’s certificate. If not set or empty, all key usages are considered acceptable.

key_usage_forbidden: `Set[str] = None`

These key usages must not be present in the signer’s certificate.

Note: This behaviour is undefined in [RFC 5280](#) (PKIX), but included for compatibility with certificate seed value settings in ISO 32000.

extd_key_usage: `Set[str] = None`

List of acceptable key purposes that can appear in an extended key usage extension in the signer's certificate, if such an extension is at all present. If not set, all extended key usages are considered acceptable.

If no extended key usage extension is present, or if the `anyExtendedKeyUsage` key purpose ID is present, the resulting behaviour depends on [explicit_extd_key_usage_required](#).

Setting this option to the empty set (as opposed to `None`) effectively bans all (presumably unrecognised) extended key usages.

Warning: Note the difference in behaviour with [key_usage](#) for empty sets of valid usages.

Warning: Contrary to what some CAs seem to believe, the criticality of the extended key usage extension is irrelevant here. Even a non-critical EKU extension **must** be enforced according to [RFC 5280](#) § 4.2.1.12.

In practice, many certificate authorities issue non-repudiation certs that can also be used for TLS authentication by only including the TLS client authentication key purpose ID in the EKU extension. Interpreted strictly, [RFC 5280](#) bans such certificates from being used to sign documents, and pyHanko will enforce these semantics if [extd_key_usage](#) is not `None`.

explicit_extd_key_usage_required: `bool = True`

New in version 0.6.0.

Require an extended key usage extension with the right key usages to be present if [extd_key_usage](#) is non-empty.

If this flag is `True`, at least one key purpose in [extd_key_usage](#) must appear in the certificate's extended key usage, and `anyExtendedKeyUsage` will be ignored.

match_all_key_usages: `bool = False`

New in version 0.6.0.

If `True`, all key usages indicated in [key_usage](#) must be present in the certificate. If `False`, one match suffices.

If [key_usage](#) is empty or `None`, this option has no effect.

validate(*cert: Certificate*)

classmethod process_entries(*config_dict*)

Hook method that can modify the configuration dictionary to overwrite or tweak some of their values (e.g. to convert string parameters into more complex Python objects)

Subclasses that override this method should call `super().process_entries()`, and leave keys that they do not recognise untouched.

Parameters

config_dict – A dictionary containing configuration values.

Raises

[ConfigurationError](#) – when there is a problem processing a relevant entry.

pyhanko.sign.validation.status module

```
class pyhanko.sign.validation.status.SignatureStatus(intact: bool, valid: bool, trust_problem_indic:  
Optional[AdESSubIndic], signing_cert:  
Certificate, pkcs7_signature_mechanism: str,  
md_algorithm: str, validation_path:  
ValidationPath)
```

Bases: `object`

Class describing the validity of a (general) CMS signature.

intact: bool

Reports whether the signature is *intact*, i.e. whether the hash of the message content (which may or may not be embedded inside the CMS object itself) matches the hash value that was signed.

valid: bool

Reports whether the signature is *valid*, i.e. whether the hash's signature actually validates.

trust_problem_indic: Optional[AdESSubIndic]

If not `None`, provides the AdES subindication indication what went wrong when validating the signer's certificate.

signing_cert: Certificate

Contains the certificate of the signer, as embedded in the CMS object.

pkcs7_signature_mechanism: str

CMS signature mechanism used.

md_algorithm: str

Message digest algorithm used.

validation_path: ValidationPath

Validation path providing a valid chain of trust from the signer's certificate to a trusted root certificate.

key_usage: ClassVar[Set[str]] = {'non_repudiation'}

Class property indicating which key usages are accepted on the signer's certificate. The default is `non_repudiation` only.

extd_key_usage: ClassVar[Optional[Set[str]]] = None

Class property indicating which extended key usage key purposes are accepted to be present on the signer's certificate.

See `KeyUsageConstraints.extd_key_usage`.

summary_fields()

property revoked: bool

Reports whether the signer's certificate has been revoked or not. If this field is `True`, then obviously `trusted` will be `False`.

property trusted: bool

Reports whether the signer's certificate is trusted w.r.t. the currently relevant validation context and key usage requirements.

summary(delimiter=',')

Provide a textual but machine-parsable summary of the validity.

```
async classmethod validate_cert_usage(validator: CertificateValidator, key_usage_settings:  
                                       Optional[KeyUsageConstraints] = None)
```

```
class pyhanko.sign.validation.status.TimestampSignatureStatus(intact: bool, valid: bool,  
                                                             trust_problem_indic:  
                                                             Optional[AdESSubIndic],  
                                                             signing_cert: Certificate,  
                                                             pkcs7_signature_mechanism: str,  
                                                             md_algorithm: str,  
                                                             validation_path: ValidationPath,  
                                                             timestamp: datetime)
```

Bases: [SignatureStatus](#)

Signature status class used when validating timestamp tokens.

```
key_usage: ClassVar[Set[str]] = {}
```

There are no (non-extended) key usage requirements for TSA certificates.

```
extd_key_usage: ClassVar[Optional[Set[str]]] = {'time_stamping'}
```

TSA certificates must have the `time_stamping` extended key usage extension (OID 1.3.6.1.5.5.7.3.8).

```
timestamp: datetime
```

Value of the timestamp token as a datetime object.

```
describe_timestamp_trust()
```

```
class pyhanko.sign.validation.status.X509AttributeInfo(attr_type: AttCertAttributeType, attr_values:  
                                                       Iterable[Asn1Value])
```

Bases: object

Info on an X.509 attribute.

```
attr_type: AttCertAttributeType
```

The certified attribute's type.

```
attr_values: Iterable[Asn1Value]
```

The certified attribute's values.

```
class pyhanko.sign.validation.status.CertifiedAttributeInfo(attr_type: AttCertAttributeType,  
                                                            attr_values: Iterable[Asn1Value],  
                                                            validation_results:  
                                                            Iterable[ACValidationResult])
```

Bases: [X509AttributeInfo](#)

Info on a certified attribute, including AC validation results.

```
validation_results: Iterable[ACValidationResult]
```

The validation details for the attribute in question (possibly several if values for the same attribute were sourced from several different ACs).

```
class pyhanko.sign.validation.status.ClaimedAttributes
```

Bases: object

Container class for extracted information on attributes asserted by a signer without an attribute certificate.

```
classmethod from_iterable(attrs: Iterable[AttCertAttribute], parse_error_fatal=False)
```

class pyhanko.sign.validation.status.**CertifiedAttributes**

Bases: object

Container class for extracted attribute certificate information.

classmethod **from_results**(*results: Iterable[ACValidationResult]*, *parse_error_fatal=False*)

class pyhanko.sign.validation.status.**CAdESSignerAttributeAssertions**(*claimed_attrs: ClaimedAttributes*, *certified_attrs: Optional[CertifiedAttributes] = None*, *ac_validation_errs: Optional[Collection[Union[PathValidationError, PathBuildingError]]] = None*, *unknown_attrs_present: bool = False*)

Bases: object

Value type describing information extracted (and, if relevant, validated) from a signer-attrs-v2 signed attribute.

claimed_attrs: *ClaimedAttributes*

Attributes claimed by the signer without additional justification. May be empty.

certified_attrs: *Optional[CertifiedAttributes] = None*

Attributes claimed by the signer using an attribute certificate.

This field will only be populated if an attribute certificate validation context is available, otherwise its value will be None, even if there are no attribute certificates present.

ac_validation_errs: *Optional[Collection[Union[PathValidationError, PathBuildingError]]] = None*

Attribute certificate validation errors.

This field will only be populated if an attribute certificate validation context is available, otherwise its value will be None, even if there are no attribute certificates present.

unknown_attrs_present: *bool = False*

Records if the signer-attrs-v2 attribute contained certificate types or signed assertions that could not be processed.

This does not affect the validation process by default, but will trigger a warning.

property **valid**

```

class pyhanko.sign.validation.status.StandardCMSSignatureStatus(intact: bool, valid: bool,
                                                                trust_problem_indic:
                                                                Optional[AdESSubIndic],
                                                                signing_cert: Certificate,
                                                                pkcs7_signature_mechanism:
                                                                str, md_algorithm: str,
                                                                validation_path: ValidationPath,
                                                                signer_reported_dt:
                                                                Optional[datetime] = None,
                                                                timestamp_validity: Op-
                                                                tional[TimestampSignatureStatus]
                                                                = None,
                                                                content_timestamp_validity: Op-
                                                                tional[TimestampSignatureStatus]
                                                                = None, ac_attrs:
                                                                Optional[CertifiedAttributes] =
                                                                None, ac_validation_errs: Op-
                                                                tional[Collection[Union[PathValidationError,
                                                                PathBuildingError]]] = None,
                                                                cades_signer_attrs: Op-
                                                                tional[CADES_SignerAttributeAssertions]
                                                                = None)

```

Bases: [SignatureStatus](#)

Status of a standard “end-entity” CMS signature, potentially with timing information embedded inside.

signer_reported_dt: `Optional[datetime] = None`

Signer-reported signing time, if present in the signature.

Generally speaking, this timestamp should not be taken as fact.

timestamp_validity: `Optional[TimestampSignatureStatus] = None`

Validation status of the signature timestamp token embedded in this signature, if present.

content_timestamp_validity: `Optional[TimestampSignatureStatus] = None`

Validation status of the content timestamp token embedded in this signature, if present.

ac_attrs: `Optional[CertifiedAttributes] = None`

Certified attributes sourced from valid attribute certificates embedded into the SignedData’s certificates field and the CADES-style signer-attrs-v2 attribute (if present).

Will be None if no validation context for attribute certificate validation was provided.

Note: There is a semantic difference between attribute certificates extracted from the certificates field and those extracted from the signer-attrs-v2 attribute. In the former case, the ACs are not covered by the signature. However, a CADES-style signer-attrs-v2 attribute is signed, so the signer is expected to have explicitly `_acknowledged_` all attributes, in the AC. See also [cades_signer_attrs](#).

ac_validation_errs: `Optional[Collection[Union[PathValidationError, PathBuildingError]]] = None`

Errors encountered while validating attribute certificates embedded into the SignedData’s certificates field and the CADES-style signer-attrs-v2 attribute (if present).

Will be None if no validation context for attribute certificate validation was provided.

cadessigner_attrs: `Optional[CadESSignerAttributeAssertions] = None`

Information extracted and validated from the signed `signer-attrs-v2` attribute defined in CAdES.

property bottom_line: `bool`

Formulates a general judgment on the validity of this signature. This takes into account the cryptographic validity of the signature, the signature's chain of trust and the validity of the timestamp token (if present).

Returns

True if all constraints are satisfied, False otherwise.

summary_fields()

pretty_print_details()

pretty_print_sections()

class `pyhanko.sign.validation.status.SignatureCoverageLevel(value)`

Bases: `OrderedEnum`

Indicate the extent to which a PDF signature (cryptographically) covers a document. Note that this does *not* pass judgment on whether uncovered updates are legitimate or not, but as a general rule, a legitimate signature will satisfy at least `ENTIRE_REVISION`.

UNCLEAR = 0

The signature's coverage is unclear and/or disconnected. In standard PDF signatures, this is usually a bad sign.

CONTIGUOUS_BLOCK_FROM_START = 1

The signature covers a contiguous block in the PDF file stretching from the first byte of the file to the last byte in the indicated `/ByteRange`. In other words, the only interruption in the byte range is fully occupied by the signature data itself.

ENTIRE_REVISION = 2

The signature covers the entire revision in which it occurs, but incremental updates may have been added later. This is not necessarily evidence of tampering. In particular, it is expected when a file contains multiple signatures. Nonetheless, caution is required.

ENTIRE_FILE = 3

The entire file is covered by the signature.

class `pyhanko.sign.validation.status.ModificationInfo(coverage: pyhanko.sign.validation.status.SignatureCoverageLevel = None, diff_result: Union[pyhanko.sign.diff_analysis.policy_api.DiffResult, pyhanko.sign.diff_analysis.policy_api.SuspiciousModification, NoneType] = None)`

Bases: `object`

coverage: `SignatureCoverageLevel` = `None`

Indicates how much of the document is covered by the signature.

diff_result: `Optional[Union[DiffResult, SuspiciousModification]] = None`

Result of the difference analysis run on the file:

- If `None`, no difference analysis was run.
- If the difference analysis was successful, this attribute will contain a `DiffResult` object.

- If the difference analysis failed due to unforeseen or suspicious modifications, the *SuspiciousModification* exception thrown by the difference policy will be stored in this attribute.

property modification_level: `Optional[ModificationLevel]`

Indicates the degree to which the document was modified after the signature was applied.

Will be `None` if difference analysis results are not available; an instance of *ModificationLevel* otherwise.

```
class pyhanko.sign.validation.status.PdfSignatureStatus(intact: bool, valid: bool,
    trust_problem_indic:
    Optional[AdESSubIndic], signing_cert:
    Certificate, pkcs7_signature_mechanism:
    str, md_algorithm: str, validation_path:
    ValidationPath, signer_reported_dt:
    Optional[datetime] = None,
    timestamp_validity:
    Optional[TimestampSignatureStatus] =
    None, content_timestamp_validity:
    Optional[TimestampSignatureStatus] =
    None, ac_attrs:
    Optional[CertifiedAttributes] = None,
    ac_validation_errs: Op-
    tional[Collection[Union[PathValidationError,
    PathBuildingError]]] = None,
    cades_signer_attrs: Op-
    tional[CAAdESSignerAttributeAssertions] =
    None, coverage:
    Optional[SignatureCoverageLevel] = None,
    diff_result: Optional[Union[DiffResult,
    SuspiciousModification]] = None,
    docmdp_ok: Optional[bool] = None,
    has_seed_values: bool = False,
    seed_value_constraint_error:
    Optional[SigSeedValueValidationError] =
    None)
```

Bases: *ModificationInfo*, *StandardCMSSignatureStatus*

Class to indicate the validation status of a PDF signature.

docmdp_ok: `Optional[bool] = None`

Indicates whether the signature's *modification_level* is in line with the document signature policy in force.

If `None`, compliance could not be determined.

has_seed_values: `bool = False`

Records whether the signature form field has seed values.

seed_value_constraint_error: `Optional[SigSeedValueValidationError] = None`

Records the reason for failure if the signature field's seed value constraints didn't validate.

property bottom_line: `bool`

Formulates a general judgment on the validity of this signature. This takes into account the cryptographic validity of the signature, the signature's chain of trust, compliance with the document modification policy, seed value constraint compliance and the validity of the timestamp token (if present).

Returns

True if all constraints are satisfied, False otherwise.

property seed_value_ok: bool

Indicates whether the signature satisfies all mandatory constraints in the seed value dictionary of the associated form field.

Warning: Currently, not all seed value entries are recognised by the signer and/or the validator, so this judgment may not be entirely accurate in some cases.

See [SigSeedValueSpec](#).

summary_fields()**pretty_print_sections()**

```
class pyhanko.sign.validation.status.DocumentTimestampStatus(intact: bool, valid: bool,
                                                            trust_problem_indic:
                                                                Optional[AdESSubIndic],
                                                            signing_cert: Certificate,
                                                            pkcs7_signature_mechanism: str,
                                                            md_algorithm: str, validation_path:
                                                                ValidationPath, timestamp: datetime,
                                                            coverage:
                                                                Optional[SignatureCoverageLevel]
                                                                = None, diff_result:
                                                                Optional[Union[DiffResult,
                                                                SuspiciousModification]] = None)
```

Bases: [ModificationInfo](#), [TimestampSignatureStatus](#)

Class to indicate the validation status of a PDF document timestamp.

pyhanko.sign.validation.utils module

```
pyhanko.sign.validation.utils.validate_raw(signature: bytes, signed_data: bytes, cert: Certificate,
                                           signature_algorithm: SignedDigestAlgorithm,
                                           md_algorithm: str, prehashed=False,
                                           weak_hash_algorithms=frozenset({'md2', 'md5', 'sha1'}))
```

Validate a raw signature. Internal API.

```
pyhanko.sign.validation.utils.extract_message_digest(signer_info: SignerInfo)
```

Direct members

This package also exports a number of convenience functions at the package level. These are all synchronous wrappers around asynchronous functions. Some are deprecated and preserved only for compatibility reasons.

```
pyhanko.sign.validation.validate_pdf_signature(embedded_sig: EmbeddedPdfSignature,
                                              signer_validation_context:
                                                  Optional[ValidationContext] = None,
                                              ts_validation_context: Optional[ValidationContext] =
                                                  None, diff_policy: Optional[DiffPolicy] = None,
                                              key_usage_settings: Optional[KeyUsageConstraints] =
                                                  None, skip_diff: bool = False) → PdfSignatureStatus
```

Changed in version 0.9.0: Wrapper around `async_validate_pdf_signature()`.

Validate a PDF signature.

Parameters

- **embedded_sig** – Embedded signature to evaluate.
- **signer_validation_context** – Validation context to use to validate the signature’s chain of trust.
- **ts_validation_context** – Validation context to use to validate the timestamp’s chain of trust (defaults to `signer_validation_context`).
- **diff_policy** – Policy to evaluate potential incremental updates that were appended to the signed revision of the document. Defaults to `DEFAULT_DIFF_POLICY`.
- **key_usage_settings** – A `KeyUsageConstraints` object specifying which key usages must or must not be present in the signer’s certificate.
- **skip_diff** – If True, skip the difference analysis step entirely.

Returns

The status of the PDF signature in question.

```
pyhanko.sign.validation.validate_cms_signature(signed_data: ~asn1crypto.cms.SignedData, status_cls:
~typing.Type[~pyhanko.sign.validation.StatusType] =
<class
'pyhanko.sign.validation.status.SignatureStatus'>,
raw_digest: ~typing.Optional[bytes] = None,
validation_context: ~typing.Optional[~pyhanko.certvalidator.context.ValidationContext]
= None, status_kwargs: ~typing.Optional[dict] =
None, key_usage_settings: ~typing.Optional[~pyhanko.sign.validation.settings.KeyUsageConstraints]
= None, encap_data_invalid=False)
```

Deprecated since version 0.9.0: Use `async_validate_cms_signature()` instead.

Changed in version 0.7.0: Now handles both detached and enveloping signatures.

Validate a CMS signature (i.e. a `SignedData` object).

Parameters

- **signed_data** – The `asn1crypto.cms.SignedData` object to validate.
- **status_cls** – Status class to use for the validation result.
- **raw_digest** – Raw digest, computed from context.
- **validation_context** – Validation context to validate the signer’s certificate.
- **status_kwargs** – Other keyword arguments to pass to the `status_class` when reporting validation results.
- **key_usage_settings** – A `KeyUsageConstraints` object specifying which key usages must or must not be present in the signer’s certificate.
- **encap_data_invalid** – If True, the encapsulated data inside the CMS is invalid, but the remaining validation logic still has to be run (e.g. a timestamp token, which requires validation of the embedded message imprint).

This option is considered internal API, the semantics of which may change without notice in the future.

Returns

A *SignatureStatus* object (or an instance of a proper subclass)

```
pyhanko.sign.validation.validate_detached_cms(input_data: Union[bytes, IO, ContentInfo,
                                                                EncapsulatedContentInfo], signed_data: SignedData,
                                              signer_validation_context: Optional[ValidationContext]
                                              = None, ts_validation_context:
                                              Optional[ValidationContext] = None,
                                              key_usage_settings: Optional[KeyUsageConstraints] =
                                              None, chunk_size=4096, max_read=None) →
                                              StandardCMSSignatureStatus
```

Deprecated since version 0.9.0: Use *generic_cms.async_validate_detached_cms()* instead.

Validate a detached CMS signature.

Parameters

- **input_data** – The input data to sign. This can be either a `bytes` object, a file-like object or a `cms.ContentInfo` / `cms.EncapsulatedContentInfo` object.
If a CMS content info object is passed in, the *content* field will be extracted.
- **signed_data** – The `cms.SignedData` object containing the signature to verify.
- **signer_validation_context** – Validation context to use to verify the signer certificate's trust.
- **ts_validation_context** – Validation context to use to verify the TSA certificate's trust, if a timestamp token is present. By default, the same validation context as that of the signer is used.
- **key_usage_settings** – Key usage parameters for the signer.
- **chunk_size** – Chunk size to use when consuming input data.
- **max_read** – Maximal number of bytes to read from the input stream.

Returns

A description of the signature's status.

```
pyhanko.sign.validation.validate_pdf_timestamp(embedded_sig: EmbeddedPdfSignature,
                                              validation_context: Optional[ValidationContext] =
                                              None, diff_policy: Optional[DiffPolicy] = None,
                                              skip_diff: bool = False) → DocumentTimestampStatus
```

Changed in version 0.9.0: Wrapper around *async_validate_pdf_timestamp()*.

Validate a PDF document timestamp.

Parameters

- **embedded_sig** – Embedded signature to evaluate.
- **validation_context** – Validation context to use to validate the timestamp's chain of trust.
- **diff_policy** – Policy to evaluate potential incremental updates that were appended to the signed revision of the document. Defaults to `DEFAULT_DIFF_POLICY`.
- **skip_diff** – If `True`, skip the difference analysis step entirely.

Returns

The status of the PDF timestamp in question.

`pyhanko.sign.validation.validate_pdf_ltv_signature`(*embedded_sig*: [EmbeddedPdfSignature](#),
validation_type: [RevocationInfoValidationType](#),
validation_context_kwargs=None,
bootstrap_validation_context=None,
force_revinfo=False, *diff_policy*:
Optional[[DiffPolicy](#)] = None, *key_usage_settings*:
Optional[[KeyUsageConstraints](#)] = None,
skip_diff: bool = False) → [PdfSignatureStatus](#)

Changed in version 0.9.0: Wrapper around `async_validate_pdf_ltv_signature()`.

Validate a PDF LTV signature according to a particular profile.

Parameters

- **embedded_sig** – Embedded signature to evaluate.
- **validation_type** – Validation profile to use.
- **validation_context_kwargs** – Keyword args to instantiate `pyhanko_certvalidator.ValidationContext` objects needed over the course of the validation.
- **bootstrap_validation_context** – Validation context used to validate the current timestamp.
- **force_revinfo** – Require all certificates encountered to have some form of live revocation checking provisions.
- **diff_policy** – Policy to evaluate potential incremental updates that were appended to the signed revision of the document. Defaults to `DEFAULT_DIFF_POLICY`.
- **key_usage_settings** – A [KeyUsageConstraints](#) object specifying which key usages must or must not be present in the signer's certificate.
- **skip_diff** – If True, skip the difference analysis step entirely.

Returns

The status of the signature.

`pyhanko.sign.validation.add_validation_info`(*embedded_sig*: [EmbeddedPdfSignature](#),
validation_context: [ValidationContext](#),
skip_timestamp=False, *add_vri_entry*=True,
in_place=False, *output*=None, *force_write*=False,
chunk_size=4096)

Changed in version 0.9.0: Wrapper around `async_add_validation_info()`

Add validation info (CRLs, OCSP responses, extra certificates) for a signature to the DSS of a document in an incremental update.

Parameters

- **embedded_sig** – The signature for which the revocation information needs to be collected.
- **validation_context** – The validation context to use.
- **skip_timestamp** – If True, do not attempt to validate the timestamp attached to the signature, if one is present.
- **add_vri_entry** – Add a /VRI entry for this signature to the document security store. Default is True.
- **output** – Write the output to the specified output stream. If None, write to a new `BytesIO` object. Default is None.

- **in_place** – Sign the original input stream in-place. This parameter overrides **output**.
- **chunk_size** – Chunk size parameter to use when copying output to a new stream (irrelevant if **in_place** is **True**).
- **force_write** – Force a new revision to be written, even if not necessary (i.e. when all data in the validation context is already present in the DSS).

Returns

The (file-like) output object to which the result was written.

3.1.2 Submodules

pyhanko.config module

class pyhanko.config.StdLogOutput(*value*)

Bases: Enum

An enumeration.

STDERR = 1

STDOUT = 2

class pyhanko.config.LogConfig(*level: Union[int, str], output: Union[pyhanko.config.StdLogOutput, str]*)

Bases: object

level: Union[int, str]

Logging level, should be one of the levels defined in the logging module.

output: Union[StdLogOutput, str]

Name of the output file, or a standard one.

static parse_output_spec(spec) → Union[StdLogOutput, str]

class pyhanko.config.CLIFConfig(*validation_contexts: Dict[str, dict], stamp_styles: Dict[str, dict], default_validation_context: str, default_stamp_style: str, time_tolerance: datetime.timedelta, retroactive_revinfo: bool, log_config: Dict[Union[str, NoneType], pyhanko.config.LogConfig], pemder_setups: Dict[str, dict], pkcs12_setups: Dict[str, dict], pkcs11_setups: Dict[str, dict], beid_module_path: Union[str, NoneType])*

Bases: object

validation_contexts: Dict[str, dict]

stamp_styles: Dict[str, dict]

default_validation_context: str

default_stamp_style: str

time_tolerance: timedelta

retroactive_revinfo: bool

log_config: Dict[Optional[str], LogConfig]

pemder_setups: Dict[str, dict]

```

pkcs12_setups: Dict[str, dict]
pkcs11_setups: Dict[str, dict]
beid_module_path: Optional[str]
get_validation_context(name=None, as_dict=False)
get_signer_key_usages(name=None) → KeyUsageConstraints
get_stamp_style(name=None) → TextStampStyle
get_pkcs11_config(name)
get_pkcs12_config(name)
get_pemder_config(name)

pyhanko.config.init_validation_context_kwargs(*, trust, trust_replace, other_certs,
                                              retroactive_revinfo=False, time_tolerance=None)

pyhanko.config.parse_trust_config(trust_config, time_tolerance, retroactive_revinfo) → dict
pyhanko.config.parse_logging_config(log_config_spec) → Dict[Optional[str], LogConfig]

class pyhanko.config.PKCS12SignatureConfig(pfx_file: str, other_certs: Optional[List[Certificate]] =
                                           None, pfx_passphrase: Optional[bytes] = None,
                                           prompt_passphrase: bool = True, prefer_pss: bool = False)

```

Bases: *ConfigurableMixin*

Configuration for a signature using key material on disk, contained in a PKCS#12 bundle.

```

pfx_file: str
    Path to the PKCS#12 file.

other_certs: List[Certificate] = None
    Other relevant certificates.

pfx_passphrase: bytes = None
    PKCS#12 passphrase (if relevant).

prompt_passphrase: bool = True
    Prompt for the PKCS#12 passphrase. Default is True.

```

Note: If key_passphrase is not None, this setting has no effect.

```

prefer_pss: bool = False
    Prefer PSS to PKCS#1 v1.5 padding when creating RSA signatures.

```

```

classmethod process_entries(config_dict)

```

Hook method that can modify the configuration dictionary to overwrite or tweak some of their values (e.g. to convert string parameters into more complex Python objects)

Subclasses that override this method should call `super().process_entries()`, and leave keys that they do not recognise untouched.

Parameters

config_dict – A dictionary containing configuration values.

Raises

ConfigurationError – when there is a problem processing a relevant entry.

instantiate(*provided_pfx_passphrase: Optional[bytes] = None*) → *SimpleSigner*

```
class pyhanko.config.PemDerSignatureConfig(key_file: str, cert_file: str, other_certs:
Optional[List[Certificate]] = None, key_passphrase:
Optional[bytes] = None, prompt_passphrase: bool = True,
prefer_pss: bool = False)
```

Bases: *ConfigurableMixin*

Configuration for a signature using PEM or DER-encoded key material on disk.

key_file: str

Signer's private key.

cert_file: str

Signer's certificate.

other_certs: List[Certificate] = None

Other relevant certificates.

key_passphrase: bytes = None

Signer's key passphrase (if relevant).

prompt_passphrase: bool = True

Prompt for the key passphrase. Default is True.

Note: If *key_passphrase* is not None, this setting has no effect.

prefer_pss: bool = False

Prefer PSS to PKCS#1 v1.5 padding when creating RSA signatures.

classmethod process_entries(*config_dict*)

Hook method that can modify the configuration dictionary to overwrite or tweak some of their values (e.g. to convert string parameters into more complex Python objects)

Subclasses that override this method should call `super().process_entries()`, and leave keys that they do not recognise untouched.

Parameters

config_dict – A dictionary containing configuration values.

Raises

ConfigurationError – when there is a problem processing a relevant entry.

instantiate(*provided_key_passphrase: Optional[bytes] = None*) → *SimpleSigner*

```
class pyhanko.config.TokenCriteria(label: Optional[str] = None, serial: Optional[bytes] = None)
```

Bases: *ConfigurableMixin*

New in version 0.14.0.

Search criteria for a PKCS#11 token.

label: Optional[str] = None

Label of the token to use. If None, there is no constraint.

serial: `Optional[bytes] = None`

Serial number of the token to use. If `None`, there is no constraint.

classmethod `process_entries(config_dict)`

Hook method that can modify the configuration dictionary to overwrite or tweak some of their values (e.g. to convert string parameters into more complex Python objects)

Subclasses that override this method should call `super().process_entries()`, and leave keys that they do not recognise untouched.

Parameters

config_dict – A dictionary containing configuration values.

Raises

`ConfigurationError` – when there is a problem processing a relevant entry.

class `pyhanko.config.PKCS11PinEntryMode(value)`

Bases: `Enum`

Pin entry behaviour if the user PIN is not supplied as part of the config.

PROMPT = 1

Prompt for a PIN (the default).

Note: This value is only processed by the CLI, and ignored when the PKCS#11 signer is called from library code. In those cases, the default is effectively **`SKIP`**.

DEFER = 2

Let the PKCS #11 module handle its own authentication during login.

Note: This applies to some devices that have physical PIN pads, for example.

SKIP = 3

Skip the login process altogether.

Note: This applies to some devices that manage user authentication outside the scope of PKCS #11 entirely.

static `parse_mode_setting(value: Any) → PKCS11PinEntryMode`

class `pyhanko.config.PKCS11SignatureConfig(module_path: str, cert_label: Optional[str] = None, cert_id: Optional[bytes] = None, signing_certificate: Optional[Certificate] = None, token_criteria: Optional[TokenCriteria] = None, other_certs: Optional[List[Certificate]] = None, key_label: Optional[str] = None, key_id: Optional[bytes] = None, slot_no: Optional[int] = None, user_pin: Optional[str] = None, prompt_pin: PKCS11PinEntryMode = PKCS11PinEntryMode.PROMPT, other_certs_to_pull: Optional[Iterable[str]] = (), bulk_fetch: bool = True, prefer_pss: bool = False, raw_mechanism: bool = False)`

Bases: **`ConfigurableMixin`**

Configuration for a PKCS#11 signature.

This class is used to load PKCS#11 setup information from YAML configuration.

module_path: `str`

Path to the PKCS#11 module shared object.

cert_label: `Optional[str] = None`

PKCS#11 label of the signer's certificate.

cert_id: `Optional[bytes] = None`

PKCS#11 ID of the signer's certificate.

signing_certificate: `Optional[Certificate] = None`

The signer's certificate. If present, `cert_id` and `cert_label` will not be used to obtain the signer's certificate from the PKCS#11 token.

Note: This can be useful in case the signer's certificate is not available on the token, or if you would like to present a different certificate than the one provided on the token.

token_criteria: `Optional[TokenCriteria] = None`

PKCS#11 token name

other_certs: `Optional[List[Certificate]] = None`

Other relevant certificates.

key_label: `Optional[str] = None`

PKCS#11 label of the signer's private key. Defaults to `cert_label` if the latter is specified and `key_id` is not.

key_id: `Optional[bytes] = None`

PKCS#11 key ID.

slot_no: `Optional[int] = None`

Slot number of the PKCS#11 slot to use.

user_pin: `Optional[str] = None`

The user's PIN. If unspecified, the user will be prompted for a PIN if `prompt_pin` is `True`.

Warning: Some PKCS#11 tokens do not allow the PIN code to be communicated in this way, but manage their own authentication instead (the Belgian eID middleware is one such example). For such tokens, leave this setting set to `None` and additionally set `prompt_pin` to `False`.

prompt_pin: `PKCS11PinEntryMode = 1`

Set PIN entry and PKCS #11 login behaviour.

Note: If `user_pin` is not `None`, this setting has no effect.

other_certs_to_pull: `Optional[Iterable[str]] = ()`

List labels of other certificates to pull from the PKCS#11 device. Defaults to the empty tuple. If `None`, pull *all* certificates.

bulk_fetch: `bool = True`

Boolean indicating the fetching strategy. If `True`, fetch all certs and filter the unneeded ones. If `False`, fetch the requested certs one by one. Default value is `True`, unless `other_certs_to_pull` has one or fewer elements, in which case it is always treated as `False`.

prefer_pss: `bool = False`

Prefer PSS to PKCS#1 v1.5 padding when creating RSA signatures.

raw_mechanism: `bool = False`

Invoke the raw variant of the PKCS#11 signing operation.

Note: This is currently only supported for ECDSA signatures.

classmethod `check_config_keys(keys_supplied: Set[str])`

Check whether all supplied keys are meaningful.

Parameters

keys_supplied – The keys supplied in the configuration.

Raises

ConfigurationError – if at least one key does not make sense.

classmethod `process_entries(config_dict)`

Hook method that can modify the configuration dictionary to overwrite or tweak some of their values (e.g. to convert string parameters into more complex Python objects)

Subclasses that override this method should call `super().process_entries()`, and leave keys that they do not recognise untouched.

Parameters

config_dict – A dictionary containing configuration values.

Raises

ConfigurationError – when there is a problem processing a relevant entry.

`pyhanko.config.parse_cli_config(yaml_str) → CLIConfig`

`pyhanko.config.process_config_dict(config_dict: dict) → dict`

pyhanko.stamp module

Utilities for stamping PDF files.

Here ‘stamping’ loosely refers to adding small overlays (QR codes, text boxes, etc.) on top of already existing content in PDF files.

The code in this module is also used by the `sign` module to render signature appearances.

class `pyhanko.stamp.AnnotAppearances(normal: IndirectObject, rollover: Optional[IndirectObject] = None, down: Optional[IndirectObject] = None)`

Bases: `object`

Convenience abstraction to set up an appearance dictionary for a PDF annotation.

Annotations can have three appearance streams, which can be roughly characterised as follows:

- *normal*: the only required one, and the default one;
- *rollover*: used when mousing over the annotation;

- *down*: used when clicking the annotation.

These are given as references to form XObjects.

Note: This class only covers the simple case of an appearance dictionary for an annotation with only one appearance state.

See § 12.5.5 in ISO 32000-1 for further information.

as_pdf_object() → *DictionaryObject*

Convert the *AnnotationAppearances* instance to a PDF dictionary.

Returns

A *DictionaryObject* that can be plugged into the /AP entry of an annotation dictionary.

```
class pyhanko.stamp.BaseStampStyle(border_width: int = 3, background:
    ~typing.Optional[~pyhanko.pdf_utils.content.PdfContent] = None,
    background_layout: ~pyhanko.pdf_utils.layout.SimpleBoxLayoutRule
    = SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MID: 2>,
    y_align=<AxisAlignment.ALIGN_MID: 2>, margins=Margins(left=5,
    right=5, top=5, bottom=5),
    inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>),
    background_opacity: float = 0.6)
```

Bases: *ConfigurableMixin*

Base class for stamp styles.

border_width: `int = 3`

Border width in user units (for the stamp, not the text box).

background: *PdfContent* = `None`

PdfContent instance that will be used to render the stamp's background.

background_layout: *SimpleBoxLayoutRule* =
SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MID: 2>,
y_align=<AxisAlignment.ALIGN_MID: 2>, margins=Margins(left=5, right=5, top=5,
bottom=5), inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>)

Layout rule to render the background inside the stamp's bounding box. Only used if the background has a fully specified *PdfContent*.box.

Otherwise, the renderer will position the cursor at (left_margin, bottom_margin) and render the content as-is.

background_opacity: `float = 0.6`

Opacity value to render the background at. This should be a floating-point number between 0 and 1.

classmethod process_entries(*config_dict*)

This implementation of *process_entries()* processes the *background* configuration value. This can either be a path to an image file, in which case it will be turned into an instance of *PdfImage*, or the special value `__stamp__`, which is an alias for *STAMP_ART_CONTENT*.

create_stamp(*writer*: *BasePdfFileWriter*, *box*: *BoxConstraints*, *text_params*: *dict*) → *BaseStamp*

```
class pyhanko.stamp.TextStampStyle(border_width: int = 3, background:
    ~typing.Optional[~pyhanko.pdf_utils.content.PdfContent] = None,
    background_layout: ~pyhanko.pdf_utils.layout.SimpleBoxLayoutRule
    = SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MID: 2>,
    y_align=<AxisAlignment.ALIGN_MID: 2>, margins=Margins(left=5,
    right=5, top=5, bottom=5),
    inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>),
    background_opacity: float = 0.6, text_box_style:
    ~pyhanko.pdf_utils.text.TextBoxStyle =
    TextBoxStyle(font=<pyhanko.pdf_utils.font.basic.SimpleFontEngineFactory
    object>, font_size=10, leading=None, border_width=0,
    box_layout_rule=None, vertical_text=False), inner_content_layout:
    ~typing.Optional[~pyhanko.pdf_utils.layout.SimpleBoxLayoutRule] =
    None, stamp_text: str = '%(ts)s', timestamp_format: str = '%Y-%m-%d
    %H:%M:%S %Z')
```

Bases: [BaseStampStyle](#)

Style for text-based stamps.

Roughly speaking, this stamp type renders some predefined (but parametrised) piece of text inside a text box, and possibly applies a background to it.

text_box_style: [TextBoxStyle](#) =
TextBoxStyle(font=<pyhanko.pdf_utils.font.basic.SimpleFontEngineFactory object>,
font_size=10, leading=None, border_width=0, box_layout_rule=None,
vertical_text=False)

The text box style for the internal text box used.

inner_content_layout: [SimpleBoxLayoutRule](#) = None

Rule determining the position and alignment of the inner text box within the stamp.

Warning: This only affects the position of the box, not the alignment of the text within.

stamp_text: str = '%(ts)s'

Text template for the stamp. The template can contain an interpolation parameter `ts` that will be replaced by the stamping time.

Additional parameters may be added if necessary. Values for these must be passed to the `__init__()` method of the [TextStamp](#) class in the `text_params` argument.

timestamp_format: str = '%Y-%m-%d %H:%M:%S %Z'

Datetime format used to render the timestamp.

create_stamp(writer: [BasePdfFileWriter](#), box: [BoxConstraints](#), text_params: dict) → [TextStamp](#)

```
class pyhanko.stamp.QRStampStyle(border_width: int = 3, background:
    ~typing.Optional[~pyhanko.pdf_utils.content.PdfContent] = None,
    background_layout: ~pyhanko.pdf_utils.layout.SimpleBoxLayoutRule =
    SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MID: 2>,
        y_align=<AxisAlignment.ALIGN_MID: 2>, margins=Margins(left=5,
            right=5, top=5, bottom=5),
        inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>),
    background_opacity: float = 0.6, text_box_style:
    ~pyhanko.pdf_utils.text.TextBoxStyle =
    TextBoxStyle(font=<pyhanko.pdf_utils.font.basic.SimpleFontEngineFactory
        object>, font_size=10, leading=None, border_width=0,
        box_layout_rule=None, vertical_text=False), inner_content_layout:
    ~typing.Optional[~pyhanko.pdf_utils.layout.SimpleBoxLayoutRule] =
    None, stamp_text: str = 'Digital version available at\nthis url:
    %(url)s\nTimestamp: %(ts)s', timestamp_format: str = '%Y-%m-%d
    %H:%M:%S %Z', innsep: int = 3, qr_inner_size: ~typing.Optional[int]
    = None, qr_position: ~pyhanko.stamp.QRPosition =
    QRPosition.LEFT_OF_TEXT, qr_inner_content:
    ~typing.Optional[~pyhanko.pdf_utils.content.PdfContent] = None)
```

Bases: [TextStampStyle](#)

Style for text-based stamps together with a QR code.

This is exactly the same as a text stamp, except that the text box is rendered with a QR code to the left of it.

innsep: int = 3

Inner separation inside the stamp.

stamp_text: str = 'Digital version available at\nthis url: %(url)s\nTimestamp: %(ts)s'

Text template for the stamp. The description of [TextStampStyle.stamp_text](#) still applies, but an additional default interpolation parameter `url` is available. This parameter will be replaced with the URL that the QR code points to.

qr_inner_size: Optional[int] = None

Size of the QR code in the inner layout. By default, this is in user units, but if the stamp has a fully defined bounding box, it may be rescaled depending on `inner_content_layout`.

If unspecified, a reasonable default will be used.

qr_position: QRPosition = SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MIN: 1>, y_align=<AxisAlignment.ALIGN_MID: 2>, margins=Margins(left=0, right=0, top=0, bottom=0), inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>)

Position of the QR code relative to the text box.

qr_inner_content: Optional[PdfContent] = None

Inner graphics content to be included in the QR code (experimental).

classmethod process_entries(config_dict)

This implementation of [process_entries\(\)](#) processes the background configuration value. This can either be a path to an image file, in which case it will be turned into an instance of [PdfImage](#), or the special value `__stamp__`, which is an alias for [STAMP_ART_CONTENT](#).

create_stamp(writer: BasePdfFileWriter, box: BoxConstraints, text_params: dict) → QRStamp

```
class pyhanko.stamp.StaticStampStyle(border_width: int = 3, background:
    ~typing.Optional[~pyhanko.pdf_utils.content.PdfContent] = None,
    background_layout:
    ~pyhanko.pdf_utils.layout.SimpleBoxLayoutRule =
    SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MID: 2>,
        y_align=<AxisAlignment.ALIGN_MID: 2>,
        margins=Margins(left=5, right=5, top=5, bottom=5),
        inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>),
    background_opacity: float = 1.0)
```

Bases: [BaseStampStyle](#)

Stamp style that does not include any custom parts; it only renders the background.

background_opacity: float = 1.0

Opacity value to render the background at. This should be a floating-point number between 0 and 1.

classmethod from_pdf_file(file_name, page_ix=0, **kwargs) → [StaticStampStyle](#)

Create a [StaticStampStyle](#) from a page from an external PDF document. This is a convenience wrapper around `ImportedPdfContent`.

The remaining keyword arguments are passed to [StaticStampStyle](#)’s init method.

Parameters

- **file_name** – File name of the external PDF document.
- **page_ix** – Page index to import. The default is 0, i.e. the first page.

create_stamp(writer: [BasePdfFileWriter](#), box: [BoxConstraints](#), text_params: dict) → [StaticContentStamp](#)

```
class pyhanko.stamp.QRPosition(value)
```

Bases: Enum

QR positioning constants, with the corresponding default content layout rule.

```
LEFT_OF_TEXT = SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MIN: 1>,
    y_align=<AxisAlignment.ALIGN_MID: 2>, margins=Margins(left=0, right=0, top=0,
    bottom=0), inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>)
```

```
RIGHT_OF_TEXT = SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MAX: 3>,
    y_align=<AxisAlignment.ALIGN_MID: 2>, margins=Margins(left=0, right=0, top=0,
    bottom=0), inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>)
```

```
ABOVE_TEXT = SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MID: 2>,
    y_align=<AxisAlignment.ALIGN_MAX: 3>, margins=Margins(left=0, right=0, top=0,
    bottom=0), inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>)
```

```
BELOW_TEXT = SimpleBoxLayoutRule(x_align=<AxisAlignment.ALIGN_MID: 2>,
    y_align=<AxisAlignment.ALIGN_MIN: 1>, margins=Margins(left=0, right=0, top=0,
    bottom=0), inner_content_scaling=<InnerScaling.SHRINK_TO_FIT: 4>)
```

property horizontal_flow

classmethod from_config(config_str) → [QRPosition](#)

Convert from a configuration string.

Parameters

config_str – A string: ‘left’, ‘right’, ‘top’, ‘bottom’

Returns

An [QRPosition](#) value.

Raises

[ConfigurationError](#) – on unexpected string inputs.

```
class pyhanko.stamp.BaseStamp(writer: BasePdfFileWriter, style, box: Optional[BoxConstraints] = None)
```

Bases: [PdfContent](#)

render()

Compile the content to graphics operators.

register() → [IndirectObject](#)

Register the stamp with the writer coupled to this instance, and cache the returned reference.

This works by calling [PdfContent.as_form_xobject\(\)](#).

Returns

An indirect reference to the form XObject containing the stamp.

apply(dest_page: int, x: int, y: int)

Apply a stamp to a particular page in the PDF writer attached to this [BaseStamp](#) instance.

Parameters

- **dest_page** – Index of the page to which the stamp is to be applied (starting at 0).
- **x** – Horizontal position of the stamp's lower left corner on the page.
- **y** – Vertical position of the stamp's lower left corner on the page.

Returns

A reference to the affected page object, together with a (width, height) tuple describing the dimensions of the stamp.

as_appearances() → [AnnotAppearances](#)

Turn this stamp into an appearance dictionary for an annotation (or a form field widget), after rendering it. Only the normal appearance will be defined.

Returns

An instance of [AnnotAppearances](#).

```
class pyhanko.stamp.TextStamp(writer: BasePdfFileWriter, style, text_params=None, box: Optional[BoxConstraints] = None)
```

Bases: [BaseStamp](#)

Class that renders a text stamp as specified by an instance of [TextStampStyle](#).

get_default_text_params()

Compute values for the default string interpolation parameters to be applied to the template string specified in the stamp style. This method does not take into account the `text_params` init parameter yet.

Returns

A dictionary containing the parameters and their values.

```
class pyhanko.stamp.QRStamp(writer: BasePdfFileWriter, url: str, style: QRStampStyle, text_params=None, box: Optional[BoxConstraints] = None)
```

Bases: [TextStamp](#)

get_default_text_params()

Compute values for the default string interpolation parameters to be applied to the template string specified in the stamp style. This method does not take into account the `text_params` init parameter yet.

Returns

A dictionary containing the parameters and their values.

apply(dest_page, x, y)

Apply a stamp to a particular page in the PDF writer attached to this [BaseStamp](#) instance.

Parameters

- **dest_page** – Index of the page to which the stamp is to be applied (starting at 0).
- **x** – Horizontal position of the stamp’s lower left corner on the page.
- **y** – Vertical position of the stamp’s lower left corner on the page.

Returns

A reference to the affected page object, together with a `(width, height)` tuple describing the dimensions of the stamp.

```
class pyhanko.stamp.StaticContentStamp(writer: BasePdfFileWriter, style: StaticStampStyle, box:
                                         BoxConstraints)
```

Bases: [BaseStamp](#)

Class representing stamps with static content.

```
pyhanko.stamp.text_stamp_file(input_name: str, output_name: str, style: TextStampStyle, dest_page: int, x:
                               int, y: int, text_params=None)
```

Add a text stamp to a file.

Parameters

- **input_name** – Path to the input file.
- **output_name** – Path to the output file.
- **style** – Text stamp style to use.
- **dest_page** – Index of the page to which the stamp is to be applied (starting at 0).
- **x** – Horizontal position of the stamp’s lower left corner on the page.
- **y** – Vertical position of the stamp’s lower left corner on the page.
- **text_params** – Additional parameters for text template interpolation.

```
pyhanko.stamp.qr_stamp_file(input_name: str, output_name: str, style: QRStampStyle, dest_page: int, x: int,
                             y: int, url: str, text_params=None)
```

Add a QR stamp to a file.

Parameters

- **input_name** – Path to the input file.
- **output_name** – Path to the output file.
- **style** – QR stamp style to use.
- **dest_page** – Index of the page to which the stamp is to be applied (starting at 0).
- **x** – Horizontal position of the stamp’s lower left corner on the page.
- **y** – Vertical position of the stamp’s lower left corner on the page.

- **url** – URL for the QR code to point to.
- **text_params** – Additional parameters for text template interpolation.

`pyhanko.stamp.STAMP_ART_CONTENT = <pyhanko.pdf_utils.content.RawContent object>`

Hardcoded stamp background that will render a stylised image of a stamp using PDF graphics operators (see below).

RELEASE HISTORY

4.1 0.14.0

Release date: 2022-09-17

4.1.1 Note

This release contains a mixture of minor and major changes. Of particular note is the addition of automated metadata management support, including XMP metadata. This change affects almost every PDF write operation in the background. While pyHanko has very good test coverage, some instability and regressions may ensue. Bug reports are obviously welcome.

4.1.2 Breaking changes

The breaking changes in this release are all relatively minor. Chances are that your code isn't affected at all, other than perhaps by the change to *PreparedByteRangeDigest*.

- `md_algorithm` attribute removed from *PreparedByteRangeDigest* since it wasn't necessary for further processing.
- Low-level change in `raw_get` for PDF container object types (*ArrayObject* and *DictionaryObject*): the `decrypt` parameter is no longer a boolean, but a tri-state enum value of type *EncryptedObjAccess*.
- Developer extension management API moved into *pyhanko.pdf_utils.extensions*.
- *get_courier()* convenience function moved into *pyhanko.pdf_utils.font.basic* and now takes a mandatory writer argument.
- The `token_label` attribute was removed from *PKCS11SignatureConfig*, but will still be parsed (with a deprecation warning).
- The `prompt_pin` attribute in *PKCS11SignatureConfig* was changed from a bool to an enum. See *PKCS11PinEntryMode*.

4.1.3 Dependency updates

- `pytest-aiohttp` updated to 1.0.4
- `certomancer` updated to 0.9.0
- `certomancer-csc-dummy` updated to 0.2.1
- Relax bounds on `uharfbuzz` to allow everything up to the current version (i.e. 0.30.0) as well.
- New optional dependency group `xmp`, which for now only contains `defusedxml`

4.1.4 Bugs fixed

- Allow certificates with no CN in the certificate subject.
- The extension dictionary handling logic can now deal with encrypted documents without actually decrypting the document contents.
- Fix processing error when passing empty strings to `uharfbuzz`; see [issue #132](#).
- Use proper PDF text string serialisation routine in simple font handler, to ensure everything is escaped correctly.
- Ensure that `output_version` is set to at least the input version in incrementally updated files.

4.1.5 New features and enhancements

Signing

- Drop the requirement for `signing_cert` to be set from the start of the signing process in an interrupted signing workflow. This has come up on several occasions in the past, since it's necessary in remote signing scenarios where the certificate is generated or provided on-demand when submitting the document digest to the signing service. See [pull #141](#) for details.
- Add convenience API to set the `/TU` entry on a signature field; see [readable_field_name](#).
- Allow greater control over the initialisation of document timestamp fields.
- New class hierarchy for (un)signed attribute provisioning; see [SignedAttributeProviderSpec](#) and [UnsignedAttributeProviderSpec](#).
- Allow greater control over annotation flags for visible signatures. This is implemented using [VisibleSigSettings](#). See [discussion #150](#).
- Factor out and improve PKCS#11 token finding; see [TokenCriteria](#) and [issue #149](#).
- Factor out and improve PKCS#11 mechanism selection, allowing more raw modes.
- Change pin entry settings for PKCS#11 to be more granular, in order to also allow `PROTECTED_AUTH`; see [issue #133](#).
- Allow the PKCS#11 PIN to be sourced from an environment variable when pyHanko is invoked through the CLI and no PIN is provided in the configuration. PyHanko will now first check the `PYHANKO_PKCS11_PIN` variable before prompting for a PIN. This also works when prompting for PIN entry is disabled altogether.

Note: The PKCS#11 code is now also tested in CI, using [SoftHSMv2](#).

Validation

- Allow validation time overrides in the CLI. Passing in the special value `claimed` tells pyHanko to take the stated signing time in the file at face value. See [issue #130](#).

Encryption

- Also return permissions on owner access to allow for easier inspection.
- Better version enforcement for security handlers.

Layout

- Allow metrics to be specified for simple fonts.
- Provide metrics for default Courier font.
- Experimental option that allows graphics to be embedded in the central area of the QR code; see [qr_inner_content](#).

Miscellaneous

- Basic XMP metadata support with optional `xmp` dependency group.
- Automated metadata management (document info dictionary, XMP metadata).
- Refactor some low-level digesting and CMS validation code.
- Make the CLI print a warning when the key passphrase is left empty.
- Tweak configuration management utilities to better cope with fallback logic for deprecated configuration parameters.
- Move all cross-reference writing logic into [pyhanko.pdf_utils.xref](#).
- Improve error classes and error reporting in the CLI so that errors in non-verbose mode still provide a little more info.

4.2 0.13.2

Release date: 2022-07-02

4.2.1 Note

This is a patch release to address some dependency issues and bugs.

4.2.2 Dependency updates

- `python-barcode` updated and pinned to 0.14.0.

4.2.3 Bugs fixed

- Fix lack of newline after XRef stream header.
- Do not write **DigestMethod** in signature reference dictionaries (deprecated/nonfunctional entry).
- Make `pyhanko.pdf_utils.writer.copy_into_new_writer()` more flexible by allowing caller-specified keyword arguments for the writer object.
- Refine settings for invisible signature fields (see `pyhanko.sign.fields.InvisSigSettings`).
- Correctly read objects from object streams in encrypted documents.

4.3 0.13.1

Release date: 2022-05-01

4.3.1 Note

This is a patch release to update `fontTools` and `uharfbuzz` to address a conflict between the latest `fontTools` and older `uharfbuzz` versions.

4.3.2 Dependency updates

- `fontTools` updated to 4.33.3
- `uharfbuzz` updated to 0.25.0

4.4 0.13.0

Release date: 2022-04-25

4.4.1 Note

Like the previous two releases, this is largely a maintenance release.

4.4.2 Dependency updates

- `asn1crypto` updated to 1.5.1
- `pyhanko-certvalidator` updated to 0.19.5
- `certomancer` updated to 0.8.2
- Depend on `certomancer-csc-dummy` for tests; get rid of `python-pae` test dependency.

4.4.3 Bugs fixed

- Various parsing robustness improvements.
- Be consistent with security handler version bounds.
- Improve coverage of encryption code.
- Ensure owner password gets prioritised in the legacy security handler.

4.4.4 New features and enhancements

Miscellaneous

- Replaced some `ValueError` usages with `PdfError`
- Improvements to error handling in strict mode.
- Make CLI stack traces less noisy by default.

Encryption

- Refactor internal `crypt` module into package.
- Add support for serialising credentials.
- Cleaner credential inheritance for incremental writers.

Signing

- Allow post-signing actions on encrypted files with serialised credentials.
- Improve `--use-pades-lta` ergonomics in CLI.
- Add `--no-pass` parameter to `pemder` CLI.

Validation

- Preparatory scaffolding for AdES status reporting.
- Provide some tolerance against malformed ACs.
- Increase robustness against invalid DNs.

4.5 0.12.1

Release date: 2022-02-26

4.5.1 Dependency updates

- uharfbuzz updated to 0.19.0
- pyhanko-certvalidator updated to 0.19.4
- certomancer updated to 0.8.1

4.5.2 Bugs fixed

- Fix typing issue in DSS reading logic (see [issue #81](#))

4.6 0.12.0

Release date: 2022-01-26

4.6.1 Note

This is largely a maintenance release, and contains no new high-level features or public API changes. As such, upgrading is strongly recommended.

The most significant change is the (rather minimalistic) support for hybrid reference files. Since working with hybrid reference files means dealing with potential ambiguity (which is dangerous when dealing with signatures), creation and validation of signatures in hybrid reference documents is only enabled in nonstrict mode. Hybrid reference files are relatively rare these days, but the internals need to be able to cope with them either way, in order to be able to update such files safely.

4.6.2 New features and enhancements

Miscellaneous

- Significant refactor of cross-reference parsing internals. This doesn't affect any public API entrypoints, but read the reference documentation for [pyhanko.pdf_utils.xref](#) if you happen to have code that directly relies on that internal logic.
- Minimal support for hybrid reference files.
- Add `strict` flag to [IncrementalPdfFileWriter](#).
- Expose `--no-strict-syntax` CLI flag in the `addsig` subcommand.

4.6.3 Bugs fixed

- Ensure that signature appearance bounding boxes are rounded to a reasonable precision. Failure to do so caused issues with some viewers.
- To be consistent with the purpose of the strictness flag, non-essential xref consistency checking is now only enabled when running in strict mode (which is the default).
- The hybrid reference support indirectly fixes some potential silent file corruption issues that could arise when working on particularly ill-behaved hybrid reference files.

4.7 0.11.0

Release date: 2021-12-23

4.7.1 Dependency changes

- Update pyhanko-certvalidator to 0.19.2
- Bump fontTools to 4.28.2
- Update certomancer test dependency to 0.7.1

4.7.2 Breaking changes

Due to import order issues resulting from refactoring of the validation code, some classes and class hierarchies in the higher-level API had to be moved. The affected classes are listed below, with links to their respective new locations in the API reference.

- [*KeyUsageConstraints*](#)
- [*SignatureValidationError*](#)
- [*WeakHashAlgorithmError*](#)
- [*SigSeedValueValidationError*](#)
- [*SignatureStatus*](#)
- [*StandardCMSSignatureStatus*](#)
- [*PdfSignatureStatus*](#)
- [*TimestampSignatureStatus*](#)
- [*DocumentTimestampStatus*](#)

The low-level function `validate_sig_integrity()` was also moved.

4.7.3 New features and enhancements

Signing

- Support embedding attribute certificates into CMS signatures, either in the `certificates` field or using the CAdES `signer-attrs-v2` attribute.
- More explicit errors on unfulfilled text parameters
- Better use of `asyncio` when collecting validation information for timestamps
- Internally disambiguate PAdES and CAdES for the purpose of attribute handling.

Validation

- Refactor `diff_analysis` module into sub-package
- Refactor `validation` module into sub-package (together with portions of [*pyhanko.sign.general*](#)); see [*Breaking changes*](#).
- Make extracted certificate information more easily accessible.
- Integrated attribute certificate validation (requires a separate validation context with trust roots for attribute authorities)
- Report on signer attributes as supplied by the CAdES `signer-attrs-v2` attribute.

Miscellaneous

- Various parsing and error handling improvements to xref processing, object streams, and object header handling.
- Use `NotImplementedError` for unimplemented stream filters instead of less-appropriate exceptions
- Always drop GPOS/GDEF/GSUB when subsetting OpenType and TrueType fonts
- Initial support for string-keyed CFF fonts as CIDFonts (subsetting is still inefficient)
- [*copy_into_new_writer\(\)*](#) is now smarter about how it deals with the `/Producer` line
- Fix a typo in the ASN.1 definition of `signature-policy-store`
- Various, largely aesthetic, cleanup & docstring fixes in internal APIs

4.7.4 Bugs fixed

- Fix a critical bug in content timestamp generation causing the wrong message imprint to be sent to the timestamping service. The bug only affected the signed `content-time-stamp` attribute from CAdES, not the (much more widely used) `signature-time-stamp` attribute. The former timestamps the content (and is part of the signed data), while the latter timestamps the signature (and is therefore not part of the signed data).
- Fix a bug causing an empty unsigned attribute sequence to be written if there were no unsigned attributes. This is not allowed (although many validators accept it), and was a regression introduced in 0.9.0.
- Ensure non-PDF CAdES signatures always have `signingTime` set.
- Fix and improve timestamp summary reporting
- Corrected TrueType subtype handling
- Properly set [*ts_validation_paths*](#)

- Gracefully deal with unsupported certificate types in CMS
- Ensure attribute inspection internals can deal with `SignerInfo` without `signedAttrs`.

4.8 0.10.0

Release date: 2021-11-28

4.8.1 Dependency changes

- Update `pyhanko-certvalidator` to 0.18.0
- Update `aiohttp` to 3.8.0 (optional dependency)
- Introduce `python-pae==0.1.0` (tests)

4.8.2 New features and enhancements

Signing

- There's a new *Signer* implementation that allows pyHanko to be used with remote signing services that implement the Cloud Signature Consortium API. Since auth handling differs from vendor to vendor, using this feature requires still the caller to supply an authentication handler implementation; see `pyhanko.sign.signers.csc_signer` for more information. *This feature is currently incubating.*

Validation

- Add CLI option to skip diff analysis.
- Add CLI flag to disable strict syntax checks.
- Use chunked digests while validating.
- Improved difference analysis logging.

Miscellaneous

- Better handling of nonexistent objects: clearer errors in strict mode, better fallback behaviour in nonstrict mode. This applies to both regular object dereferencing and xref history analysis.
- Added many new tests for various edge cases, mainly in validation code.
- Added Python 3.9 and Python 3.10 classifiers to distribution.

4.8.3 Bugs fixed

- Fix bug in output handler in timestamp updater that caused empty output in some configurations.
- Fix a config parsing error when no stamp styles are defined in the configuration file.

4.9 0.9.0

Release date: 2021-10-31

4.9.1 Dependency changes

- Update `pyhanko-certvalidator` to 0.17.3
- Update `fontTools` to 4.27.1
- Update `certomancer` to 0.6.0 (tests)
- Introduce `pytest-aiohttp` ~0.3.0 and `aiohttp` >=3.7.4 (tests)

4.9.2 API-breaking changes

This is a pretty big release, with a number of far-reaching changes in the lower levels of the API that may cause breakage. Much of pyHanko's internal logic has been refactored to prefer asynchronous I/O wherever possible (`pyhanko-certvalidator` was also refactored accordingly). Some compromises were made to allow non-async-aware code to continue working as-is.

If you'd like a quick overview of how you can take advantage of the new asynchronous library functions, take a look at [this section in the signing docs](#).

Here's an overview of low-level functionality that changed:

- CMS signing logic was refactored and made asynchronous (only relevant if you implemented your own custom signers)
- Time stamp client API was refactored and made asynchronous (only relevant if you implemented your own time stamping clients)
- The *interrupted signing* workflow now involves more `asyncio` as well.
- `perform_presign_validation()` was made asynchronous.
- `prepare_tbs_document()`: the `bytes_reserved` parameter is mandatory now.
- `post_signature_processing()` was made asynchronous.
- `collect_validation_info()` was made asynchronous

Other functions have been deprecated in favour of asynchronous equivalents; such deprecations are documented in [the API reference](#). The section on extending *Signer* has also been updated.

Warning: Even though we have pretty good test coverage, due to the volume of changes, some instability may ensue. Please do not hesitate to report bugs on [the issue tracker](#)!

4.9.3 New features and enhancements

Signing

- Async-first signing API
- Relax `token-label` requirements in PKCS#11 config, allowing `slot-no` as an alternative
- Allow selecting keys and certificates by ID in the PKCS#11 signer
- Allow the signer's certificate to be sourced from a file in the PKCS#11 signer
- Allow BeID module path to be specified in config
- Tweak cert querying logic in PKCS#11 signer
- Add support for raw ECDSA to the PKCS#11 signer
- Basic DSA support (for completeness w.r.t. ISO 32000)
- Choose a default message digest more cleverly, based on the signing algorithm and key size
- Fail loudly when trying to add a certifying signature to an already-signed document using the high-level signing API
- Provide a flag to skip embedding root certificates

Validation

- Async-first validation API
- Use non-zero exit code on failed CLI validation

Miscellaneous

- Minor reorganisation of `config.py` functions
- Move PKCS#11 pin prompt logic to `cli.py`
- Improve font embedding efficiency (better stream management)
- Ensure idempotence of object stream flushing
- Improve PKCS#11 signer logging
- Make `stream_xrefs=False` by default in `copy_into_new_writer()`
- Removed a piece of fallback logic for `md_algorithm` that relied on obsolete parts of the standard
- Fixed a number of issues related to unexpected cycles in PDF structures

4.9.4 Bugs fixed

- Treat ASCII form feed (`\f`) as PDF whitespace
- Fix a corner case with null incremental updates
- Fix some font compatibility issues (relax assumptions about the presence of certain tables/entries)
- Be more tolerant when parsing name objects
- Correct some issues related to DSS update validation
- Correct `pdf_date()` output for negative UTC offsets

4.10 0.8.0

Release date: 2021-08-23

4.10.1 Dependency changes

- Update `pyhanko-certvalidator` to 0.16.0.

4.10.2 API-breaking changes

Some fields and method names in the config API misspelled `pkcs11`` as ``pcks11`. This has been corrected in this release. This is unlikely to cause issues for library users (since the config API is primarily used by the CLI code), but it's a breaking change all the same. If you do have code that relies on the config API, simply substituting `s/pcks/pkcs/g` should fix things.

4.10.3 New features and enhancements

Signing

- Make certificate fetching in the PKCS#11 signer more flexible.
 - Allow passing in the signer's certificate from outside the token.
 - Improve certificate registry initialisation.
- Give more control over updating the DSS in complex signature workflows. By default, pyHanko now tries to update the DSS in the revision that adds a document timestamp, after the signature (if applicable). In the absence of a timestamp, the old behaviour persists.
- Added a flag to (attempt to) produce CMS signature containers without any padding.
- Use `signing-certificate-v2` instead of `signing-certificate` when producing signatures.
- Default to empty appearance streams for empty signature fields.
- Much like the `pkcs11-setups` config entry, there are now `pemder-setups` and `pkcs12-setups` at the top level of pyHanko's config file. You can use those to store arguments for the `pemder` and `pkcs12` subcommands of pyHanko's `addsig` command, together with passphrases for non-interactive use. See *Named setups for on-disk key material*.

Validation

- Enforce the end-entity cert constraint imposed by the `signing-certificate` or `signing-certificate-v2` attribute (if present).
- Improve issuer-serial matching logic.
- Improve CMS attribute lookup routines.

Encryption

- Add a flag to suppress creating “legacy compatibility” entries in the encryption dictionary if they aren’t actually required or meaningful (for now, this only applies to `/Length`).

Miscellaneous

- Lazily load the version entry in the catalog.
- Minor internal I/O handling improvements.
- Allow constructing an `IncrementalPdfFileWriter` from a `PdfFileReader` object.
- Expose common API to modify (most) trailer entries.
- Automatically recurse into all configurable fields when processing configuration data.
- Replace some certificate storage/indexing classes by references to their corresponding classes in `pyhanko-certvalidator`.

4.10.4 Bugs fixed

- Add `/NeedAppearances` in the AcroForm dictionary to the whitelist for incremental update analysis.
- Fixed several bugs related to difference analysis on encrypted files.
- Improve behaviour of dev extensions in difference analysis.
- Fix encoding issues with `SignedDigestAlgorithm`, in particular ensuring that the signature mechanism encodes the relevant digest when using ECDSA.
- Process passfile contents more robustly in the CLI.
- Correct timestamp revinfo fetching (by ensuring that a dummy response is present)

4.11 0.7.0

Release date: 2021-07-25

4.11.1 Dependency changes

Warning: If you used OTF/TTF fonts with pyHanko prior to the 0.7.0 release, you'll need HarfBuzz going forward. Install pyHanko with the [opentype] optional dependency group to grab everything you need.

- Update pyhanko-certvalidator to 0.15.3
- TrueType/OpenType support moved to new optional dependency group labelled [opentype].
 - Dependency on fontTools moved from core dependencies to [opentype] group.
 - We now use HarfBuzz (uharfbuzz==0.16.1) for text shaping with OTF/TTF fonts.

4.11.2 API-breaking changes

Warning: If you use any of pyHanko's lower-level APIs, review this section carefully before updating.

Signing code refactor

This release includes a refactor of the `pyhanko.sign.signers` module into a *package* with several submodules. The original API exposed by this module is reexported in full at the package level, so existing code using pyHanko's publicly documented signing APIs *should* continue to work **without modification**.

There is one notable exception: as part of this refactor, the low-level *PdfCMSEmbedder* protocol was tweaked slightly, to support the new interrupted signing workflow (see below). The required changes to existing code should be minimal; have a look at *the relevant section* in the library documentation for a concrete description of the changes, and an updated usage example.

In addition, if you extended the *PdfSigner* class, then you'll have to adapt to the new internal signing workflow as well. This may be tricky due to the fact that the separation of concerns between different steps in the signing process is now enforced more strictly. I'm not aware of use cases requiring *PdfSigner* to be extended, but if you're having trouble migrating your custom subclass to the new API structure, feel free to open *an issue*. Merely having subclassed *Signer* shouldn't require you to change anything.

Fonts

The low-level font loading API has been refactored to make font resource handling less painful, to provide smoother HarfBuzz integration and to expose more OpenType tweaks in the API.

To this end, the old `pyhanko.pdf_utils.font` module was turned into a package containing three modules: *api*, *basic* and *opentype*. The *api* module contains the definitions for the general *FontEngine* and *FontEngineFactory* classes, together with some other general plumbing logic. The *basic* module provides a minimalist implementation with a (non-embedded) monospaced font. If you need TrueType/OpenType support, you'll need the *opentype* module together with the optional dependencies in the [opentype] dependency group (currently *fontTools* and *uharfbuzz*, see above). Take a look at the section for `pyhanko.pdf_utils.font` in *the API reference documentation* for further details.

For the time being, there are no plans to support embedding **Type1** fonts, or to offer support for **Type3** fonts at all.

Miscellaneous

- The `content_stream` parameter was removed from `import_page_as_xobject()`. Content streams are now merged automatically, since treating a page content stream array non-atomically is a bad idea.
- `PdfSigner` is no longer a subclass of `PdfTimeStamper`.

4.11.3 New features and enhancements

Signing

- *Interrupted signing* workflow: segmented signing workflow that can be interrupted partway through and resumed later (possibly in a different process or on a different machine). Useful for dealing with signing processes that rely on user interaction and/or remote signing services.
- *Generic data signing* support: construct CMS `signedData` objects for arbitrary data (not necessarily for use in PDF signature fields).
- Experimental API for signing individual embedded files (nonstandard).
- PKCS#11 settings can now be set in the configuration file.

Validation

- Add support for validating CMS `signedData` structures against arbitrary payloads (see also: *Generic data signing*)
- Streamline CMS timestamp validation.
- Support reporting on (CAvES) content timestamps in addition to signature timestamps.
- Allow signer certificates to be identified by the `subjectKeyIdentifier` extension.

Encryption

- Support granular crypt filters for embedded files
- Add convenient API to encrypt and wrap a PDF document as a binary blob. The resulting file will open as usual in a viewer that supports PDF collections; a fallback page with alternative instructions is shown otherwise.

Miscellaneous

- Complete overhaul of appearance generation & layout system. Most of these changes are internal, except for some font loading mechanics (see above). All use of OpenType / TrueType fonts now requires the `[opentype]` optional dependency group. New features:
 - Use HarfBuzz for shaping (incl. complex scripts)
 - Support TrueType fonts and OpenType fonts without a CFF table.
 - Support vertical writing (among other OpenType features).
 - Use `ActualText` marked content in addition to `ToUnicode`.
 - Introduce simple box layout & alignment rules, and apply them uniformly across all layout decisions where possible. See `pyhanko.stamp` and `pyhanko.pdf_utils.layout` for API documentation.
- Refactored stamp style dataclass hierarchy. This should not affect existing code.

- Allow externally generated PDF content to be used as a stamp appearance.
- Utility API for embedding files into PDF documents.
- Added support for PDF developer extension declarations.

4.11.4 Bugs fixed

Signing

- Declare ESIC extension when producing a PAdES signature on a PDF 1.x file.

Validation

- Fix handling of orphaned objects in diff analysis.
- Tighten up tolerances for (visible) signature field creation.
- Fix typo in `BaseFieldModificationRule`
- Deal with some VRI-related corner cases in the DSS diffing logic.

Encryption

- Improve identity crypt filter behaviour when applied to text strings.
- Correct handling of non-default public-key crypt filters.

Miscellaneous

- Promote stream manipulation methods to base writer.
- Correct some edge cases w.r.t. PDF content import
- Use floats for `MediaBox`.
- Handle escapes in PDF name objects.
- Correct `ToUnicode CMap` formatting.
- Do not close over GSUB when computing font subsets.
- Fix `output_version` handling oversight.
- Misc. export list & type annotation corrections.

4.12 0.6.1

Release date: 2021-05-22

4.12.1 Dependency changes

- Update `pyhanko-certvalidator` to 0.15.2
- Replace constraint on `certomancer` and `pyhanko-certvalidator` by soft minor version constraint (`~=`)
- Set version bound for `freezegun`

4.12.2 Bugs fixed

- Add `/Q` and `/DA` keys to the whitelist for incremental update analysis on form fields.

4.13 0.6.0

Release date: 2021-05-15

4.13.1 Dependency changes

Warning: pyHanko’s 0.6.0 release includes quite a few changes to dependencies, some of which may break compatibility with existing code. Review this section carefully before updating.

The `pyhanko-certvalidator` dependency was updated to 0.15.1. This update adds support for name constraints, RSASSA-PSS and EdDSA for the purposes of X.509 path validation, OCSP checking and CRL validation.

Warning: Since `pyhanko-certvalidator` has considerably diverged from “mainline” `certvalidator`, the Python package containing its modules was also renamed from `certvalidator` to `pyhanko_certvalidator`, to avoid potential namespace conflicts down the line. You should update your code to reflect this change.

Concretely,

```
from certvalidator import ValidationContext
```

turns into

```
from pyhanko_certvalidator import ValidationContext
```

in the new release.

There were several changes to dependencies with native binary components:

- The `Pillow` dependency has been relaxed to `>=7.2.0`, and is now optional. The same goes for `python-barcode`. Image & 1D barcode support now needs to be installed explicitly using the `[image-support]` installation parameter.
- `PKCS#11` support has also been made optional, and can be added using the `[pkcs11]` installation parameter.

The test suite now makes use of `Certomancer`. This also removed the dependency on `ocspbuilder`.

4.13.2 New features and enhancements

Signing

- Make preferred hash inference more robust.
- Populate /AP when creating an empty visible signature field (necessary in PDF 2.0)

Validation

- Timestamp and DSS handling tweaks:
 - Preserve OCSP resps / CRLs from validation kwargs when reading the DSS.
 - Gracefully process revisions that don't have a DSS.
 - When creating document timestamps, the `validation_context` parameter is now optional.
- Enforce `certvalidator`'s `weak_hash_algos` when validating PDF signatures as well. Previously, this setting only applied to certificate validation. By default, MD5 and SHA-1 are considered weak (for digital signing purposes).
- Expose `DocTimeStamp/Sig` distinction in a more user-friendly manner.
 - The `sig_object_type` property on `EmbeddedPdfSignature` now returns the signature's type as a PDF name object.
 - `PdfFileReader` now has two extra convenience properties named `embedded_regular_signatures` and `embedded_timestamp_signatures`, that return a list of all regular signatures and document timestamps, respectively.

Encryption

- Refactor internal APIs in pyHanko's security handler implementation to make them easier to extend. Note that while anyone is free to register their own crypt filters for whatever purpose, pyHanko's security handler is still considered internal API, so behaviour is subject to change between minor version upgrades (even after 1.0.0).

Miscellaneous

- Broaden the scope of `--soft-revocation-check`.
- Corrected a typo in the signature of `validate_sig_integrity`.
- Less opaque error message on missing PKCS#11 key handle.
- Ad-hoc hash selection now relies on `pyca/cryptography` rather than `hashlib`.

4.13.3 Bugs fixed

- Correct handling of DocMDP permissions in approval signatures.
- Refactor & correct handling of SigFlags when signing prepared form fields in unsigned files.
- Fixed issue with trailing whitespace and/or NUL bytes in array literals.
- Corrected the export lists of various modules.

4.14 0.5.1

Release date: 2021-03-24

4.14.1 Bugs fixed

- Fixed a packaging blunder that caused an import error on fresh installs.

4.15 0.5.0

Release date: 2021-03-22

4.15.1 Dependency changes

Update `pyhanko-certvalidator` dependency to `0.13.0`. Dependency on `cryptography` is now mandatory, and `oscrypto` has been marked optional. This is because we now use the `cryptography` library for all signing and encryption operations, but some cryptographic algorithms listed in the PDF standard are not available in `cryptography`, so we rely on `oscrypto` for those. This is only relevant for the *decryption* of files encrypted with a public-key security handler that uses DES, triple DES or RC2 to encrypt the key seed.

In the public API, we exclusively work with `asn1crypto` representations of ASN.1 objects, to remain as backend-independent as possible.

Note: While `oscrypto` is listed as optional in pyHanko's dependency list, it is still required in practice, since `pyhanko-certvalidator` depends on it.

4.15.2 New features and enhancements

Encryption

- Enforce `keyEncipherment` key extension by default when using public-key encryption
- Show a warning when signing a document using public-key encryption through the CLI. We currently don't support using separate encryption credentials in the CLI, and using the same key pair for decryption and signing is bad practice.
- Several minor CLI updates.

Signing

- Allow customisation of key usage requirements in signer & validator, also in the CLI.
- Actively preserve document timestamp chain in new PAdES-LTA signatures.
- Support setups where fields and annotations are separate (i.e. unmerged).
- Set the lock bit in the annotation flags by default.
- Tolerate signing fields that don't have any annotation associated with them.
- Broader support for PAdES / CAdES signed attributes.

Validation

- Support validating PKCS #7 signatures that don't use `signedAttrs`. Nowadays, those are rare in the wild, but there's at least one common commercial PDF library that outputs such signatures by default (vendor name redacted to protect the guilty).
- **Timestamp-related fixes:**
 - Improve signature vs. document timestamp handling in the validation CLI.
 - Improve & test handling of malformed signature dictionaries in PDF files.
 - Align document timestamp updating logic with validation logic.
 - Correct key usage check for time stamp validation.
- Allow customisation of key usage requirements in signer & validator, also in the CLI.
- Allow LTA update function to be used to start the timestamp chain as well as continue it.
- Tolerate indirect references in signature reference dictionaries.
- Improve some potential ambiguities in the PAdES-LT and PAdES-LTA validation logic.
- **Revocation info handling changes:**
 - Support “retroactive” mode for revocation info (i.e. treat revocation info as valid in the past).
 - Added functionality to append current revocation information to existing signatures.
 - Related CLI updates.

Miscellaneous

- Some key material loading functions were cleaned up a little to make them easier to use.
- I/O tweaks: use chunked writes with a fixed buffer when copying data for an incremental update
- Warn when revocation info is embedded with an offline validation context.
- Improve SV validation reporting.

4.15.3 Bugs fixed

- Fix issue with `/Certs` not being properly dereferenced in the DSS (#4).
- Fix loss of precision on `FloatObject` serialisation (#5).
- Add missing dunder methods to `BooleanObject`.
- Do not use `.dump()` with `force=True` in validation.
- Corrected digest algorithm selection in timestamp validation.
- Correct handling of writes with empty user password.
- Do not automatically add xref streams to the object cache. This avoids a class of bugs with some kinds of updates to files with broken xref streams.
- Due to a typo, the `/Annots` array of a page would not get updated correctly if it was an indirect object. This has been corrected.

4.16 0.4.0

Release date: 2021-02-14

4.16.1 New features and enhancements

Encryption

- Expose permission flags outside security handler
- Make file encryption key straightforward to grab

Signing

- Mildly refactor `PdfSignedData` for non-signing uses
- **Make DSS API more flexible**
 - Allow direct input of cert/ocsp/CRL objects as opposed to only certvalidator output
 - Allow input to not be associated with any concrete VRI.
- **Greatly improved PKCS#11 support**
 - Added support for RSASSA-PSS and ECDSA.
 - Added tests for RSA functionality using SoftHSMv2.
 - Added a command to the CLI for generic PKCS#11.
 - *Note:* Tests don't run in CI, and ECDSA is not included in the test suite yet (SoftHSMv2 doesn't seem to expose all the necessary mechanisms).
- Factor out `unsigned_attrs` in signer, added a `digest_algorithm` parameter to `signed_attrs`.
- Allow signing with any `BasePdfFileWriter` (in particular, this allows creating signatures in the initial revision of a PDF file)
- Add `CMSAlgorithmProtection` attribute when possible * *Note:* Not added to PAdES signatures for the time being.

- Improved support for deep fields in the form hierarchy (arguably orthogonal to the standard, but it doesn't hurt to be flexible)

Validation

- **Path handling improvements:**
 - Paths in the structure tree are also simplified.
 - Paths can be resolved relative to objects in a file.
- **Limited support for tagged PDF in the validator.**
 - Existing form fields can be filled in without tripping up the modification analysis module.
 - Adding new form fields to the structure tree after signing is not allowed for the time being.
- **Internal refactoring in CMS validation logic:**
 - Isolate cryptographic integrity validation from trust validation
 - Rename *externally_invalid* API parameter to *encap_data_invalid*
 - Validate *CMSAlgorithmProtection* when present.
- Improved support for deep fields in the form hierarchy (arguably orthogonal to the standard, but it doesn't hurt to be flexible).
- Added

Miscellaneous

- Export *copy_into_new_writer*.
- Transparently handle non-seekable output streams in the signer.
- Remove unused *__iadd__* implementation from VRI class.
- Clean up some corner cases in *container_ref* handling.
- Refactored *SignatureFormField* initialisation (internal API).

4.16.2 Bugs fixed

- Deal with some XRef processing edge cases.
- Make *signed_revision* on embedded signatures more robust.
- Fix an issue where DocTimeStamp additions would trigger */All*-type field locks.
- Fix some issues with *modification_level* handling in validation status reports.
- Fix a few logging calls.
- Fix some minor issues with signing API input validation logic.

4.17 0.3.0

Release date: 2021-01-26

4.17.1 New features and enhancements

Encryption

- Reworked internal crypto API.
- Added support for PDF 2.0 encryption.
- Added support for public key encryption.
- Got rid of the homegrown *RC4* class (not that it matters all to much, *RC4* isn't secure anyhow); all cryptographic operations in *crypt.py* are now delegated to *oscrypto*.

Signing

- Encrypted files can now be signed from the CLI.
- With the optional *cryptography* dependency, pyHanko can now create RSASSA-PSS signatures.
- Factored out a low-level *PdfCMSEmbedder* API to cater to remote signing needs.

Miscellaneous

- The document ID can now be accessed more conveniently.
- The version number is now single-sourced in *version.py*.
- Initialising the page tree in a *PdfFileWriter* is now optional.
- Added a convenience function for copying files.

Validation

- With the optional *cryptography* dependency, pyHanko can now validate RSASSA-PSS signatures.
- Difference analysis checker was upgraded with capabilities to handle multiply referenced objects in a more straightforward way. This required API changes, and it comes at a significant performance cost, but the added cost is probably justified. The changes to the API are limited to the *diff_analysis* module itself, and do not impact the general validation API whatsoever.

4.17.2 Bugs fixed

- Allow */DR* and */Version* updates in diff analysis
- Fix revision handling in *trailer.flatten()*

4.18 0.2.0

Release date: 2021-01-10

4.18.1 New features and enhancements

Signing

- Allow the caller to specify an output stream when signing.

Validation

- The incremental update analysis functionality has been heavily refactored into something more rule-based and modular. The new difference analysis system is also much more user-configurable, and a (sufficiently motivated) library user could even plug in their own implementation.
- The new validation system treats `/Metadata` updates more correctly, and fixes a number of other minor stability problems.
- Improved validation logging and status reporting mechanisms.
- Improved seed value constraint enforcement support: this includes added support for `/V`, `/MDP`, `/LockDocument`, `/KeyUsage` and (passive) support for `/AppearanceFilter` and `/LegalAttestation`.

CLI

- You can now specify negative page numbers on the command line to refer to the pages of a document in reverse order.

General PDF API

- Added convenience functions to retrieve references from dictionaries and arrays.
- Tweaked handling of object freeing operations; these now produce PDF `null` objects instead of (Python) `None`.

4.18.2 Bugs fixed

- `root_ref` now consistently returns a `Reference` object
- Corrected wrong usage of `@freeze_time` in tests that caused some failures due to certificate expiry issues.
- Fixed a gnarly caching bug in `HistoricalResolver` that sometimes leaked state from later revisions into older ones.
- Prevented cross-reference stream updates from accidentally being saved with the same settings as their predecessor in the file. This was a problem when updating files generated by other PDF processing software.

4.19 0.1.0

Release date: 2020-12-30

Initial release.

FREQUENTLY ASKED QUESTIONS (FAQ)

Read these before filing bug reports.

5.1 Errors and other unexpected behaviour

5.1.1 I'm getting an error about hybrid reference files when trying to sign / validate a file. What gives?

This is explained in the [release notes](#) for version 0.12.0.

Hybrid reference files were introduced as a transitional compatibility measure between PDF 1.4 and PDF 1.5. Since PDF 1.5 support is all but universal now, they're no longer useful these days, and therefore relatively rare. Nevertheless, some tools still routinely generate such files.

Prior to 0.12.0, pyHanko would actually not process hybrid files correctly and would sometimes even accidentally corrupt them. That bug has been fixed, but there's more to it than that. The problem with hybrid files is that *by design* there's no single unambiguous way to parse them, which makes them inherently less secure than non-hybrid PDFs. That's a problem when dealing with document signatures, and also the reason why pyHanko 0.12.0 makes hybrid files an "opt-in" feature: you have to disable strict parsing mode to be able to use them.

For API users, that means passing `strict=False` to any [IncrementalPdfFileWriter](#) or [PdfFileReader](#) objects that could touch hybrid files.

For CLI users, there's the `--no-strict-syntax` switch, which is available for both signing and validation subcommands.

5.1.2 Why am I getting path building errors?

There are many reasons why path building could fail, but the most common ones are

- missing intermediate certificates that pyHanko is not aware of;
- a certificate pathing up to a root that is not a trust anchor.

In either case, you probably need to review your [validation context settings](#).

KNOWN ISSUES

This page lists some TODOs and known limitations of pyHanko.

- Expand, polish and rigorously test the validation functionality. The test suite covers a variety of scenarios already, but the difference checker in particular is still far from perfect.
- LTV validation was implemented ad-hoc, and likely does not fully adhere to the PAdES specification. This will require some effort to implement correctly. In the meantime, you should treat the result as a pyHanko-specific interpretation of the validity of the chain of trust based on the validation info present in the file, not as a final judgment on whether the signature complies with any particular PAdES profile.
- The most lenient document modification policy (i.e. addition of comments and annotations) is not supported. Comments added to a signed PDF will therefore be considered “unsafe” changes, regardless of the policy set by the signer.
- There is currently no explicit support for signing and stamping PDF/A and PDF/UA files. That is to say, pyHanko treats these as any other PDF file and will produce output that may not comply with the provisions of these standards. As of 0.14.0, it is possible to generate compliant output using pyHanko in most cases, but pyHanko itself will not attempt to enforce any additional restrictions.
- CLI support for signing files encrypted using PDF’s public-key encryption functionality is limited.

LICENSES

7.1 pyHanko License

MIT License

Copyright (c) 2020-2022 Matthias Valvekens

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

7.2 Original PyPDF2 license

This package contains various elements based on code from the [PyPDF2](#) project, of which we reproduce the license below.

This package contains various elements based on code from the PyPDF2 project, of which we reproduce the license below.

Copyright (c) 2006-2008, Mathieu Fenniak

Some contributions copyright (c) 2007, Ashish Kulkarni <kulkarni.ashish@gmail.com>

Some contributions copyright (c) 2014, Steve Witham <switham_github@mac-guyver.com>

(continues on next page)

(continued from previous page)

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

p

- pyhanko.config, 248
- pyhanko.pdf_utils.barcodes, 53
- pyhanko.pdf_utils.config_utils, 54
- pyhanko.pdf_utils.content, 55
- pyhanko.pdf_utils.crypt, 57
- pyhanko.pdf_utils.embed, 76
- pyhanko.pdf_utils.extensions, 79
- pyhanko.pdf_utils.filters, 81
- pyhanko.pdf_utils.font.api, 83
- pyhanko.pdf_utils.font.basic, 85
- pyhanko.pdf_utils.font.opentype, 86
- pyhanko.pdf_utils.generic, 88
- pyhanko.pdf_utils.images, 99
- pyhanko.pdf_utils.incremental_writer, 100
- pyhanko.pdf_utils.layout, 102
- pyhanko.pdf_utils.metadata.model, 106
- pyhanko.pdf_utils.misc, 112
- pyhanko.pdf_utils.reader, 114
- pyhanko.pdf_utils.rw_common, 118
- pyhanko.pdf_utils.text, 120
- pyhanko.pdf_utils.writer, 121
- pyhanko.pdf_utils.xref, 127
- pyhanko.sign.adcs.api, 132
- pyhanko.sign.adcs.asn1_util, 134
- pyhanko.sign.adcs.cades_asn1, 134
- pyhanko.sign.adcs.report, 135
- pyhanko.sign.attributes, 137
- pyhanko.sign.beid, 139
- pyhanko.sign.diff_analysis.common, 141
- pyhanko.sign.diff_analysis.form_rules_api, 142
- pyhanko.sign.diff_analysis.policies, 144
- pyhanko.sign.diff_analysis.policy_api, 146
- pyhanko.sign.diff_analysis.rules.file_structure_rules, 148
- pyhanko.sign.diff_analysis.rules.form_field_rules, 149
- pyhanko.sign.diff_analysis.rules.metadata_rules, 151
- pyhanko.sign.diff_analysis.rules_api, 152
- pyhanko.sign.fields, 153
- pyhanko.sign.general, 165
- pyhanko.sign.pkcs11, 170
- pyhanko.sign.signers.cms_embedder, 173
- pyhanko.sign.signers.constants, 182
- pyhanko.sign.signers.csc_signer, 176
- pyhanko.sign.signers.functions, 183
- pyhanko.sign.signers.pdf_byterange, 184
- pyhanko.sign.signers.pdf_cms, 187
- pyhanko.sign.signers.pdf_signer, 198
- pyhanko.sign.timestamps.aiohttp_client, 218
- pyhanko.sign.timestamps.api, 216
- pyhanko.sign.timestamps.common_utils, 220
- pyhanko.sign.timestamps.dummy_client, 219
- pyhanko.sign.timestamps.requests_client, 219
- pyhanko.sign.validation, 244
- pyhanko.sign.validation.dss, 220
- pyhanko.sign.validation.errors, 224
- pyhanko.sign.validation.generic_cms, 225
- pyhanko.sign.validation.ltv, 230
- pyhanko.sign.validation.pdf_embedded, 232
- pyhanko.sign.validation.settings, 236
- pyhanko.sign.validation.status, 238
- pyhanko.sign.validation.utils, 244
- pyhanko.stamp, 253

INDEX

A

- ABOVE_TEXT (pyhanko.stamp.QRPosition attribute), 257
- ac_attrs (pyhanko.sign.validation.status.StandardCMSSignatureStatus attribute), 241
- ac_validation_context (pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute), 201
- ac_validation_errs (pyhanko.sign.validation.status.CAdESSignerAttributeAssertions attribute), 240
- ac_validation_errs (pyhanko.sign.validation.status.StandardCMSSignatureStatus attribute), 241
- ac_validation_paths (pyhanko.sign.signers.pdf_signer.PreSignValidationStatus attribute), 214
- access_on() (pyhanko.pdf_utils.reader.RawPdfPath method), 118
- access_reference_on() (pyhanko.pdf_utils.reader.RawPdfPath method), 118
- action (pyhanko.sign.fields.FieldMDPSpec attribute), 162
- add_content_to_page() (pyhanko.pdf_utils.writer.BasePdfFileWriter method), 125
- add_crypt_filter() (pyhanko.pdf_utils.generic.StreamObject method), 95
- add_dss() (pyhanko.sign.validation.dss.DocumentSecurityStore class method), 222
- add_object() (pyhanko.pdf_utils.writer.BasePdfFileWriter method), 123
- add_object() (pyhanko.pdf_utils.xref.ObjectStream method), 132
- add_recipients() (pyhanko.pdf_utils.crypt.PubKeyCryptFilter method), 69
- add_recipients() (pyhanko.pdf_utils.crypt.PubKeySecurityHandler method), 64
- ADD_REV_INFO (pyhanko.sign.fields.SigSeedValFlags attribute), 155
- add_rev_info (pyhanko.sign.fields.SigSeedValueSpec attribute), 158
- add_stream_to_page() (pyhanko.pdf_utils.writer.BasePdfFileWriter method), 124
- add_subset() (pyhanko.pdf_utils.font.api.FontSubsetCollection method), 84
- add_trailer_revision() (pyhanko.pdf_utils.xref.TrailerDictionary method), 131
- add_validation_info() (in module pyhanko.sign.validation), 247
- ades_status (pyhanko.sign.validation.errors.SignatureValidationError property), 224
- AdESFailure (class in pyhanko.sign.ades.report), 136
- AdESIndeterminate (class in pyhanko.sign.ades.report), 136
- AdESStatus (class in pyhanko.sign.ades.report), 135
- AdESSubIndic (class in pyhanko.sign.ades.report), 136
- ADOBE_PKCS7_DETACHED (pyhanko.sign.fields.SigSeedSubFilter attribute), 160
- adobe_revinfo_attr (pyhanko.sign.signers.pdf_cms.PdfCMSSignedAttributes attribute), 197
- adobe_revinfo_attr (pyhanko.sign.signers.pdf_signer.PreSignValidationStatus attribute), 214
- ADOBE_STYLE (pyhanko.sign.validation.ltv.RevocationInfoValidationType attribute), 230
- AdobeRevinfoProvider (class in pyhanko.sign.attributes), 138
- AES256 (pyhanko.pdf_utils.crypt.SecurityHandlerVersion attribute), 65
- AES256 (pyhanko.pdf_utils.crypt.StandardSecuritySettingsRevision attribute), 65
- AESCryptFilterMixin (class in pyhanko.pdf_utils.crypt), 72
- af_relationship (pyhanko.pdf_utils.embed.FileSpec attribute), 77
- AIOHttpTimeStamper (class in py-

`hanko.sign.timestamps.aiohttp_client)`, 218
`align()` (`pyhanko.pdf_utils.layout.AxisAlignment` method), 103
`ALIGN_MAX` (`pyhanko.pdf_utils.layout.AxisAlignment` attribute), 103
`ALIGN_MID` (`pyhanko.pdf_utils.layout.AxisAlignment` attribute), 103
`ALIGN_MIN` (`pyhanko.pdf_utils.layout.AxisAlignment` attribute), 103
`ALL` (`pyhanko.sign.fields.FieldMDPAction` attribute), 162
`allocate_placeholder()` (`pyhanko.pdf_utils.writer.BasePdfFileWriter` method), 123
`ALTERNATIVE` (`pyhanko.pdf_utils.metadata.model.XmpArrayType` attribute), 109
`alternative()` (`pyhanko.pdf_utils.metadata.model.XmpArrayType` class method), 110
`ALWAYS` (`pyhanko.pdf_utils.extensions.DevExtensionMultivalued` attribute), 79
`AnnotAppearances` (class in `pyhanko.stamp`), 253
`ANNOTATE` (`pyhanko.sign.fields.MDPPerm` attribute), 162
`ANNOTATIONS` (`pyhanko.sign.diff_analysis.policy_api.ModificationRule` attribute), 146
`api_ver` (`pyhanko.sign.signers.csc_signer.CSCServiceSession` attribute), 179
`appearance` (`pyhanko.sign.fields.SigSeedValueSpec` attribute), 158
`APPEARANCE_FILTER` (`pyhanko.sign.fields.SigSeedValFlags` attribute), 156
`appearance_setup` (`pyhanko.sign.signers.cms_embedder.SigObjSetup` attribute), 175
`append_signature_field()` (in module `pyhanko.sign.fields`), 164
`apply()` (`pyhanko.sign.diff_analysis.form_rules_api.FieldMDPRule` method), 143
`apply()` (`pyhanko.sign.diff_analysis.form_rules_api.FormUpdatingRule` method), 143
`apply()` (`pyhanko.sign.diff_analysis.policies.StandardDiffPolicy` method), 145
`apply()` (`pyhanko.sign.diff_analysis.policy_api.DiffPolicy` method), 147
`apply()` (`pyhanko.sign.diff_analysis.rules.file_structure_rules.BaseFileStructureRule` method), 148
`apply()` (`pyhanko.sign.diff_analysis.rules.file_structure_rules.XmpSubjectRule` method), 148
`apply()` (`pyhanko.sign.diff_analysis.rules.form_field_rules.BaseFieldModificationRule` method), 150
`apply()` (`pyhanko.sign.diff_analysis.rules.form_field_rules.DSSComplianceRule` method), 149
`apply()` (`pyhanko.sign.diff_analysis.rules.form_field_rules.SigFieldCreationRule` method), 149
`apply()` (`pyhanko.sign.diff_analysis.rules.metadata_rules.DocInfoRule` method), 151
`apply()` (`pyhanko.sign.diff_analysis.rules.metadata_rules.MetadataUpdateRule` method), 152
`apply()` (`pyhanko.sign.diff_analysis.rules_api.WhitelistRule` method), 153
`apply()` (`pyhanko.sign.signers.cms_embedder.SigAppearanceSetup` method), 175
`apply()` (`pyhanko.sign.signers.cms_embedder.SigMDPSetup` method), 174
`apply()` (`pyhanko.stamp.BaseStamp` method), 258
`apply()` (`pyhanko.stamp.QRStamp` method), 259
`apply_adobe_revocation_info()` (in module `pyhanko.sign.validation.ltv`), 230
`apply_filter()` (`pyhanko.pdf_utils.generic.StreamObject` method), 95
`apply_qualified()` (`pyhanko.sign.diff_analysis.rules.file_structure_rules.CatalogModificationRule` method), 148
`apply_qualified()` (`pyhanko.sign.diff_analysis.rules_api.QualifiedWhitelistRule` method), 152
`apply_sig_field_spec_properties()` (in module `pyhanko.sign.fields`), 165
`array_type` (`pyhanko.pdf_utils.metadata.model.XmpArrayType` attribute), 110
`ArrayObject` (class in `pyhanko.pdf_utils.generic`), 93
`as_appearances()` (`pyhanko.stamp.BaseStamp` method), 258
`as_cert_store()` (`pyhanko.sign.signers.csc_signer.CSCCredentialInfo` method), 180
`as_cm()` (`pyhanko.pdf_utils.layout.Positioning` method), 106
`as_form_xobject()` (`pyhanko.pdf_utils.content.PdfContent` method), 106
`as_numeric()` (`pyhanko.pdf_utils.generic.FloatObject` method), 91
`as_numeric()` (`pyhanko.pdf_utils.generic.NumberObject` method), 92
`as_pdf_object()` (`pyhanko.pdf_utils.content.PdfResources` method), 56
`as_pdf_object()` (`pyhanko.pdf_utils.crypt.CryptFilter` method), 68
`as_pdf_object()` (`pyhanko.pdf_utils.crypt.CryptFilterConfiguration` method), 68
`as_pdf_object()` (`pyhanko.pdf_utils.crypt.IdentityCryptFilter` method), 70
`as_pdf_object()` (`pyhanko.pdf_utils.crypt.PubKeyCryptFilter` method), 70

<code>as_pdf_object()</code> (py- hanko.pdf_utils.crypt.PubKeySecurityHandler method), 64	<code>hanko.sign.fields.FieldMDPSpec</code> method), 163
<code>as_pdf_object()</code> (py- hanko.pdf_utils.crypt.SecurityHandler method), 59	<code>as_signing_certificate()</code> (in module py- hanko.sign.general), 169
<code>as_pdf_object()</code> (py- hanko.pdf_utils.crypt.SecurityHandlerVersion method), 65	<code>as_signing_certificate_v2()</code> (in module py- hanko.sign.general), 169
<code>as_pdf_object()</code> (py- hanko.pdf_utils.crypt.StandardCryptFilter method), 69	<code>as_transform_params()</code> (py- hanko.sign.fields.FieldMDPSpec method), 162
<code>as_pdf_object()</code> (py- hanko.pdf_utils.crypt.StandardSecurityHandler method), 62	<code>as_tuple()</code> (pyhanko.sign.validation.ltv.RevocationInfoValidationType class method), 230
<code>as_pdf_object()</code> (py- hanko.pdf_utils.crypt.StandardSecuritySettingsRevision method), 65	<code>as_validation_context()</code> (py- hanko.sign.validation.dss.DocumentSecurityStore method), 221
<code>as_pdf_object()</code> (pyhanko.pdf_utils.embed.FileSpec method), 78	<code>ASCII85Decode</code> (class in pyhanko.pdf_utils.filters), 81
<code>as_pdf_object()</code> (py- hanko.pdf_utils.extensions.DeveloperExtension method), 80	<code>ASCIISignDecode</code> (class in pyhanko.pdf_utils.filters), 82
<code>as_pdf_object()</code> (py- hanko.pdf_utils.xref.ObjectStream method), 132	<code>as_sign()</code> (pyhanko.sign.adess.api.GenericCommitment prop- erty), 133
<code>as_pdf_object()</code> (pyhanko.sign.fields.FieldMDPSpec method), 162	<code>aspect_ratio</code> (pyhanko.pdf_utils.layout.BoxConstraints property), 102
<code>as_pdf_object()</code> (py- hanko.sign.fields.SigCertConstraints method), 157	<code>aspect_ratio_defined</code> (py- hanko.pdf_utils.layout.BoxConstraints property), 103
<code>as_pdf_object()</code> (py- hanko.sign.fields.SigSeedValueSpec method), 159	<code>assert_not_stream()</code> (in module py- hanko.sign.diff_analysis.common), 142
<code>as_pdf_object()</code> (py- hanko.sign.validation.dss.DocumentSecurityStore method), 221	<code>assert_viable()</code> (py- hanko.sign.signers.pdf_signer.DSSContentSettings method), 202
<code>as_pdf_object()</code> (pyhanko.sign.validation.dss.VRI method), 220	<code>assert_viable()</code> (py- hanko.sign.signers.pdf_signer.TimestampDSSContentSettings method), 202
<code>as_pdf_object()</code> (pyhanko.stamp.AnnotAppearances method), 254	<code>assert_writable_and_random_access()</code> (in module pyhanko.pdf_utils.misc), 113
<code>as_qualified()</code> (pyhanko.sign.diff_analysis.rules_api.WholeDocumentRule method), 153	<code>async_add_validation_info()</code> (in module py- hanko.sign.validation.dss), 223
<code>as_rdf()</code> (pyhanko.pdf_utils.metadata.model.XmpArrayType method), 109	<code>async_digest_doc_for_signing()</code> (py- hanko.sign.signers.pdf_signer.PdfSigner method), 208
<code>as_resource()</code> (pyhanko.pdf_utils.font.api.FontEngine method), 84	<code>async_dummy_response()</code> (py- hanko.sign.timestamps.api.TimeStamper method), 217
<code>as_resource()</code> (pyhanko.pdf_utils.font.basic.SimpleFontEngine method), 85	<code>async_finish_signing()</code> (py- hanko.sign.signers.pdf_signer.PdfTBSDocument class method), 213
<code>as_resource()</code> (pyhanko.pdf_utils.font.opentype.GlyphAccumulator method), 87	<code>async_request_headers()</code> (py- hanko.sign.timestamps.aiohttp_client.AIOHttpTimeStamper method), 218
<code>as_set_of()</code> (in module pyhanko.sign.adess.asn1_util), 134	<code>async_request_tsa_response()</code> (py- hanko.sign.timestamps.aiohttp_client.AIOHttpTimeStamper method), 218
<code>as_sig_field_lock()</code> (py- hanko.sign.timestamps.api.TimeStamper method), 217	<code>async_request_tsa_response()</code> (py- hanko.sign.timestamps.aiohttp_client.AIOHttpTimeStamper method), 217

hanko.sign.timestamps.dummy_client.DummyTimeStamper *hanko.sign.validation.pdf_embedded*), 235
method), 219

async_request_tsa_response() (py-
hanko.sign.timestamps.requests_client.HTTPTimeStamper
method), 219

async_sign() (pyhanko.sign.signers.pdf_cms.Signer
method), 190

async_sign_general_data() (py-
hanko.sign.signers.pdf_cms.Signer *method*),
 192

async_sign_pdf() (in module py-
hanko.sign.signers.functions), 183

async_sign_pdf() (py-
hanko.sign.signers.pdf_signer.PdfSigner
method), 209

async_sign_prescribed_attributes() (py-
hanko.sign.signers.pdf_cms.Signer *method*),
 191

async_sign_raw() (py-
hanko.sign.pkcs11.PKCS11Signer *method*),
 171

async_sign_raw() (py-
hanko.sign.signers.csc_signer.CSCSigner
method), 178

async_sign_raw() (py-
hanko.sign.signers.pdf_cms.ExternalSigner
method), 197

async_sign_raw() (py-
hanko.sign.signers.pdf_cms.Signer *method*),
 189

async_sign_raw() (py-
hanko.sign.signers.pdf_cms.SimpleSigner
method), 195

async_timestamp() (py-
hanko.sign.timestamps.aiohttp_client.AIOHttpTimeStamper
method), 218

async_timestamp() (py-
hanko.sign.timestamps.api.TimeStamper
method), 217

async_timestamp_pdf() (py-
hanko.sign.signers.pdf_signer.PdfTimeStamper
method), 204

async_update_archival_timestamp_chain() (py-
hanko.sign.signers.pdf_signer.PdfTimeStamper
method), 206

async_validate_cms_signature() (in module py-
hanko.sign.validation.generic_cms), 225

async_validate_detached_cms() (in module py-
hanko.sign.validation.generic_cms), 227

async_validate_pdf_ltv_signature() (in module
pyhanko.sign.validation.ltv), 230

async_validate_pdf_signature() (in module py-
hanko.sign.validation.pdf_embedded), 234

async_validate_pdf_timestamp() (in module py-

asyncify_signer() (in module py-
hanko.sign.signers.pdf_cms), 198

attached_timestamp_data (py-
hanko.sign.validation.pdf_embedded.EmbeddedPdfSignature
property), 233

attr_type (pyhanko.sign.validation.status.X509AttributeInfo
attribute), 239

attr_values (pyhanko.sign.validation.status.X509AttributeInfo
attribute), 239

attribute_certs (py-
hanko.sign.general.SignedDataCerts *attribute*),
 167

attribute_certs (py-
hanko.sign.signers.pdf_cms.Signer *attribute*),
 188

attribute_type (pyhanko.sign.attributes.AdobeRevinfoProvider
attribute), 138

attribute_type (pyhanko.sign.attributes.CMSAlgorithmProtectionProvia
attribute), 139

attribute_type (pyhanko.sign.attributes.CMSAttributeProvider
attribute), 137

attribute_type (pyhanko.sign.attributes.SigningCertificateV2Provider
attribute), 138

attribute_type (pyhanko.sign.attributes.SigningTimeProvider
attribute), 138

attribute_type (pyhanko.sign.attributes.TSTProvider
attribute), 139

auth_headers (pyhanko.sign.signers.csc_signer.CSCAuthorizationManag
property), 181

auth_headers (pyhanko.sign.signers.csc_signer.CSCServiceSessionInfo
property), 179

authenticate() (pyhanko.pdf_utils.crypt.PubKeyCryptFilter
method), 69

authenticate() (pyhanko.pdf_utils.crypt.PubKeySecurityHandler
method), 64

authenticate() (pyhanko.pdf_utils.crypt.SecurityHandler
method), 60

authenticate() (pyhanko.pdf_utils.crypt.StandardSecurityHandler
method), 62

author (pyhanko.pdf_utils.metadata.model.DocumentMetadata
attribute), 107

author_sig (pyhanko.sign.validation.pdf_embedded.DocMDPInfo
property), 234

authorize_signature() (py-
hanko.sign.signers.csc_signer.CSCAuthorizationManager
method), 180

authorize_signature() (py-
hanko.sign.signers.csc_signer.PrefetchedSADAAuthorizationManag
method), 182

AuthResult (class in pyhanko.pdf_utils.crypt), 64

AuthStatus (class in pyhanko.pdf_utils.crypt), 65

autodetected_encoding (py-
hanko.pdf_utils.generic.TextStringObject

- attribute*), 92
- AxisAlignment** (*class in pyhanko.pdf_utils.layout*), 103
- ## B
- background** (*pyhanko.stamp.BaseStampStyle attribute*), 254
- background_layout** (*pyhanko.stamp.BaseStampStyle attribute*), 254
- background_opacity** (*pyhanko.stamp.BaseStampStyle attribute*), 254
- background_opacity** (*pyhanko.stamp.StaticStampStyle attribute*), 257
- BarcodeBox** (*class in pyhanko.pdf_utils.barcodes*), 53
- base_postscript_name** (*pyhanko.pdf_utils.font.api.FontSubsetCollection attribute*), 84
- base_version** (*pyhanko.pdf_utils.extensions.DeveloperExtension attribute*), 80
- BaseFieldModificationRule** (*class in pyhanko.sign.diff_analysis.rules.form_field_rules*), 150
- BasePdfFileWriter** (*class in pyhanko.pdf_utils.writer*), 121
- BaseStamp** (*class in pyhanko.stamp*), 258
- BaseStampStyle** (*class in pyhanko.stamp*), 254
- beid_module_path** (*pyhanko.config.CLIConfig attribute*), 249
- BEIDSigner** (*class in pyhanko.sign.beid*), 139
- BELOW_TEXT** (*pyhanko.stamp.QRPosition attribute*), 257
- blanket_approve** (*pyhanko.sign.diff_analysis.rules_api.ReferenceUpdate attribute*), 153
- BooleanObject** (*class in pyhanko.pdf_utils.generic*), 91
- border_width** (*pyhanko.pdf_utils.text.TextBoxStyle attribute*), 120
- border_width** (*pyhanko.stamp.BaseStampStyle attribute*), 254
- bottom** (*pyhanko.pdf_utils.layout.Margins attribute*), 103
- bottom_line** (*pyhanko.sign.validation.status.PdfSignatureStatus property*), 243
- bottom_line** (*pyhanko.sign.validation.status.StandardCMSSignatureStatus property*), 242
- box** (*pyhanko.sign.fields.SigFieldSpec attribute*), 154
- box_layout_rule** (*pyhanko.pdf_utils.text.TextBoxStyle attribute*), 120
- box_out_of_bounds** (*pyhanko.sign.fields.InvisSigSettings attribute*), 164
- BoxConstraints** (*class in pyhanko.pdf_utils.layout*), 102
- BoxSpecificationError**, 102
- build()** (*pyhanko.pdf_utils.crypt.SecurityHandler static method*), 59
- build_attr_value()** (*pyhanko.sign.attributes.AdobeRevinfoProvider method*), 138
- build_attr_value()** (*pyhanko.sign.attributes.CMSAlgorithmProtectionProvider method*), 139
- build_attr_value()** (*pyhanko.sign.attributes.CMSAttributeProvider method*), 137
- build_attr_value()** (*pyhanko.sign.attributes.SigningCertificateV2Provider method*), 138
- build_attr_value()** (*pyhanko.sign.attributes.SigningTimeProvider method*), 138
- build_attr_value()** (*pyhanko.sign.attributes.TSTProvider method*), 139
- build_crypt_filter()** (*in module pyhanko.pdf_utils.crypt*), 75
- build_from_certs()** (*pyhanko.pdf_utils.crypt.PubKeySecurityHandler class method*), 63
- build_from_pw()** (*pyhanko.pdf_utils.crypt.StandardSecurityHandler class method*), 62
- build_from_pw_legacy()** (*pyhanko.pdf_utils.crypt.StandardSecurityHandler class method*), 61
- build_timestamper()** (*pyhanko.sign.fields.SigSeedValueSpec method*), 159
- bulk_fetch** (*pyhanko.config.PKCS11SignatureConfig attribute*), 252
- byte_range_digest()** (*in module pyhanko.sign.general*), 169
- ByteDot** (*pyhanko.pdf_utils.generic.NumberObject attribute*), 92
- ByteStringObject** (*class in pyhanko.pdf_utils.generic*), 92
- ## C
- cache_get_indirect_object()** (*pyhanko.pdf_utils.reader.PdfFileReader method*), 115
- cache_indirect_object()** (*pyhanko.pdf_utils.reader.PdfFileReader method*), 115
- cades_signed_attr_spec** (*pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute*), 201
- cades_signer_attrs** (*pyhanko.sign.validation.status.StandardCMSSignatureStatus attribute*), 241

CAdESSignedAttrSpec (class in pyhanko.sign.ades.api), 133
 CAdESSignerAttributeAssertions (class in pyhanko.sign.validation.status), 240
 CatalogModificationRule (class in pyhanko.sign.diff_analysis.rules.file_structure_rules), 148
 cert (pyhanko.sign.fields.SigSeedValueSpec attribute), 157
 cert_file (pyhanko.config.PemDerSignatureConfig attribute), 250
 cert_id (pyhanko.config.PKCS11SignatureConfig attribute), 252
 cert_label (pyhanko.config.PKCS11SignatureConfig attribute), 252
 cert_registry (pyhanko.sign.pkcs11.PKCS11Signer property), 171
 cert_registry (pyhanko.sign.signers.csc_signer.CSCSigner attribute), 177
 cert_registry (pyhanko.sign.signers.pdf_cms.ExternalSigner attribute), 197
 cert_registry (pyhanko.sign.signers.pdf_cms.Signer attribute), 188
 CERTIFICATE_CHAIN_GENERAL_FAILURE (pyhanko.sign.ades.report.AdESIndeterminate attribute), 136
 CertificateStore (class in pyhanko.sign.general), 166
 certified_attrs (pyhanko.sign.ades.api.SignerAttrSpec attribute), 134
 certified_attrs (pyhanko.sign.validation.status.CAdESSignerAttributeAssertions attribute), 240
 CertifiedAttributeChoices (class in pyhanko.sign.ades.cades_asn1), 135
 CertifiedAttributeInfo (class in pyhanko.sign.validation.status), 239
 CertifiedAttributes (class in pyhanko.sign.validation.status), 239
 CertifiedAttributesV2 (class in pyhanko.sign.ades.cades_asn1), 135
 certify (pyhanko.sign.signers.cms_embedder.SigMDPSetup attribute), 174
 certify (pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute), 199
 certs (pyhanko.sign.validation.dss.VRI attribute), 220
 chain (pyhanko.sign.signers.csc_signer.CSCCredentialInfo attribute), 179
 CHAIN_CONSTRAINTS_FAILURE (pyhanko.sign.ades.report.AdESIndeterminate attribute), 136
 changed_form_fields (pyhanko.sign.diff_analysis.policy_api.DiffResult attribute), 147
 check_config_keys() (in module pyhanko.pdf_utils.config_utils), 55
 check_config_keys() (pyhanko.config.PKCS11SignatureConfig class method), 253
 check_config_keys() (pyhanko.pdf_utils.config_utils.ConfigurableMixin class method), 54
 check_ess_certid() (in module pyhanko.sign.general), 169
 check_form_field() (pyhanko.sign.diff_analysis.rules.form_field_rules.BaseFieldModification method), 151
 check_form_field() (pyhanko.sign.diff_analysis.rules.form_field_rules.GenericFieldModification method), 150
 check_form_field() (pyhanko.sign.diff_analysis.rules.form_field_rules.SigFieldModification method), 149
 check_key_length() (pyhanko.pdf_utils.crypt.SecurityHandlerVersion method), 65
 chunk_size (pyhanko.sign.signers.cms_embedder.SigIOSetup attribute), 175
 chunk_stream() (in module pyhanko.pdf_utils.misc), 113
 chunked_digest() (in module pyhanko.pdf_utils.misc), 113
 chunked_write() (in module pyhanko.pdf_utils.misc), 113
 claimed_attrs (pyhanko.sign.ades.api.SignerAttrSpec attribute), 133
 claimed_attrs (pyhanko.sign.validation.status.CAdESSignerAttributeAssertions attribute), 240
 ClaimedAttributes (class in pyhanko.sign.validation.status), 239
 CLIConfig (class in pyhanko.config), 248
 cms_basic_validation() (in module pyhanko.sign.validation.generic_cms), 228
 CMSAlgorithmProtectionError, 225
 CMSAlgorithmProtectionProvider (class in pyhanko.sign.attributes), 138
 CMSAttributeProvider (class in pyhanko.sign.attributes), 137
 CMSExtractionError, 169
 CMSStructuralError, 170
 collect_dependencies() (pyhanko.pdf_utils.reader.HistoricalResolver method), 117
 collect_signer_attr_status() (in module pyhanko.sign.validation.generic_cms), 229
 collect_timing_info() (in module pyhanko.sign.validation.generic_cms), 226
 collect_validation_info() (in module py-

- hanko.sign.validation.dss*), 224
- COLOR_SPACE (*pyhanko.pdf_utils.content.ResourceType* attribute), 55
- combine_annotation (*pyhanko.sign.fields.SigFieldSpec* attribute), 154
- command_stream (*pyhanko.pdf_utils.barcodes.PdfStreamBarcodeWriter* property), 53
- commit() (*pyhanko.sign.signers.csc_signer.CSCSigner* method), 178
- commitment_type (*pyhanko.sign.ades.api.CAdESSignedAttrSpec* attribute), 133
- CommitmentTypeIdentifier (class in *pyhanko.sign.ades.cades_asn1*), 134
- CommitmentTypeIndication (class in *pyhanko.sign.ades.cades_asn1*), 134
- CommitmentTypeQualifier (class in *pyhanko.sign.ades.cades_asn1*), 134
- CommitmentTypeQualifiers (class in *pyhanko.sign.ades.cades_asn1*), 134
- compare_by_level (*pyhanko.pdf_utils.extensions.DeveloperExtension* attribute), 80
- compare_dicts() (in module *pyhanko.sign.diff_analysis.common*s), 142
- compare_fields() (*pyhanko.sign.diff_analysis.rules.form_field_rules.BaseFieldModificationRule* method), 150
- compare_key_refs() (in module *pyhanko.sign.diff_analysis.common*s), 142
- compress() (*pyhanko.pdf_utils.generic.StreamObject* method), 96
- compute_digest() (*pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature* method), 233
- compute_integrity_info() (*pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature* method), 233
- compute_signature_tst_digest() (in module *pyhanko.sign.validation.generic_cms*), 228
- compute_tst_digest() (*pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature* method), 233
- ConfigurableMixin (class in *pyhanko.pdf_utils.config_utils*), 54
- ConfigurationError, 54
- cons() (*pyhanko.pdf_utils.misc.ConsList* method), 114
- ConsList (class in *pyhanko.pdf_utils.misc*), 113
- container_ref (*pyhanko.pdf_utils.generic.DecryptedObject* property), 99
- container_ref (*pyhanko.pdf_utils.generic.PdfObject* attribute), 89
- content (*pyhanko.pdf_utils.text.TextBox* property), 121
- content_lines (*pyhanko.pdf_utils.text.TextBox* property), 121
- content_timestamp_validity (*pyhanko.sign.validation.status.StandardCMSSignatureStatus* attribute), 241
- CONTIGUOUS_BLOCK_FROM_START (*pyhanko.sign.validation.status.SignatureCoverageLevel* attribute), 242
- copy_into_new_writer() (in module *pyhanko.pdf_utils.writer*), 127
- country_code (*pyhanko.pdf_utils.misc.StringWithLanguage* attribute), 112
- coverage (*pyhanko.sign.validation.status.ModificationInfo* attribute), 242
- create_font_engine() (*pyhanko.pdf_utils.font.api.FontEngineFactory* method), 84
- create_font_engine() (*pyhanko.pdf_utils.font.basic.SimpleFontEngineFactory* method), 85
- create_font_engine() (*pyhanko.pdf_utils.font.opentype.GlyphAccumulatorFactory* method), 88
- create_objstream_if_needed (*pyhanko.pdf_utils.font.opentype.GlyphAccumulatorFactory* attribute), 87
- create_stamp() (*pyhanko.stamp.BaseStampStyle* method), 256
- create_stamp() (*pyhanko.stamp.QRStampStyle* method), 256
- create_stamp() (*pyhanko.stamp.StaticStampStyle* method), 257
- create_stamp() (*pyhanko.stamp.TextStampStyle* method), 255
- creation_date (*pyhanko.pdf_utils.metadata.model.DocumentMetadata* attribute), 107
- creator (*pyhanko.pdf_utils.metadata.model.DocumentMetadata* attribute), 107
- credential_id (*pyhanko.sign.signers.csc_signer.CSCServiceSessionInfo* attribute), 178
- credential_type (*pyhanko.pdf_utils.crypt.SerialisedCredential* attribute), 74
- crls (*pyhanko.sign.validation.dss.VRI* attribute), 220
- crls_to_embed (*pyhanko.sign.signers.pdf_signer.PreSignValidationStatus* attribute), 214
- CryptFilter (class in *pyhanko.pdf_utils.crypt*), 67
- CryptFilterConfiguration (class in *pyhanko.pdf_utils.crypt*), 66
- CRYPTO_CONSTRAINTS_FAILURE (*pyhanko.sign.ades.report.AdESIndeterminate* attribute), 136

CRYPTO_CONSTRAINTS_FAILURE_NO_POE (pyhanko.sign.ades.report.AdESIndeterminate attribute), 136	decrypt_pubkey() (pyhanko.pdf_utils.reader.PdfFileReader method), 116
CSCAuthorizationInfo (class in pyhanko.sign.signers.csc_signer), 180	decrypted (pyhanko.pdf_utils.generic.DecryptedObjectProxy property), 98
CSCAuthorizationManager (class in pyhanko.sign.signers.csc_signer), 180	DecryptedObjectProxy (class in pyhanko.pdf_utils.generic), 98
CSCCredentialInfo (class in pyhanko.sign.signers.csc_signer), 179	DEFAULT_CHUNK_SIZE (in module pyhanko.pdf_utils.misc), 113
CSCServiceSessionInfo (class in pyhanko.sign.signers.csc_signer), 178	DEFAULT_DIFF_POLICY (in module pyhanko.sign.diff_analysis.policies), 145
CSCSigner (class in pyhanko.sign.signers.csc_signer), 177	default_factory() (pyhanko.pdf_utils.font.basic.SimpleFontEngineFactory method), 85
curry_ref() (pyhanko.sign.diff_analysis.rules_api.ReferenceUpdatestatic class method), 153	DEFAULT_MD (in module pyhanko.sign.signers.constants), 182
D	default_md_for_signer (pyhanko.sign.signers.pdf_signer.PdfSigner property), 207
data (pyhanko.pdf_utils.crypt.SerialisedCredential attribute), 74	DEFAULT_SIG_SUBFILTER (in module pyhanko.sign.signers.constants), 182
data (pyhanko.pdf_utils.generic.StreamObject property), 95	DEFAULT_SIGNER_KEY_USAGE (in module pyhanko.sign.signers.constants), 182
DC_CREATOR (in module pyhanko.pdf_utils.metadata.model), 111	DEFAULT_SIGNING_STAMP_STYLE (in module pyhanko.sign.signers.constants), 182
DC_DESCRIPTION (in module pyhanko.pdf_utils.metadata.model), 111	default_stamp_style (pyhanko.config.CLIFConfig attribute), 248
DC_TITLE (in module pyhanko.pdf_utils.metadata.model), 111	default_validation_context (pyhanko.config.CLIFConfig attribute), 248
declared_startxref (pyhanko.pdf_utils.xref.XRefSectionMetaInfo attribute), 130	DEFER (pyhanko.config.PKCS11PinEntryMode attribute), 251
decode() (pyhanko.pdf_utils.filters.ASCII85Decode method), 81	DELIMITER_PATTERN (pyhanko.pdf_utils.generic.NameObject attribute), 93
decode() (pyhanko.pdf_utils.filters.ASCIIHexDecode method), 82	Dereferenceable (class in pyhanko.pdf_utils.generic), 88
decode() (pyhanko.pdf_utils.filters.Decoder method), 81	derive_object_key() (pyhanko.pdf_utils.crypt.AESCryptFilterMixin method), 72
decode() (pyhanko.pdf_utils.filters.FlateDecode method), 82	derive_object_key() (pyhanko.pdf_utils.crypt.CryptFilter method), 68
decode() (pyhanko.pdf_utils.generic.TextStringEncoding method), 97	derive_object_key() (pyhanko.pdf_utils.crypt.IdentityCryptFilter method), 70
Decoder (class in pyhanko.pdf_utils.filters), 81	derive_object_key() (pyhanko.pdf_utils.crypt.RC4CryptFilterMixin method), 72
decrypt() (pyhanko.pdf_utils.crypt.AESCryptFilterMixin method), 72	derive_shared_encryption_key() (pyhanko.pdf_utils.crypt.CryptFilter method), 68
decrypt() (pyhanko.pdf_utils.crypt.CryptFilter method), 68	derive_shared_encryption_key() (pyhanko.pdf_utils.crypt.IdentityCryptFilter method), 70
decrypt() (pyhanko.pdf_utils.crypt.EnvelopeKeyDecrypter method), 73	
decrypt() (pyhanko.pdf_utils.crypt.IdentityCryptFilter method), 71	
decrypt() (pyhanko.pdf_utils.crypt.RC4CryptFilterMixin method), 71	
decrypt() (pyhanko.pdf_utils.crypt.SimpleEnvelopeKeyDecrypter method), 74	
decrypt() (pyhanko.pdf_utils.reader.PdfFileReader method), 115	

`derive_shared_encryption_key()` (py-
`hanko.pdf_utils.crypt.PubKeyCryptFilter`
`method`), 70
`derive_shared_encryption_key()` (py-
`hanko.pdf_utils.crypt.StandardCryptFilter`
`method`), 69
`describe_timestamp_trust()` (py-
`hanko.sign.validation.status.TimestampSignatureStatus`
`method`), 239
`description` (`pyhanko.pdf_utils.embed.FileSpec` at-
`tribute`), 77
`descriptor` (`pyhanko.pdf_utils.font.basic.SimpleFontMeta` at-
`tribute`), 86
`deserialise()` (`pyhanko.pdf_utils.crypt.SerialisableCredentials`
`static method`), 75
`DeveloperExtension` (class in `py-`
`hanko.pdf_utils.extensions`), 79
`DevExtensionMultivalued` (class in `py-`
`hanko.pdf_utils.extensions`), 79
`DictionaryObject` (class in `pyhanko.pdf_utils.generic`),
94
`diff_result` (`pyhanko.sign.validation.status.ModificationInfo`
`attribute`), 242
`DiffPolicy` (class in `py-`
`hanko.sign.diff_analysis.policy_api`), 147
`DiffResult` (class in `py-`
`hanko.sign.diff_analysis.policy_api`), 147
`digest_doc_for_signing()` (py-
`hanko.sign.signers.pdf_signer.PdfSigner`
`method`), 207
`DIGEST_METHOD` (`pyhanko.sign.fields.SigSeedValFlags`
`attribute`), 155
`digest_methods` (`pyhanko.sign.fields.SigSeedValueSpec`
`attribute`), 158
`digest_tbs_document()` (py-
`hanko.sign.signers.pdf_signer.PdfTBSDocument`
`method`), 211
`DisplayText` (class in `pyhanko.sign.ades.cades_asn1`),
135
`DO_NOT_LOCK` (`pyhanko.sign.fields.SeedLockDocument`
`attribute`), 160
`doc_mdp_update_value` (py-
`hanko.sign.fields.SigFieldSpec` `attribute`),
154
`DocInfoRule` (class in `py-`
`hanko.sign.diff_analysis.rules.metadata_rules`),
151
`docmdp_level` (`pyhanko.sign.validation.pdf_embedded.Embedded`
`property`), 233
`docmdp_ok` (`pyhanko.sign.validation.status.PdfSignatureStatus`
`attribute`), 243
`docmdp_permissions` (py-
`hanko.sign.signers.pdf_signer.PdfSignatureMetadata`
`attribute`), 200
`docmdp_perms` (`pyhanko.sign.signers.cms_embedder.SigMDPSetup`
`attribute`), 174
`DocMDPInfo` (class in `py-`
`hanko.sign.validation.pdf_embedded`), 234
`document_digest` (py-
`hanko.sign.signers.pdf_byterange.PreparedByteRangeDigest`
`attribute`), 185
`document_id` (`pyhanko.pdf_utils.reader.HistoricalResolver`
`property`), 117
`document_id` (`pyhanko.pdf_utils.reader.PdfFileReader`
`property`), 114
`document_id` (`pyhanko.pdf_utils.rw_common.PdfHandler`
`property`), 119
`document_id` (`pyhanko.pdf_utils.writer.BasePdfFileWriter`
`property`), 122
`document_meta` (`pyhanko.pdf_utils.writer.BasePdfFileWriter`
`property`), 121
`document_meta_view` (py-
`hanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter`
`property`), 101
`document_meta_view` (py-
`hanko.pdf_utils.reader.HistoricalResolver`
`property`), 117
`document_meta_view` (py-
`hanko.pdf_utils.reader.PdfFileReader` `prop-`
`erty`), 114
`document_meta_view` (py-
`hanko.pdf_utils.rw_common.PdfHandler`
`property`), 119
`document_meta_view` (py-
`hanko.pdf_utils.writer.BasePdfFileWriter`
`property`), 121
`DocumentMetadata` (class in `py-`
`hanko.pdf_utils.metadata.model`), 106
`DocumentSecurityStore` (class in `py-`
`hanko.sign.validation.dss`), 220
`DocumentTimestamp` (class in `py-`
`hanko.sign.signers.pdf_byterange`), 187
`DocumentTimestampStatus` (class in `py-`
`hanko.sign.validation.status`), 244
`dss_settings` (`pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata`
`attribute`), 201
`dss_settings` (`pyhanko.sign.signers.pdf_signer.PostSignInstructions`
`attribute`), 215
`DSSCompareRule` (class in `py-`
`hanko.sign.diff_analysis.rules.form_field_rules`),
149
`DSSCompareSettings` (class in `py-`
`hanko.sign.signers.pdf_signer`), 201
`dummy_digest()` (in module `py-`
`hanko.sign.timestamps.common_utils`), 220
`DummyTimeStamper` (class in `py-`
`hanko.sign.timestamps.dummy_client`), 219

E

- `effective()` (`pyhanko.pdf_utils.layout.Margins` static method), 104
- `effective_height()` (`pyhanko.pdf_utils.layout.Margins` method), 104
- `effective_width()` (`pyhanko.pdf_utils.layout.Margins` method), 104
- `embed_checksum` (`pyhanko.pdf_utils.embed.EmbeddedFileParams` attribute), 77
- `embed_file()` (in module `pyhanko.pdf_utils.embed`), 76
- `embed_payload_with_cms()` (in module `pyhanko.sign.signers.functions`), 184
- `embed_roots` (`pyhanko.sign.signers.pdf_signer.PostSignInstructions` attribute), 215
- `embed_size` (`pyhanko.pdf_utils.embed.EmbeddedFileParams` attribute), 77
- `embed_validation_info` (`pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata` attribute), 200
- `embedded_attr_certs` (`pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature` property), 232
- `embedded_data` (`pyhanko.pdf_utils.embed.FileSpec` attribute), 77
- `embedded_data` (`pyhanko.pdf_utils.embed.RelatedFileSpec` attribute), 78
- `embedded_file_filter_name` (`pyhanko.pdf_utils.crypt.CryptFilterConfiguration` property), 67
- `embedded_regular_signatures` (`pyhanko.pdf_utils.reader.PdfFileReader` property), 116
- `embedded_signatures` (`pyhanko.pdf_utils.reader.PdfFileReader` property), 116
- `embedded_timestamp_signatures` (`pyhanko.pdf_utils.reader.PdfFileReader` property), 116
- `EmbeddedFileObject` (class in `pyhanko.pdf_utils.embed`), 76
- `EmbeddedFileParams` (class in `pyhanko.pdf_utils.embed`), 76
- `EmbeddedPdfSignature` (class in `pyhanko.sign.validation.pdf_embedded`), 232
- `empty()` (`pyhanko.pdf_utils.misc.ConsList` static method), 113
- `empty_field_appearance` (`pyhanko.sign.fields.SigFieldSpec` attribute), 154
- `encode()` (`pyhanko.pdf_utils.filters.ASCII85Decode` method), 81
- `encode()` (`pyhanko.pdf_utils.filters.ASCIIHexDecode` method), 82
- `encode()` (`pyhanko.pdf_utils.filters.Decoder` method), 81
- `encode()` (`pyhanko.pdf_utils.filters.FlateDecode` method), 83
- `encode()` (`pyhanko.pdf_utils.generic.TextStringEncoding` method), 97
- `encode_to_sv_string()` (`pyhanko.sign.fields.SigCertKeyUsage` method), 161
- `encoded_data` (`pyhanko.pdf_utils.generic.StreamObject` property), 95
- `encrypt()` (`pyhanko.pdf_utils.crypt.AESCryptFilterMixin` method), 72
- `encrypt()` (`pyhanko.pdf_utils.crypt.CryptFilter` method), 67
- `encrypt()` (`pyhanko.pdf_utils.crypt.IdentityCryptFilter` method), 71
- `encrypt()` (`pyhanko.pdf_utils.crypt.RC4CryptFilterMixin` method), 71
- `encrypt()` (`pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter` method), 101
- `encrypt()` (`pyhanko.pdf_utils.writer.PdfFileWriter` method), 125
- `encrypt_pubkey()` (`pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter` method), 101
- `encrypt_pubkey()` (`pyhanko.pdf_utils.writer.PdfFileWriter` method), 126
- `encrypted` (`pyhanko.pdf_utils.reader.PdfFileReader` property), 116
- `EncryptedObjAccess` (class in `pyhanko.pdf_utils.generic`), 97
- `end_location` (`pyhanko.pdf_utils.xref.XRefSectionMetaInfo` attribute), 130
- `endpoint_url()` (`pyhanko.sign.signers.csc_signer.CSCServiceSessionInfo` method), 179
- `ensure_objects_loaded()` (`pyhanko.sign.pkcs11.PKCS11Signer` method), 171
- `ensure_output_version()` (`pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter` method), 100
- `ensure_output_version()` (`pyhanko.pdf_utils.writer.BasePdfFileWriter` method), 122
- `ensure_sig_flags()` (in module `pyhanko.sign.fields`), 164
- `ENTIRE_FILE` (`pyhanko.sign.validation.status.SignatureCoverageLevel` attribute), 242
- `ENTIRE_REVISION` (`pyhanko.sign.validation.status.SignatureCoverageLevel` attribute), 242
- `entries` (`pyhanko.pdf_utils.metadata.model.XmpArray` attribute), 110

[enumerate_sig_fields\(\)](#) (in module `pyhanko.sign.fields`), 164
[EnvelopeKeyDecrypter](#) (class in `pyhanko.pdf_utils.crypt`), 73
[err_limit](#) (`pyhanko.pdf_utils.xref.XRefBuilder` attribute), 128
[establish_timestamp_trust\(\)](#) (in module `pyhanko.sign.validation.ltv`), 231
[estimate_signature_container_size\(\)](#) (`pyhanko.sign.signers.pdf_signer.PdfSigningSession` method), 210
[ETSI_RFC3161](#) (`pyhanko.sign.fields.SigSeedSubFilter` attribute), 160
[evaluate_modifications\(\)](#) (`pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature` method), 234
[evaluate_signature_coverage\(\)](#) (`pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature` method), 234
[EXCLUDE](#) (`pyhanko.sign.fields.FieldMDPAction` attribute), 162
[ExpandedName](#) (class in `pyhanko.pdf_utils.metadata.model`), 108
[expected_paths\(\)](#) (`pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonSpec` method), 144
[EXPIRED](#) (`pyhanko.sign.adess.report.AdESSIndeterminate` attribute), 136
[expires_at](#) (`pyhanko.sign.signers.csc_signer.CSCAuthorizationInfo` attribute), 180
[explicit_extd_key_usage_required](#) (`pyhanko.sign.validation.settings.KeyUsageConstraints` attribute), 237
[explicit_refs_in_revision\(\)](#) (`pyhanko.pdf_utils.reader.HistoricalResolver` method), 117
[explicit_refs_in_revision\(\)](#) (`pyhanko.pdf_utils.xref.XRefCache` method), 128
[EXT_G_STATE](#) (`pyhanko.pdf_utils.content.ResourceType` attribute), 55
[extd_key_usage](#) (`pyhanko.sign.validation.settings.KeyUsageConstraints` attribute), 236
[extd_key_usage](#) (`pyhanko.sign.validation.status.SignatureStatus` attribute), 238
[extd_key_usage](#) (`pyhanko.sign.validation.status.TimestampSignatureStatus` attribute), 239
[extension_level](#) (`pyhanko.pdf_utils.extensions.DeveloperExtension` attribute), 80
[extension_revision](#) (`pyhanko.pdf_utils.extensions.DeveloperExtension` attribute), 80
[ExternalSigner](#) (class in `pyhanko.sign.signers.pdf_cms`), 197
[extract_certificate_info\(\)](#) (in module `pyhanko.sign.general`), 168
[extract_certs_for_validation\(\)](#) (in module `pyhanko.sign.validation.generic_cms`), 229
[extract_contents\(\)](#) (in module `pyhanko.sign.validation.pdf_embedded`), 236
[extract_credential\(\)](#) (`pyhanko.pdf_utils.crypt.SecurityHandler` method), 59
[extract_message_digest\(\)](#) (in module `pyhanko.sign.validation.utils`), 244
[extract_self_reported_ts\(\)](#) (in module `pyhanko.sign.validation.generic_cms`), 229
[extract_signer_info\(\)](#) (in module `pyhanko.sign.general`), 167
[extract_ts_certs\(\)](#) (in module `pyhanko.sign.timestamps.common_utils`), 220
[extract_tst_data\(\)](#) (in module `pyhanko.sign.validation.generic_cms`), 228

F

[f_related_files](#) (`pyhanko.pdf_utils.embed.FileSpec` attribute), 77
[FAILED](#) (`pyhanko.pdf_utils.crypt.AuthStatus` attribute), 65
[FAILED](#) (`pyhanko.sign.adess.report.AdESSStatus` attribute), 136
[fetch_certs_in_csc_credential\(\)](#) (in module `pyhanko.sign.signers.csc_signer`), 180
[field_lock](#) (`pyhanko.sign.signers.cms_embedder.SigMDPSetup` attribute), 174
[field_mdp_spec](#) (`pyhanko.sign.fields.SigFieldSpec` attribute), 154
[field_name](#) (`pyhanko.sign.diff_analysis.form_rules_api.FormUpdate` attribute), 143
[field_name](#) (`pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata` attribute), 199
[field_name](#) (`pyhanko.sign.signers.pdf_signer.PdfTimeStamper` property), 203
[field_name](#) (`pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature` property), 232
[field_specs](#) (`pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonSpec` attribute), 144
[field_type](#) (`pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonSpec` attribute), 144
[FieldComparisonContext](#) (class in `pyhanko.sign.diff_analysis.form_rules_api`), 144
[FieldComparisonSpec](#) (class in `pyhanko.sign.diff_analysis.form_rules_api`), 143
[fieldmdp](#) (`pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature` property), 233

FieldMDPAction (class in *pyhanko.sign.fields*), 162
 FieldMDPRule (class in *pyhanko.sign.diff_analysis.form_rules_api*), 143
 FieldMDPSpec (class in *pyhanko.sign.fields*), 162
 fields (*pyhanko.sign.fields.FieldMDPSpec* attribute), 162
 file_credential (*pyhanko.sign.signers.pdf_signer.PostSignInstruction* attribute), 216
 file_name (*pyhanko.pdf_utils.embed.FileSpec* attribute), 77
 file_spec_string (*pyhanko.pdf_utils.embed.FileSpec* attribute), 77
 FileSpec (class in *pyhanko.pdf_utils.embed*), 77
 fill() (*pyhanko.sign.signers.pdf_byterange.PdfByteRangeDigest* method), 186
 FILL_FORMS (*pyhanko.sign.fields.MDPPerm* attribute), 162
 fill_reserved_region() (*pyhanko.sign.signers.pdf_byterange.PreparedByteRangeDigest* method), 185
 fill_with_cms() (*pyhanko.sign.signers.pdf_byterange.PreparedByteRangeDigest* method), 185
 FILTER (*pyhanko.sign.fields.SigSeedValFlags* attribute), 155
 filters() (*pyhanko.pdf_utils.crypt.CryptFilterConfiguration* method), 66
 finalise_output() (in module *pyhanko.pdf_utils.misc*), 113
 find_cms_attribute() (in module *pyhanko.sign.general*), 165
 find_page_container() (*pyhanko.pdf_utils.rw_common.PdfHandler* method), 119
 find_page_for_modification() (*pyhanko.pdf_utils.rw_common.PdfHandler* method), 119
 find_token() (in module *pyhanko.sign.pkcs11*), 172
 find_unique_cms_attribute() (in module *pyhanko.sign.general*), 165
 finish_signing() (*pyhanko.sign.signers.pdf_signer.PdfTBSDocument* class method), 212
 first_char (*pyhanko.pdf_utils.font.basic.SimpleFontMeta* attribute), 85
 fit() (*pyhanko.pdf_utils.layout.SimpleBoxLayoutRule* method), 105
 flags (*pyhanko.sign.fields.SigCertConstraints* attribute), 156
 flags (*pyhanko.sign.fields.SigSeedValueSpec* attribute), 157
 FlateDecode (class in *pyhanko.pdf_utils.filters*), 82
 flatten() (*pyhanko.pdf_utils.xref.TrailerDictionary* method), 131
 flipped (*pyhanko.pdf_utils.layout.AxisAlignment* property), 103
 FloatObject (class in *pyhanko.pdf_utils.generic*), 91
 fmt_related_files() (*pyhanko.pdf_utils.embed.RelatedFileSpec* class method), 78
 FONT (*pyhanko.pdf_utils.content.ResourceType* attribute), 55
 font (*pyhanko.pdf_utils.text.TextStyle* attribute), 120
 font_file (*pyhanko.pdf_utils.font.opentype.GlyphAccumulatorFactory* attribute), 87
 font_size (*pyhanko.pdf_utils.font.opentype.GlyphAccumulatorFactory* attribute), 87
 font_size (*pyhanko.pdf_utils.text.TextStyle* attribute), 120
 FontEngine (class in *pyhanko.pdf_utils.font.api*), 83
 FontEngineFactory (class in *pyhanko.pdf_utils.font.api*), 84
 FontSubsetCollection (class in *pyhanko.pdf_utils.font.api*), 84
 forbidden_set() (*pyhanko.sign.fields.SigCertKeyUsage* method), 161
 force_output_encoding (*pyhanko.pdf_utils.generic.TextStringObject* attribute), 92
 FORM_FILLING (*pyhanko.sign.diff_analysis.policy_api.ModificationLevel* attribute), 146
 format_attributes() (in module *pyhanko.sign.signers.pdf_cms*), 197
 format_csc_auth_request() (*pyhanko.sign.signers.csc_signer.CSCAuthorizationManager* method), 181
 format_csc_signing_req() (*pyhanko.sign.signers.csc_signer.CSCSigner* method), 178
 FORMAT_FAILURE (*pyhanko.sign.adès.report.AdESFailure* attribute), 136
 format_lock_dictionary() (*pyhanko.sign.fields.SigFieldSpec* method), 155
 format_revinfo() (*pyhanko.sign.signers.pdf_cms.Signer* static method), 188
 format_signed_attributes() (in module *pyhanko.sign.signers.pdf_cms*), 198
 FormUpdate (class in *pyhanko.sign.diff_analysis.form_rules_api*), 143
 FormUpdatingRule (class in *pyhanko.sign.diff_analysis.form_rules_api*), 142

FREE (pyhanko.pdf_utils.xref.XRefType attribute), 129
 from_certs() (pyhanko.sign.general.SimpleCertificateStore class method), 166
 from_config() (pyhanko.pdf_utils.config_utils.ConfigurableMixin class method), 54
 from_config() (pyhanko.pdf_utils.layout.InnerScaling class method), 105
 from_config() (pyhanko.pdf_utils.layout.Margins class method), 104
 from_config() (pyhanko.stamp.QRPosition class method), 257
 from_file_data() (pyhanko.pdf_utils.embed.EmbeddedFileObject class method), 76
 from_iterable() (pyhanko.sign.validation.status.ClaimedAttributes class method), 239
 from_number() (pyhanko.pdf_utils.crypt.SecurityHandlerVersion class method), 65
 from_number() (pyhanko.pdf_utils.crypt.StandardSecurityHandlerVersion class method), 66
 from_pdf_file() (pyhanko.stamp.StaticStampStyle class method), 257
 from_pdf_object() (pyhanko.sign.fields.FieldMDPSpec class method), 163
 from_pdf_object() (pyhanko.sign.fields.SigCertConstraints class method), 156
 from_pdf_object() (pyhanko.sign.fields.SigSeedValueSpec class method), 159
 from_reader() (pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter class method), 100
 from_results() (pyhanko.sign.validation.status.CertifiedAttributes class method), 240
 from_sets() (pyhanko.sign.fields.SigCertKeyUsage class method), 161
 from_x_align() (pyhanko.pdf_utils.layout.AxisAlignment class method), 103
 from_y_align() (pyhanko.pdf_utils.layout.AxisAlignment class method), 103

G

gather_encryption_metadata() (pyhanko.pdf_utils.crypt.StandardSecurityHandler class method), 62
 gather_pub_key_metadata() (pyhanko.pdf_utils.crypt.PubKeySecurityHandler class method), 64
 GeneralDSSContentSettings (class in pyhanko.sign.signers.pdf_signer), 202
 generation (pyhanko.pdf_utils.generic.IndirectObject property), 90
 generation (pyhanko.pdf_utils.generic.Reference attribute), 88
 generation (pyhanko.pdf_utils.xref.XRefEntry attribute), 129
 GENERIC (pyhanko.sign.adess.report.AdESIndeterminate attribute), 137
 GenericCommitment (class in pyhanko.sign.adess.api), 132
 GenericFieldModificationRule (class in pyhanko.sign.diff_analysis.rules.form_field_rules), 150
 get_and_apply() (in module pyhanko.pdf_utils.misc), 112
 get_and_apply() (pyhanko.pdf_utils.generic.DictionaryObject method), 94
 get_attribute() (pyhanko.sign.attributes.CMSAttributeProvider method), 137
 get_certificate_ref() (pyhanko.pdf_utils.generic.PdfObject method), 90
 get_courier() (in module pyhanko.pdf_utils.font.basic), 86
 get_default_text_params() (pyhanko.stamp.QRStamp method), 258
 get_default_text_params() (pyhanko.stamp.TextStamp method), 258
 get_embedded_file_filter() (pyhanko.pdf_utils.crypt.SecurityHandler method), 60
 get_file_encryption_key() (pyhanko.pdf_utils.crypt.PubKeySecurityHandler method), 64
 get_file_encryption_key() (pyhanko.pdf_utils.crypt.SecurityHandler method), 60
 get_file_encryption_key() (pyhanko.pdf_utils.crypt.StandardSecurityHandler method), 62
 get_for_embedded_file() (pyhanko.pdf_utils.crypt.CryptFilterConfiguration method), 66
 get_for_stream() (pyhanko.pdf_utils.crypt.CryptFilterConfiguration method), 66
 get_for_string() (pyhanko.pdf_utils.crypt.CryptFilterConfiguration method), 66
 get_generic_decoder() (in module pyhanko.pdf_utils.filters), 83
 get_historical_ref() (pyhanko.pdf_utils.xref.XRefCache method), 128

`get_historical_resolver()` (py- *method*), 90
hanko.pdf_utils.reader.PdfFileReader method), 116
`get_historical_root()` (py- *method*), 114
hanko.pdf_utils.reader.PdfFileReader method), 114
`get_introducing_revision()` (py- *method*), 127
hanko.pdf_utils.xref.XRefCache method), 127
`get_last_change()` (py- *method*), 127
hanko.pdf_utils.xref.XRefCache method), 127
`get_min_pdf_version()` (py- *method*), 61
hanko.pdf_utils.crypt.SecurityHandler method), 61
`get_name()` (pyhanko.pdf_utils.crypt.PubKeySecurityHandler *class method*), 63
class method), 59
`get_name()` (pyhanko.pdf_utils.crypt.SecurityHandler *class method*), 59
class method), 74
`get_name()` (pyhanko.pdf_utils.crypt.SerialisableCredential *class method*), 74
class method), 73
`get_name()` (pyhanko.pdf_utils.crypt.SimpleEnvelopeKeyDecrypter *class method*), 73
class method), 61
`get_name()` (pyhanko.pdf_utils.crypt.StandardSecurityHandler *class method*), 61
class method), 220
`get_nonce()` (in module py- *method*), 220
hanko.sign.timestamps.common_utils), 220
`get_object()` (pyhanko.pdf_utils.generic.DecryptedObjectProxy *method*), 98
method), 98
`get_object()` (pyhanko.pdf_utils.generic.Dereferenceable *method*), 88
method), 90
`get_object()` (pyhanko.pdf_utils.generic.IndirectObject *method*), 90
method), 90
`get_object()` (pyhanko.pdf_utils.generic.PdfObject *method*), 90
method), 89
`get_object()` (pyhanko.pdf_utils.generic.Reference *method*), 89
method), 89
`get_object()` (pyhanko.pdf_utils.generic.TrailerReference *method*), 89
method), 100
`get_object()` (pyhanko.pdf_utils.incremental_writer.IncrementalWriter *method*), 100
method), 117
`get_object()` (pyhanko.pdf_utils.reader.HistoricalResolver *method*), 117
method), 115
`get_object()` (pyhanko.pdf_utils.reader.PdfFileReader *method*), 115
method), 118
`get_object()` (pyhanko.pdf_utils.rw_common.PdfHandler *method*), 118
method), 122
`get_object()` (pyhanko.pdf_utils.writer.BasePdfFileWriter *method*), 122
method), 88
`get_pdf_handler()` (py- *method*), 88
hanko.pdf_utils.generic.Dereferenceable method), 88
`get_pdf_handler()` (py- *method*), 127
hanko.pdf_utils.generic.IndirectObject method), 127
`get_pdf_handler()` (py- *method*), 89
hanko.pdf_utils.generic.Reference method), 89
`get_pdf_handler()` (py- *method*), 89
hanko.pdf_utils.generic.TrailerReference method), 89
`get_pemder_config()` (pyhanko.config.CLIFConfig *method*), 249
method), 249
`get_pkcs11_config()` (pyhanko.config.CLIFConfig *method*), 249
method), 249
`get_pkcs12_config()` (pyhanko.config.CLIFConfig *method*), 249
method), 168
`get_pyca_cryptography_hash()` (in module py- *method*), 168
hanko.sign.general), 168
`get_session()` (pyhanko.sign.timestamps.aiohttp_client.AIOHttpTimeStamps *method*), 218
method), 201
`get_settings_for_ts()` (py- *method*), 201
hanko.sign.signers.pdf_signer.DSSContentSettings method), 201
`get_signature_mechanism()` (py- *method*), 177
hanko.sign.signers.csc_signer.CSCSigner method), 177
`get_signature_mechanism()` (py- *method*), 188
hanko.sign.signers.pdf_cms.Signer method), 188
`get_signer_key_usages()` (py- *method*), 249
hanko.config.CLIFConfig method), 249
`get_stamp_style()` (pyhanko.config.CLIFConfig *method*), 249
method), 249
`get_startxref_for_revision()` (py- *method*), 128
hanko.pdf_utils.xref.XRefCache method), 128
`get_stream_filter()` (py- *method*), 60
hanko.pdf_utils.crypt.SecurityHandler method), 60
`get_string_filter()` (py- *method*), 60
hanko.pdf_utils.crypt.SecurityHandler method), 60
`get_using_file_writer()` (py- *method*), 121
hanko.pdf_utils.writer.BasePdfFileWriter method), 121
`get_timestamp_chain()` (in module py- *method*), 230
hanko.sign.validation.ltv), 230
`get_validation_context()` (py- *method*), 249
hanko.config.CLIFConfig method), 249
`get_value_as_reference()` (py- *method*), 94
hanko.pdf_utils.generic.DictionaryObject method), 94
`get_xref_container_info()` (py- *method*), 127
hanko.pdf_utils.xref.XRefCache method), 127
`get_xref_data()` (pyhanko.pdf_utils.xref.XRefCache *method*), 127
method), 127

- GlyphAccumulator (class in *pyhanko.pdf_utils.font.opentype*), 86
- GlyphAccumulatorFactory (class in *pyhanko.pdf_utils.font.opentype*), 87
- graphics_ops (*pyhanko.pdf_utils.font.api.ShapeResult* attribute), 83
- ## H
- handle_tsp_response() (in module *pyhanko.sign.timestamps.common_utils*), 220
- has_non_lang_qual (pyhanko.pdf_utils.metadata.model.Qualifiers property), 109
- has_seed_values (pyhanko.sign.validation.status.PdfSignatureStatus attribute), 243
- has_xref_stream (pyhanko.pdf_utils.reader.PdfFileReader attribute), 114
- HASH_FAILURE (*pyhanko.sign.ad.es.report.AdESFailure* attribute), 136
- hash_pinning_required (pyhanko.sign.signers.csc_signer.CSCCredentialInfo attribute), 179
- head (*pyhanko.pdf_utils.misc.ConsList* attribute), 113
- height (*pyhanko.pdf_utils.layout.BoxConstraints* property), 102
- height_defined (*pyhanko.pdf_utils.layout.BoxConstraints* property), 102
- higher_generation_refs() (pyhanko.pdf_utils.xref.XRefSectionData method), 130
- HistoricalResolver (class in *pyhanko.pdf_utils.reader*), 116
- horizontal_flow (*pyhanko.stamp.QRPosition* property), 257
- HTTPTimeStamper (class in *pyhanko.sign.timestamps.requests_client*), 219
- HYBRID_MAIN (*pyhanko.pdf_utils.xref.XRefSectionType* attribute), 130
- HYBRID_STREAM (*pyhanko.pdf_utils.xref.XRefSectionType* attribute), 130
- hybrid_xrefs_present (pyhanko.pdf_utils.xref.XRefCache property), 128
- ## I
- IdentityCryptFilter (class in *pyhanko.pdf_utils.crypt*), 70
- idnum (*pyhanko.pdf_utils.generic.IndirectObject* property), 90
- idnum (*pyhanko.pdf_utils.generic.Reference* attribute), 88
- idnum (*pyhanko.pdf_utils.xref.XRefEntry* attribute), 129
- image_ref (*pyhanko.pdf_utils.images.PdfImage* property), 99
- import_object() (*pyhanko.pdf_utils.writer.BasePdfFileWriter* method), 124
- import_page_as_xobject() (*pyhanko.pdf_utils.writer.BasePdfFileWriter* method), 124
- import_resources() (*pyhanko.pdf_utils.content.PdfContent* method), 56
- ImportedPdfPage (class in *pyhanko.pdf_utils.content*), 57
- IN_OBJ_STREAM (*pyhanko.pdf_utils.xref.XRefType* attribute), 129
- in_place (*pyhanko.sign.signers.cms_embedder.SigIOSetup* attribute), 175
- INCLUDE (*pyhanko.sign.fields.FieldMDPAction* attribute), 162
- include_vri (*pyhanko.sign.signers.pdf_signer.GeneralDSSContentSetting* attribute), 202
- IncrementalPdfFileWriter (class in *pyhanko.pdf_utils.incremental_writer*), 100
- INDETERMINATE (*pyhanko.sign.ad.es.report.AdESStatus* attribute), 136
- IndirectObject (class in *pyhanko.pdf_utils.generic*), 90
- IndirectObjectExpected, 112
- info_url (*pyhanko.sign.fields.SigCertConstraints* attribute), 156
- init_signing_session() (*pyhanko.sign.signers.pdf_signer.PdfSigner* method), 207
- init_validation_context_kwargs() (in module *pyhanko.config*), 249
- init_xobject_dictionary() (in module *pyhanko.pdf_utils.writer*), 126
- inner_content_layout (*pyhanko.stamp.TextStampStyle* attribute), 255
- inner_content_scaling (*pyhanko.pdf_utils.layout.SimpleBoxLayoutRule* attribute), 105
- InnerScaling (class in *pyhanko.pdf_utils.layout*), 104
- innsep (*pyhanko.stamp.QRStampStyle* attribute), 256
- input_version (*pyhanko.pdf_utils.reader.PdfFileReader* property), 114
- insert_page() (*pyhanko.pdf_utils.writer.BasePdfFileWriter* method), 123
- instance_test() (in module *pyhanko.pdf_utils.misc*), 113
- instantiate() (*pyhanko.config.PemDerSignatureConfig* method), 250
- instantiate() (*pyhanko.config.PKCS12SignatureConfig* method), 250

instantiate_from_pdf_object() (pyhanko.pdf_utils.crypt.PubKeySecurityHandler class method), 64
 instantiate_from_pdf_object() (pyhanko.pdf_utils.crypt.SecurityHandler class method), 59
 instantiate_from_pdf_object() (pyhanko.pdf_utils.crypt.StandardSecurityHandler class method), 62
 intact (pyhanko.sign.validation.status.SignatureStatus attribute), 238
 invis_sig_settings (pyhanko.sign.fields.SigFieldSpec attribute), 154
 InvisSigSettings (class in pyhanko.sign.fields), 163
 IO_CHUNK_SIZE (pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter attribute), 100
 is_authenticated() (pyhanko.pdf_utils.crypt.SecurityHandler method), 59
 is_embedded_file_stream (pyhanko.pdf_utils.generic.StreamObject property), 96
 is_locked() (pyhanko.sign.fields.FieldMDPSpec method), 163
 is_ref_available() (pyhanko.pdf_utils.reader.HistoricalResolver method), 117
 is_regular_character() (in module pyhanko.pdf_utils.misc), 112
 is_well_formed_xml() (pyhanko.sign.diff_analysis.rules.metadata_rules.MetadataUpdateRule static method), 152
 isoparse() (in module pyhanko.pdf_utils.misc), 114
 ISSUER (pyhanko.sign.fields.SigCertConstraintFlags attribute), 159
 issuers (pyhanko.sign.fields.SigCertConstraints attribute), 156
 items() (pyhanko.pdf_utils.xref.TrailerDictionary method), 131
 iter_equals() (pyhanko.pdf_utils.metadata.model.Qualifier method), 108
 ix_in_stream (pyhanko.pdf_utils.xref.ObjStreamRef attribute), 129

K

key_file (pyhanko.config.PemDerSignatureConfig attribute), 250
 key_id (pyhanko.config.PKCS11SignatureConfig attribute), 252
 key_label (pyhanko.config.PKCS11SignatureConfig attribute), 252
 key_passphrase (pyhanko.config.PemDerSignatureConfig attribute), 250
 KEY_USAGE (pyhanko.sign.fields.SigCertConstraintFlags attribute), 159
 key_usage (pyhanko.sign.fields.SigCertConstraints attribute), 156
 key_usage (pyhanko.sign.validation.settings.KeyUsageConstraints attribute), 236
 key_usage (pyhanko.sign.validation.status.SignatureStatus attribute), 238
 key_usage (pyhanko.sign.validation.status.TimestampSignatureStatus attribute), 239
 key_usage_forbidden (pyhanko.sign.validation.settings.KeyUsageConstraints attribute), 236
 keylen (pyhanko.pdf_utils.crypt.AESCryptFilterMixin attribute), 72
 keylen (pyhanko.pdf_utils.crypt.CryptFilter property), 67
 keylen (pyhanko.pdf_utils.crypt.IdentityCryptFilter attribute), 70
 keylen (pyhanko.pdf_utils.crypt.RC4CryptFilterMixin attribute), 71
 keys() (pyhanko.pdf_utils.xref.TrailerDictionary method), 131
 KeyUsageConstraints (class in pyhanko.sign.validation.settings), 236
 keywords (pyhanko.pdf_utils.metadata.model.DocumentMetadata attribute), 107

L

label (pyhanko.config.TokenCriteria attribute), 250
 lang (pyhanko.pdf_utils.metadata.model.Qualifiers property), 109
 lang_as_qual() (pyhanko.pdf_utils.metadata.model.Qualifiers class method), 108
 lang_code (pyhanko.pdf_utils.misc.StringWithLanguage attribute), 112
 last_char (pyhanko.pdf_utils.font.basic.SimpleFontMeta attribute), 85
 last_modified (pyhanko.pdf_utils.metadata.model.DocumentMetadata attribute), 107
 last_startxref (pyhanko.pdf_utils.reader.PdfFileReader attribute), 114
 LayoutError, 102
 leading (pyhanko.pdf_utils.text.TextBox property), 121
 leading (pyhanko.pdf_utils.text.TextStyle attribute), 120
 left (pyhanko.pdf_utils.layout.Margins attribute), 103
 LEFT_OF_TEXT (pyhanko.stamp.QRPosition attribute), 257
 LEGAL_ATTESTATION (pyhanko.sign.fields.SigSeedValFlags attribute), 155
 legal_attestations (pyhanko.sign.fields.SigSeedValueSpec attribute), 158

level (pyhanko.config.LogConfig attribute), 248
 load() (pyhanko.pdf_utils.crypt.SimpleEnvelopeKeyDecrypter static method), 74
 load() (pyhanko.sign.signers.pdf_cms.SimpleSigner class method), 196
 load_cert_from_pemder() (in module pyhanko.sign.general), 168
 load_certs() (pyhanko.sign.validation.dss.DocumentSecurityStore attribute), 238
 load_certs() (pyhanko.sign.validation.dss.DocumentSecurityStore method), 221
 load_certs_from_pemder() (in module pyhanko.sign.general), 168
 load_pkcs12() (pyhanko.pdf_utils.crypt.SimpleEnvelopeKeyDecrypter class method), 74
 load_pkcs12() (pyhanko.sign.signers.pdf_cms.SimpleSigner class method), 196
 load_private_key_from_pemder() (in module pyhanko.sign.general), 168
 local_name (pyhanko.pdf_utils.metadata.model.ExpandedName attribute), 108
 location (pyhanko.pdf_utils.xref.XRefEntry attribute), 129
 location (pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute), 199
 LOCK (pyhanko.sign.fields.SeedLockDocument attribute), 160
 LOCK_DOCUMENT (pyhanko.sign.fields.SigSeedValFlags attribute), 155
 lock_document (pyhanko.sign.fields.SigSeedValueSpec attribute), 158
 log_config (pyhanko.config.CLIFConfig attribute), 248
 LogConfig (class in pyhanko.config), 248
 LTA_UPDATES (pyhanko.sign.diff_analysis.policy_api.ModificationLevel attribute), 146
M
 Margins (class in pyhanko.pdf_utils.layout), 103
 margins (pyhanko.pdf_utils.layout.SimpleBoxLayoutRule attribute), 105
 mark_update() (pyhanko.pdf_utils.incremental_writer.IncrementalPdfWriter method), 100
 mark_update() (pyhanko.pdf_utils.writer.BasePdfFileWriter method), 122
 marked_content_property_list() (pyhanko.pdf_utils.font.opentype.GlyphAccumulator method), 86
 match_all_key_usages (pyhanko.sign.validation.settings.KeyUsageConstraints attribute), 237
 match_issuer_serial() (in module pyhanko.sign.general), 169
 max_batch_size (pyhanko.sign.signers.csc_signer.CSCCredentialed attribute), 179
 MAYBE (pyhanko.pdf_utils.extensions.DevExtensionMultivalued attribute), 79
 md_algorithm (pyhanko.sign.signers.cms_embedder.SigIOSetup attribute), 175
 md_algorithm (pyhanko.sign.signers.cms_embedder.SigMDPSetup attribute), 174
 md_algorithm (pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute), 199
 md_algorithm (pyhanko.sign.validation.status.SignatureStatus attribute), 238
 mdp_setup (pyhanko.sign.signers.cms_embedder.SigObjSetup attribute), 175
 MDPPerm (class in pyhanko.sign.fields), 161
 MetadataResources() (pyhanko.pdf_utils.writer.BasePdfFileWriter method), 125
 meta_info (pyhanko.pdf_utils.xref.XRefSection attribute), 129
 MetadataUpdateRule (class in pyhanko.sign.diff_analysis.rules.metadata_rules), 151
 MetaString (in module pyhanko.pdf_utils.metadata.model), 108
 method (pyhanko.pdf_utils.crypt.AESCryptFilterMixin attribute), 72
 method (pyhanko.pdf_utils.crypt.CryptFilter property), 67
 method (pyhanko.pdf_utils.crypt.IdentityCryptFilter attribute), 70
 method (pyhanko.pdf_utils.crypt.RC4CryptFilterMixin attribute), 71
 modification_date (pyhanko.pdf_utils.embed.EmbeddedFileParams attribute), 77
 modification_level (pyhanko.sign.diff_analysis.policy_api.DiffResult attribute), 147
 modification_level (pyhanko.sign.validation.status.ModificationInfo property), 243
 ModificationInfo (class in pyhanko.sign.validation.status), 242
 ModificationLevel (class in pyhanko.sign.diff_analysis.policy_api), 146
 modified (pyhanko.sign.validation.dss.DocumentSecurityStore property), 220
 module
 pyhanko.config, 248
 pyhanko.pdf_utils.barcodes, 53
 pyhanko.pdf_utils.config_utils, 54
 pyhanko.pdf_utils.content, 55
 pyhanko.pdf_utils.crypt, 57
 pyhanko.pdf_utils.embed, 76
 pyhanko.pdf_utils.extensions, 79
 pyhanko.pdf_utils.filters, 81
 pyhanko.pdf_utils.font.api, 83

pyhanko.pdf_utils.font.basic, 85
 pyhanko.pdf_utils.font.opentype, 86
 pyhanko.pdf_utils.generic, 88
 pyhanko.pdf_utils.images, 99
 pyhanko.pdf_utils.incremental_writer, 100
 pyhanko.pdf_utils.layout, 102
 pyhanko.pdf_utils.metadata.model, 106
 pyhanko.pdf_utils.misc, 112
 pyhanko.pdf_utils.reader, 114
 pyhanko.pdf_utils.rw_common, 118
 pyhanko.pdf_utils.text, 120
 pyhanko.pdf_utils.writer, 121
 pyhanko.pdf_utils.xref, 127
 pyhanko.sign.ades.api, 132
 pyhanko.sign.ades.asn1_util, 134
 pyhanko.sign.ades.cades_asn1, 134
 pyhanko.sign.ades.report, 135
 pyhanko.sign.attributes, 137
 pyhanko.sign.beid, 139
 pyhanko.sign.diff_analysis.common, 141
 pyhanko.sign.diff_analysis.form_rules_api, 142
 pyhanko.sign.diff_analysis.policies, 144
 pyhanko.sign.diff_analysis.policy_api, 146
 pyhanko.sign.diff_analysis.rules.file_structure_rules, 148
 pyhanko.sign.diff_analysis.rules.form_field_rules, 149
 pyhanko.sign.diff_analysis.rules.metadata_rules, 151
 pyhanko.sign.diff_analysis.rules_api, 152
 pyhanko.sign.fields, 153
 pyhanko.sign.general, 165
 pyhanko.sign.pkcs11, 170
 pyhanko.sign.signers.cms_embedder, 173
 pyhanko.sign.signers.constants, 182
 pyhanko.sign.signers.csc_signer, 176
 pyhanko.sign.signers.functions, 183
 pyhanko.sign.signers.pdf_byterange, 184
 pyhanko.sign.signers.pdf_cms, 187
 pyhanko.sign.signers.pdf_signer, 198
 pyhanko.sign.timestamps.aiohttp_client, 218
 pyhanko.sign.timestamps.api, 216
 pyhanko.sign.timestamps.common_utils, 220
 pyhanko.sign.timestamps.dummy_client, 219
 pyhanko.sign.timestamps.requests_client, 219
 pyhanko.sign.validation, 244
 pyhanko.sign.validation.dss, 220
 pyhanko.sign.validation.errors, 224
 pyhanko.sign.validation.generic_cms, 225
 pyhanko.sign.validation.ltv, 230
 pyhanko.sign.validation.pdf_embedded, 232
 pyhanko.sign.validation.settings, 236
 pyhanko.sign.validation.status, 238
 pyhanko.sign.validation.utils, 244
 pyhanko.stamp, 253
 module_path (pyhanko.config.PKCS11SignatureConfig attribute), 252
 multivalued (pyhanko.pdf_utils.extensions.DeveloperExtension attribute), 80
 MultivaluedAttributeError, 166
 must_have_set() (pyhanko.sign.fields.SigCertKeyUsage method), 161

N

name (pyhanko.pdf_utils.embed.RelatedFileSpec attribute), 78
 name (pyhanko.sign.signers.cms_embedder.SigAppearanceSetup attribute), 175
 name (pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute), 199
 NameObject (class in pyhanko.pdf_utils.generic), 93
 NEVER (pyhanko.pdf_utils.extensions.DevExtensionMultivalued attribute), 79
 new (pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonContext attribute), 144
 new_field (pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonContext attribute), 144
 new_field_ref (pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonContext attribute), 144
 next_ts_settings (pyhanko.sign.signers.pdf_signer.DSSContentSettings attribute), 201
 NO_CERTIFICATE_CHAIN_FOUND (pyhanko.sign.ades.report.AdESIndeterminate attribute), 136
 NO_CHANGES (pyhanko.sign.fields.MDPPerm attribute), 162
 NO_CHANGES_DIFF_POLICY (in module pyhanko.sign.diff_analysis.policies), 146
 NO_POE (pyhanko.sign.ades.report.AdESIndeterminate attribute), 136
 NO_SCALING (pyhanko.pdf_utils.layout.InnerScaling attribute), 104
 NO_SIGNING_CERTIFICATE_FOUND (pyhanko.sign.ades.report.AdESIndeterminate attribute), 136
 NoDSSFoundError, 224
 non_trailer_keys (pyhanko.pdf_utils.xref.TrailerDictionary attribute), 130
 NONE (pyhanko.sign.diff_analysis.policy_api.ModificationLevel attribute), 146
 NonexistentAttributeError, 166

NOT_YET_VALID (*pyhanko.sign.ades.report.AdESIndeterminate* attribute), 136

NoticeNumbers (class in *pyhanko.sign.ades.cades_asn1*), 134

NoticeReference (class in *pyhanko.sign.ades.cades_asn1*), 134

NS (in module *pyhanko.pdf_utils.metadata.model*), 110

ns (*pyhanko.pdf_utils.metadata.model.ExpandedName* attribute), 108

NullObject (class in *pyhanko.pdf_utils.generic*), 91

NumberObject (class in *pyhanko.pdf_utils.generic*), 91

NumberPattern (*pyhanko.pdf_utils.generic.NumberObject* attribute), 92

O

oauth_token (*pyhanko.sign.signers.csc_signer.CSCServiceSessionInfo* attribute), 179

obj_stream_id (*pyhanko.pdf_utils.xref.ObjStreamRef* attribute), 129

object_streams (*pyhanko.pdf_utils.writer.PdfFileWriter* attribute), 126

object_streams_used() (*pyhanko.pdf_utils.reader.HistoricalResolver* method), 117

object_streams_used_in() (*pyhanko.pdf_utils.xref.XRefCache* method), 127

ObjectHeaderReadError, 129

ObjectStream (class in *pyhanko.pdf_utils.xref*), 131

ObjectStreamRule (class in *pyhanko.sign.diff_analysis.rules.file_structure_rules*), 148

ObjStreamRef (class in *pyhanko.pdf_utils.xref*), 129

ocsps (*pyhanko.sign.validation.dss.VRI* attribute), 220

ocsps_to_embed (*pyhanko.sign.signers.pdf_signer.PreSignValidationStatus* attribute), 214

of() (*pyhanko.pdf_utils.metadata.model.Qualifiers* class method), 108

of() (*pyhanko.pdf_utils.metadata.model.XmpStructure* class method), 109

OID (*pyhanko.sign.fields.SigCertConstraintFlags* attribute), 159

old (*pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonContext* attribute), 144

old_canonical_path (*pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonSpec* attribute), 144

old_field (*pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonSpec* property), 144

old_field_ref (*pyhanko.sign.diff_analysis.form_rules_api.FieldComparisonSpec* attribute), 144

on_page (*pyhanko.sign.fields.SigFieldSpec* attribute), 154

open_beid_session() (in module *pyhanko.sign.beid*), 139

open_pkcs11_session() (in module *pyhanko.sign.pkcs11*), 171

optimal_pss_params() (in module *pyhanko.sign.general*), 168

ORDERED (*pyhanko.pdf_utils.metadata.model.XmpArrayType* attribute), 109

ordered() (*pyhanko.pdf_utils.metadata.model.XmpArray* class method), 110

OrderedEnum (class in *pyhanko.pdf_utils.misc*), 112

original_bytes (*pyhanko.pdf_utils.generic.ByteStringObject* property), 92

original_bytes (*pyhanko.pdf_utils.generic.TextStringObject* property), 92

ot_language_tag (*pyhanko.pdf_utils.font.opentype.GlyphAccumulatorFactory* attribute), 87

ot_script_tag (*pyhanko.pdf_utils.font.opentype.GlyphAccumulatorFactory* attribute), 87

OTHER (*pyhanko.pdf_utils.crypt.SecurityHandlerVersion* attribute), 65

OTHER (*pyhanko.pdf_utils.crypt.StandardSecuritySettingsRevision* attribute), 65

OTHER (*pyhanko.sign.diff_analysis.policy_api.ModificationLevel* attribute), 146

other_certs (*pyhanko.config.PemDerSignatureConfig* attribute), 250

other_certs (*pyhanko.config.PKCS11SignatureConfig* attribute), 252

other_certs (*pyhanko.config.PKCS12SignatureConfig* attribute), 249

other_certs (*pyhanko.sign.general.SignedDataCerts* attribute), 167

other_certs_to_pull (*pyhanko.config.PKCS11SignatureConfig* attribute), 252

other_embedded_certs (*pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature* property), 232

OtherAttrCert (class in *pyhanko.sign.ades.cades_asn1*), 135

OtherAttrCertId (class in *pyhanko.sign.ades.cades_asn1*), 135

OUT_OF_BOUNDS_NO_POE (*pyhanko.sign.ades.report.AdESIndeterminate* attribute), 136

OUTPUT (*pyhanko.config.LogConfig* attribute), 248

output (*pyhanko.sign.signers.cms_embedder.SigIOSetup* attribute), 176

output_version (*pyhanko.pdf_utils.writer.BasePdfFileWriter* attribute), 121

OWNER (*pyhanko.pdf_utils.crypt.AuthStatus* attribute), 65

P

- PADES (pyhanko.sign.fields.SigSeedSubFilter attribute), 160
- PADES_LT (pyhanko.sign.validation.ltv.RevocationInfoValidationType attribute), 230
- PADES_LTA (pyhanko.sign.validation.ltv.RevocationInfoValidationType attribute), 230
- PageObject (class in pyhanko.pdf_utils.writer), 125
- parse_catalog_version() (in module pyhanko.pdf_utils.reader), 118
- parse_cli_config() (in module pyhanko.config), 253
- parse_csc_auth_response() (pyhanko.sign.signers.csc_signer.CSCAuthorizationManager static method), 181
- parse_logging_config() (in module pyhanko.config), 249
- parse_mode_setting() (pyhanko.config.PKCS11PinEntryMode static method), 251
- parse_output_spec() (pyhanko.config.LogConfig static method), 248
- parse_trust_config() (in module pyhanko.config), 249
- parse_xref_stream() (in module pyhanko.pdf_utils.xref), 131
- parse_xref_table() (in module pyhanko.pdf_utils.xref), 131
- PASSED (pyhanko.sign.adess.report.AdESSStatus attribute), 135
- paths_checked (pyhanko.sign.diff_analysis.rules_api.ReferenceUpdate attribute), 153
- PATTERN (pyhanko.pdf_utils.content.ResourceType attribute), 55
- pdf (pyhanko.pdf_utils.generic.Reference attribute), 88
- PDF_1_5 (pyhanko.sign.fields.SeedValueDictVersion attribute), 160
- PDF_1_7 (pyhanko.sign.fields.SeedValueDictVersion attribute), 160
- PDF_2_0 (pyhanko.sign.fields.SeedValueDictVersion attribute), 160
- pdf_date() (in module pyhanko.pdf_utils.generic), 96
- PDF_DOC (pyhanko.pdf_utils.generic.TextStringEncoding attribute), 97
- PDF_KEYWORDS (in module pyhanko.pdf_utils.metadata.model), 111
- pdf_name (in module pyhanko.pdf_utils.generic), 96
- PDF_PRODUCER (in module pyhanko.pdf_utils.metadata.model), 111
- pdf_string() (in module pyhanko.pdf_utils.generic), 96
- PdfByteRangeDigest (class in pyhanko.sign.signers.pdf_byterange), 185
- PdfCMSEmbedder (class in pyhanko.sign.signers.cms_embedder), 173
- PdfCMSSignedAttributes (class in pyhanko.sign.signers.pdf_cms), 197
- PdfContent (class in pyhanko.pdf_utils.content), 56
- PdfError, 112
- PdfFileReader (class in pyhanko.pdf_utils.reader), 114
- PdfFileWriter (class in pyhanko.pdf_utils.writer), 125
- PdfHandler (class in pyhanko.pdf_utils.rw_common), 118
- PdfImage (class in pyhanko.pdf_utils.images), 99
- PdfKeyNotAvailableError, 75
- PdfObject (class in pyhanko.pdf_utils.generic), 89
- PdfPostSignatureDocument (class in pyhanko.sign.signers.pdf_signer), 213
- PdfReadError, 112
- PdfResources (class in pyhanko.pdf_utils.content), 55
- PdfSignatureMetadata (class in pyhanko.sign.signers.pdf_signer), 198
- PdfSignatureStatus (class in pyhanko.sign.validation.status), 243
- PdfSignedData (class in pyhanko.sign.signers.pdf_byterange), 186
- PdfSigner (class in pyhanko.sign.signers.pdf_signer), 206
- PdfSigningSession (class in pyhanko.sign.signers.pdf_signer), 210
- PdfStreamBarcodeWriter (class in pyhanko.pdf_utils.barcodes), 53
- PdfStreamError, 112
- PdfStrictReadError, 112
- PdfTBSDocument (class in pyhanko.sign.signers.pdf_signer), 211
- PdfTimeStamper (class in pyhanko.sign.signers.pdf_signer), 203
- PdfWriteError, 112
- peek() (in module pyhanko.pdf_utils.misc), 113
- pemder_setups (pyhanko.config.CLIConfig attribute), 248
- PemDerSignatureConfig (class in pyhanko.config), 250
- perform_presign_validation() (pyhanko.sign.signers.pdf_signer.PdfSigningSession method), 210
- perform_signature() (pyhanko.sign.signers.pdf_signer.PdfTBSDocument method), 211
- permission (pyhanko.sign.validation.pdf_embedded.DocMDPInfo property), 234
- permission_flags (pyhanko.pdf_utils.crypt.AuthResult attribute), 65
- pfx_file (pyhanko.config.PKCS12SignatureConfig attribute), 249
- pfx_passphrase (pyhanko.config.PKCS12SignatureConfig attribute), 249
- pil_image() (in module pyhanko.pdf_utils.images), 99
- pkcs11_setups (pyhanko.config.CLIConfig attribute),

- 249
- PKCS11PinEntryMode (class in *pyhanko.config*), 251
- PKCS11SignatureConfig (class in *pyhanko.config*), 251
- PKCS11Signer (class in *pyhanko.sign.pkcs11*), 170
- PKCS11SigningContext (class in *pyhanko.sign.pkcs11*), 172
- pkcs12_setups (*pyhanko.config.CLIConfig* attribute), 248
- PKCS12SignatureConfig (class in *pyhanko.config*), 249
- pkcs7_signature_mechanism (*pyhanko.sign.validation.status.SignatureStatus* attribute), 238
- placement (*pyhanko.sign.signers.pdf_signer.DSSContentSettings* attribute), 201
- POLICY_PROCESSING_ERROR (*pyhanko.sign.ades.report.AdESIndeterminate* attribute), 136
- Positioning (class in *pyhanko.pdf_utils.layout*), 106
- post_signature_processing() (*pyhanko.sign.signers.pdf_signer.PdfPostSignatureDocument* class method), 213
- PostSignInstructions (class in *pyhanko.sign.signers.pdf_signer*), 214
- prefer_pss (*pyhanko.config.PemDerSignatureConfig* attribute), 250
- prefer_pss (*pyhanko.config.PKCS11SignatureConfig* attribute), 253
- prefer_pss (*pyhanko.config.PKCS12SignatureConfig* attribute), 249
- PrefetchedSADAAuthorizationManager (class in *pyhanko.sign.signers.csc_signer*), 181
- prefix_name (*pyhanko.pdf_utils.extensions.DeveloperExtension* attribute), 80
- prepare_object_stream() (*pyhanko.pdf_utils.writer.BasePdfFileWriter* method), 123
- prepare_providers() (*pyhanko.sign.ades.api.CAdESSignedAttrSpec* method), 133
- prepare_rw_output_stream() (in module *pyhanko.pdf_utils.misc*), 113
- prepare_sig_field() (in module *pyhanko.sign.fields*), 165
- prepare_tbs_document() (*pyhanko.sign.signers.pdf_signer.PdfSigningSession* method), 210
- prepare_write() (*pyhanko.pdf_utils.font.api.FontEngine* method), 84
- prepare_write() (*pyhanko.pdf_utils.font.opentype.GlyphAccumulator* method), 87
- PreparedByteRangeDigest (class in *pyhanko.sign.signers.pdf_byterange*), 184
- PreSignValidationStatus (class in *pyhanko.sign.signers.pdf_signer*), 213
- pretty_print_details() (*pyhanko.sign.validation.status.StandardCMSSignatureStatus* method), 242
- pretty_print_sections() (*pyhanko.sign.validation.status.PdfSignatureStatus* method), 244
- pretty_print_sections() (*pyhanko.sign.validation.status.StandardCMSSignatureStatus* method), 242
- print_signature (*pyhanko.sign.fields.VisibleSigSettings* attribute), 164
- process_bit_string_flags() (in module *pyhanko.pdf_utils.config_utils*), 55
- process_config_dict() (in module *pyhanko.config*), 253
- process_crypt_filters() (*pyhanko.pdf_utils.crypt.PubKeySecurityHandler* class method), 64
- process_crypt_filters() (*pyhanko.pdf_utils.crypt.SecurityHandler* class method), 60
- process_data_at_eof() (in module *pyhanko.pdf_utils.reader*), 118
- process_entries() (*pyhanko.config.PemDerSignatureConfig* class method), 250
- process_entries() (*pyhanko.config.PKCS11SignatureConfig* class method), 253
- process_entries() (*pyhanko.config.PKCS12SignatureConfig* class method), 249
- process_entries() (*pyhanko.config.TokenCriteria* class method), 251
- process_entries() (*pyhanko.pdf_utils.config_utils.ConfigurableMixin* class method), 54
- process_entries() (*pyhanko.pdf_utils.layout.SimpleBoxLayoutRule* class method), 105
- process_entries() (*pyhanko.pdf_utils.text.TextStyle* class method), 120
- process_entries() (*pyhanko.pdf_utils.xref.XRefSectionData* method), 130
- process_entries() (*pyhanko.sign.validation.settings.KeyUsageConstraints* class method), 237
- process_entries() (*pyhanko.stamp.BaseStampStyle* class method), 254
- process_entries() (*pyhanko.stamp.QRStampStyle* class method), 254

<i>class method</i>), 256			
process_hybrid_entries()	(py-	pyhanko.config	
<i>hanko.pdf_utils.xref.XRefSectionData method</i>), 130		module, 248	
process_oid()	(in module py-	pyhanko.pdf_utils.barcodes	
<i>hanko.pdf_utils.config_utils</i>), 55		module, 53	
process_oids()	(in module py-	pyhanko.pdf_utils.config_utils	
<i>hanko.pdf_utils.config_utils</i>), 55		module, 54	
process_pss_params()	(in module py-	pyhanko.pdf_utils.content	
<i>hanko.sign.general</i>), 168		module, 55	
PROMPT	(pyhanko.config.PKCS11PinEntryMode at-	pyhanko.pdf_utils.crypt	
<i>tribute</i>), 251		module, 57	
prompt_passphrase	(py-	pyhanko.pdf_utils.embed	
<i>hanko.config.PemDerSignatureConfig</i>	at-	module, 76	
<i>tribute</i>), 250		pyhanko.pdf_utils.extensions	
prompt_passphrase	(py-	module, 79	
<i>hanko.config.PKCS12SignatureConfig</i>	at-	pyhanko.pdf_utils.filters	
<i>tribute</i>), 249		module, 81	
prompt_pin	(pyhanko.config.PKCS11SignatureConfig at-	pyhanko.pdf_utils.font.api	
<i>tribute</i>), 252		module, 83	
PROOF_OF_APPROVAL	(py-	pyhanko.pdf_utils.font.basic	
<i>hanko.sign.ades.api.GenericCommitment</i>		module, 85	
<i>tribute</i>), 133		pyhanko.pdf_utils.font.opentype	
PROOF_OF_CREATION	(py-	module, 86	
<i>hanko.sign.ades.api.GenericCommitment</i>		pyhanko.pdf_utils.generic	
<i>tribute</i>), 133		module, 88	
PROOF_OF_DELIVERY	(py-	pyhanko.pdf_utils.images	
<i>hanko.sign.ades.api.GenericCommitment</i>		module, 99	
<i>tribute</i>), 132		pyhanko.pdf_utils.incremental_writer	
PROOF_OF_ORIGIN	(py-	module, 100	
<i>hanko.sign.ades.api.GenericCommitment</i>		pyhanko.pdf_utils.layout	
<i>tribute</i>), 132		module, 102	
PROOF_OF_RECEIPT	(py-	pyhanko.pdf_utils.metadata.model	
<i>hanko.sign.ades.api.GenericCommitment</i>		module, 106	
<i>tribute</i>), 132		pyhanko.pdf_utils.misc	
PROOF_OF_SENDER	(py-	module, 112	
<i>hanko.sign.ades.api.GenericCommitment</i>		pyhanko.pdf_utils.reader	
<i>tribute</i>), 133		module, 114	
PROPERTIES	(pyhanko.pdf_utils.content.ResourceType at-	pyhanko.pdf_utils.rw_common	
<i>tribute</i>), 55		module, 118	
PROXY	(pyhanko.pdf_utils.generic.EncryptedObjAccess at-	pyhanko.pdf_utils.text	
<i>tribute</i>), 97		module, 120	
PubKeyAdbSubFilter	(class in py-	pyhanko.pdf_utils.writer	
<i>hanko.pdf_utils.crypt</i>), 66		module, 121	
PubKeyAESCryptFilter	(class in py-	pyhanko.pdf_utils.xref	
<i>hanko.pdf_utils.crypt</i>), 73		module, 127	
PubKeyCryptFilter	(class in pyhanko.pdf_utils.crypt), 69	pyhanko.sign.ades.api	
PubKeyRC4CryptFilter	(class in py-	module, 132	
<i>hanko.pdf_utils.crypt</i>), 73		pyhanko.sign.ades.asn1_util	
PubKeySecurityHandler	(class in py-	module, 134	
<i>hanko.pdf_utils.crypt</i>), 62		pyhanko.sign.ades.cades_asn1	
put_string_line()	(pyhanko.pdf_utils.text.TextBox method), 121	module, 134	
		pyhanko.sign.ades.report	
		module, 135	
		pyhanko.sign.attributes	
		module, 137	

pyhanko.sign.beid
 module, 139
 pyhanko.sign.diff_analysis.common
 module, 141
 pyhanko.sign.diff_analysis.form_rules_api
 module, 142
 pyhanko.sign.diff_analysis.policies
 module, 144
 pyhanko.sign.diff_analysis.policy_api
 module, 146
 pyhanko.sign.diff_analysis.rules.file_structur
 module, 148
 pyhanko.sign.diff_analysis.rules.form_field_rules
 module, 149
 pyhanko.sign.diff_analysis.rules.metadata_rules
 module, 151
 pyhanko.sign.diff_analysis.rules_api
 module, 152
 pyhanko.sign.fields
 module, 153
 pyhanko.sign.general
 module, 165
 pyhanko.sign.pkcs11
 module, 170
 pyhanko.sign.signers.cms_embedder
 module, 173
 pyhanko.sign.signers.constants
 module, 182
 pyhanko.sign.signers.csc_signer
 module, 176
 pyhanko.sign.signers.functions
 module, 183
 pyhanko.sign.signers.pdf_byterange
 module, 184
 pyhanko.sign.signers.pdf_cms
 module, 187
 pyhanko.sign.signers.pdf_signer
 module, 198
 pyhanko.sign.timestamps.aiohttp_client
 module, 218
 pyhanko.sign.timestamps.api
 module, 216
 pyhanko.sign.timestamps.common_utils
 module, 220
 pyhanko.sign.timestamps.dummy_client
 module, 219
 pyhanko.sign.timestamps.requests_client
 module, 219
 pyhanko.sign.validation
 module, 244
 pyhanko.sign.validation.dss
 module, 220
 pyhanko.sign.validation.errors
 module, 224

pyhanko.sign.validation.generic_cms
 module, 225
 pyhanko.sign.validation.ltv
 module, 230
 pyhanko.sign.validation.pdf_embedded
 module, 232
 pyhanko.sign.validation.settings
 module, 236
 pyhanko.sign.validation.status
 module, 238
 pyhanko.sign.validation.utils
 module, 244
 pyhanko.stamp
 module, 253

Q

qr_inner_content (*pyhanko.stamp.QRStampStyle* attribute), 256
 qr_inner_size (*pyhanko.stamp.QRStampStyle* attribute), 256
 qr_position (*pyhanko.stamp.QRStampStyle* attribute), 256
 qr_stamp_file() (in module *pyhanko.stamp*), 259
 QRPosition (class in *pyhanko.stamp*), 257
 QRStamp (class in *pyhanko.stamp*), 258
 QRStampStyle (class in *pyhanko.stamp*), 255
 QualifiedWhitelistRule (class in *pyhanko.sign.diff_analysis.rules_api*), 152
 Qualifiers (class in *pyhanko.pdf_utils.metadata.model*), 108
 qualifiers (*pyhanko.pdf_utils.metadata.model.XmpValue* attribute), 109
 qualify() (in module *pyhanko.sign.diff_analysis.common*), 141

R

RAW (*pyhanko.pdf_utils.generic.EncryptedObjAccess* attribute), 98
 raw_get() (*pyhanko.pdf_utils.generic.ArrayObject* method), 93
 raw_get() (*pyhanko.pdf_utils.generic.DictionaryObject* method), 94
 raw_get() (*pyhanko.pdf_utils.xref.TrailerDictionary* method), 131
 raw_mechanism (*pyhanko.config.PKCS11SignatureConfig* attribute), 253
 raw_object (*pyhanko.pdf_utils.generic.DecryptedObjectProxy* attribute), 98
 RawContent (class in *pyhanko.pdf_utils.content*), 57
 RawPdfPath (class in *pyhanko.pdf_utils.reader*), 118
 RC4_40 (*pyhanko.pdf_utils.crypt.SecurityHandlerVersion* attribute), 65
 RC4_BASIC (*pyhanko.pdf_utils.crypt.StandardSecuritySettingsRevision* attribute), 65

RC4_EXTENDED (pyhanko.pdf_utils.crypt.StandardSecuritySettingsRevision attribute), 65	pyhanko.pdf_utils.generic.IndirectObject static method), 91
RC4_LONGER_KEYS (pyhanko.pdf_utils.crypt.SecurityHandlerVersion attribute), 65	read_from_stream() (pyhanko.pdf_utils.generic.NameObject static method), 93
RC4_OR_AES128 (pyhanko.pdf_utils.crypt.SecurityHandlerVersion attribute), 65	read_from_stream() (pyhanko.pdf_utils.generic.NullObject static method), 91
RC4_OR_AES128 (pyhanko.pdf_utils.crypt.StandardSecuritySettingsRevision attribute), 65	read_from_stream() (pyhanko.pdf_utils.generic.NumberObject static method), 92
RC4CryptFilterMixin (class in pyhanko.pdf_utils.crypt), 71	read_from_sv_string() (pyhanko.pdf_utils.generic.NumberObject static method), 92
rd() (in module pyhanko.pdf_utils.misc), 114	read_from_sv_string() (pyhanko.pdf_utils.generic.NumberObject static method), 92
RDF_ABOUT (in module pyhanko.pdf_utils.metadata.model), 111	read_non_whitespace() (in module pyhanko.pdf_utils.misc), 112
RDF_ALT (in module pyhanko.pdf_utils.metadata.model), 111	read_object() (in module pyhanko.pdf_utils.generic), 96
RDF_BAG (in module pyhanko.pdf_utils.metadata.model), 110	read_object_header() (in module pyhanko.pdf_utils.xref), 131
RDF_DESCRIPTION (in module pyhanko.pdf_utils.metadata.model), 111	read_until_regex() (in module pyhanko.pdf_utils.misc), 112
RDF_LI (in module pyhanko.pdf_utils.metadata.model), 111	read_until_whitespace() (in module pyhanko.pdf_utils.misc), 112
RDF_PARSE_TYPE (in module pyhanko.pdf_utils.metadata.model), 111	read_xrefs() (pyhanko.pdf_utils.xref.XRefBuilder method), 128
RDF_RDF (in module pyhanko.pdf_utils.metadata.model), 110	readable_field_name (pyhanko.sign.fields.SigFieldSpec attribute), 154
RDF_RESOURCE (in module pyhanko.pdf_utils.metadata.model), 111	reason (pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute), 199
RDF_SEQ (in module pyhanko.pdf_utils.metadata.model), 110	REASONS (pyhanko.sign.fields.SigSeedValFlags attribute), 155
RDF_VALUE (in module pyhanko.pdf_utils.metadata.model), 111	reasons (pyhanko.sign.fields.SigSeedValueSpec attribute), 157
read() (pyhanko.pdf_utils.reader.PdfFileReader method), 115	Reference (class in pyhanko.pdf_utils.generic), 88
read_certification_data() (in module pyhanko.sign.validation.pdf_embedded), 234	ReferenceUpdate (class in pyhanko.sign.diff_analysis.rules_api), 153
read_cf_dictionary() (pyhanko.pdf_utils.crypt.PubKeySecurityHandler class method), 64	refs_freed_in_revision() (pyhanko.pdf_utils.reader.HistoricalResolver method), 117
read_cf_dictionary() (pyhanko.pdf_utils.crypt.SecurityHandler class method), 60	refs_freed_in_revision() (pyhanko.pdf_utils.xref.XRefCache method), 128
read_dss() (pyhanko.sign.validation.dss.DocumentSecurityStore class method), 221	register() (pyhanko.pdf_utils.crypt.SecurityHandler static method), 58
read_from_stream() (pyhanko.pdf_utils.generic.ArrayObject static method), 94	register() (pyhanko.pdf_utils.crypt.SerialisableCredential static method), 74
read_from_stream() (pyhanko.pdf_utils.generic.BooleanObject static method), 91	register() (pyhanko.sign.general.CertificateStore method), 166
read_from_stream() (pyhanko.pdf_utils.generic.DictionaryObject static method), 94	register() (pyhanko.sign.general.SimpleCertificateStore method), 166
read_from_stream() (pyhanko.pdf_utils.generic.DictionaryObject static method), 94	register() (pyhanko.stamp.BaseStamp method), 258
read_from_stream() (pyhanko.pdf_utils.generic.DictionaryObject static method), 94	register_annotation() (pyhanko.pdf_utils.generic.DictionaryObject static method), 94

hanko.pdf_utils.writer.BasePdfFileWriter
method), 123
register_cms_attribute() (in module *py-*
hanko.sign.ades.asn1_util), 134
register_crypt_filter() (py-
hanko.pdf_utils.crypt.SecurityHandler class
method), 60
register_extension() (py-
hanko.pdf_utils.writer.BasePdfFileWriter
method), 122
register_multiple() (py-
hanko.sign.general.CertificateStore *method*),
 166
register_vri() (*pyhanko.sign.validation.dss.DocumentSignature*
method), 221
register_widget_annotation() (py-
hanko.sign.fields.SignatureFormField *method*),
 163
RelatedFileSpec (class in *pyhanko.pdf_utils.embed*),
 78
render() (*pyhanko.pdf_utils.barcodes.BarcodeBox*
method), 53
render() (*pyhanko.pdf_utils.content.ImportedPdfPage*
method), 57
render() (*pyhanko.pdf_utils.content.PdfContent*
method), 56
render() (*pyhanko.pdf_utils.content.RawContent*
method), 57
render() (*pyhanko.pdf_utils.images.PdfImage* *method*),
 99
render() (*pyhanko.pdf_utils.text.TextBox* *method*), 121
render() (*pyhanko.stamp.BaseStamp* *method*), 258
report_seed_value_validation() (in module *py-*
hanko.sign.validation.pdf_embedded), 235
request_cms() (*pyhanko.sign.timestamps.api.TimeStamper*
method), 216
request_headers() (py-
hanko.sign.timestamps.requests_client.HTTPTimeStamper
method), 219
request_tsa_response() (py-
hanko.sign.timestamps.dummy_client.DummyTimeStamper
method), 219
RESERVED (*pyhanko.sign.fields.SigCertConstraintFlags*
attribute), 159
reserved_region_end (py-
hanko.sign.signers.pdf_byterange.PreparedByteRangeDigest
attribute), 185
reserved_region_start (py-
hanko.sign.signers.pdf_byterange.PreparedByteRangeDigest
attribute), 185
ResourceManagementError, 55
resources (*pyhanko.pdf_utils.content.PdfContent* prop-
erty), 56
ResourceType (class in *pyhanko.pdf_utils.content*), 55
response_data (*pyhanko.sign.signers.csc_signer.CSCCredentialInfo*
attribute), 179
resume_signing() (py-
hanko.sign.signers.pdf_signer.PdfTBSDocument
class method), 212
retrieve_adobe_revocation_info() (in module *py-*
hanko.sign.validation.ltv), 230
retrieve_by_issuer_serial() (py-
hanko.sign.general.SimpleCertificateStore
method), 167
retrieve_by_name() (py-
hanko.sign.general.SimpleCertificateStore
method), 167
retrieve_by_serial_number_by_key_identifier() (py-
hanko.sign.general.SimpleCertificateStore
method), 166
retroactive_revinfo (*pyhanko.config.CLConfig* *at-*
tribute), 248
review_file() (*pyhanko.sign.diff_analysis.policies.StandardDiffPolicy*
method), 145
review_file() (*pyhanko.sign.diff_analysis.policy_api.DiffPolicy*
method), 147
RevocationInfoValidationType (class in *py-*
hanko.sign.validation.ltv), 230
REVOKED (*pyhanko.sign.ades.report.AdESFailure* *at-*
tribute), 136
revoked (*pyhanko.sign.validation.status.SignatureStatus*
property), 238
REVOKED_CA_NO_POE (py-
hanko.sign.ades.report.AdESIndeterminate
attribute), 136
REVOKED_NO_POE (*pyhanko.sign.ades.report.AdESIndeterminate*
attribute), 136
RFC
 RFC 3161, 8, 9, 33, 216
 RFC 5126, 24
 RFC 5280, 18, 161, 236, 237
 RFC 5652, 4, 27, 41, 165
 RFC 5755, 228, 231, 235
 RFC 6211, 229
 RFC 8933, 216–218
right (*pyhanko.pdf_utils.layout.Margins* *attribute*), 103
RIGHT_OF_TEXT (*pyhanko.stamp.QRPosition* *attribute*),
 257
root (*pyhanko.pdf_utils.rw_common.PdfHandler* prop-
erty), 119
root_ref (*pyhanko.pdf_utils.reader.HistoricalResolver*
property), 117
root_ref (*pyhanko.pdf_utils.reader.PdfFileReader*
property), 114
root_ref (*pyhanko.pdf_utils.rw_common.PdfHandler*
property), 119
root_ref (*pyhanko.pdf_utils.writer.BasePdfFileWriter*
property), 122

rotate_with_page (pyhanko.sign.fields.VisibleSigSettings attribute), 164
S
 S3 (pyhanko.pdf_utils.crypt.PubKeyAdbSubFilter attribute), 66
 S4 (pyhanko.pdf_utils.crypt.PubKeyAdbSubFilter attribute), 66
 S5 (pyhanko.pdf_utils.crypt.PubKeyAdbSubFilter attribute), 66
 sad (pyhanko.sign.signers.csc_signer.CSCAuthorizationInfo attribute), 180
 safe_whitelist() (in module pyhanko.sign.diff_analysis.common), 141
 satisfied_by() (pyhanko.sign.fields.SigCertConstraints method), 157
 save() (pyhanko.pdf_utils.barcodes.PdfStreamBarcodeWriter method), 53
 scale_with_page_zoom (pyhanko.sign.fields.VisibleSigSettings attribute), 164
 security_handler (pyhanko.pdf_utils.writer.PdfFileWriter attribute), 126
 SecurityHandler (class in pyhanko.pdf_utils.crypt), 58
 SecurityHandlerVersion (class in pyhanko.pdf_utils.crypt), 65
 seed_signature_type (pyhanko.sign.fields.SigSeedValueSpec attribute), 158
 seed_value_constraint_error (pyhanko.sign.validation.status.PdfSignatureStatus attribute), 243
 seed_value_dict (pyhanko.sign.fields.SigFieldSpec attribute), 154
 seed_value_ok (pyhanko.sign.validation.status.PdfSignatureStatus property), 244
 seed_value_spec (pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature property), 233
 SeedLockDocument (class in pyhanko.sign.fields), 160
 SeedValueDictVersion (class in pyhanko.sign.fields), 160
 select_pkcs11_signing_params() (in module pyhanko.sign.pkcs11), 172
 select_suitable_signing_md() (in module pyhanko.sign.signers.pdf_cms), 198
 self_reported_timestamp (pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature property), 232
 SEPARATE_REVISION (pyhanko.sign.signers.pdf_signer.SigDSSPlacementPreference attribute), 203
 serial (pyhanko.config.TokenCriteria attribute), 250
 SerialisableCredential (class in pyhanko.pdf_utils.crypt), 74
 serialise() (pyhanko.pdf_utils.crypt.SerialisableCredential method), 75
 SerialisedCredential (class in pyhanko.pdf_utils.crypt), 74
 service_url (pyhanko.sign.signers.csc_signer.CSCServiceSessionInfo attribute), 178
 set_custom_trailer_entry() (pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter method), 101
 set_custom_trailer_entry() (pyhanko.pdf_utils.writer.BasePdfFileWriter method), 122
 set_custom_trailer_entry() (pyhanko.pdf_utils.writer.PdfFileWriter method), 126
 set_embedded_only() (pyhanko.pdf_utils.crypt.CryptFilter method), 68
 set_hidden_flag (pyhanko.sign.fields.InvisSigSettings attribute), 163
 set_info() (pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter method), 101
 set_info() (pyhanko.pdf_utils.writer.BasePdfFileWriter method), 122
 set_print_flag (pyhanko.sign.fields.InvisSigSettings attribute), 163
 set_resource() (pyhanko.pdf_utils.content.PdfContent method), 56
 set_security_handler() (pyhanko.pdf_utils.crypt.CryptFilterConfiguration method), 66
 set_tsp_headers() (in module pyhanko.sign.timestamps.common_utils), 220
 set_writer() (pyhanko.pdf_utils.content.PdfContent method), 57
 setdefault() (pyhanko.pdf_utils.generic.DictionaryObject method), 94
 SHADING (pyhanko.pdf_utils.content.ResourceType attribute), 55
 shape() (pyhanko.pdf_utils.font.api.FontEngine method), 84
 shape() (pyhanko.pdf_utils.font.basic.SimpleFontEngine method), 85
 shape() (pyhanko.pdf_utils.font.opentype.GlyphAccumulator method), 86
 ShapeResult (class in pyhanko.pdf_utils.font.api), 83
 signed_key (pyhanko.pdf_utils.crypt.CryptFilter property), 68
 SHRINK_TO_FIT (pyhanko.pdf_utils.layout.InnerScaling attribute), 105
 SIG_CONSTRAINTS_FAILURE (pyhanko.pdf_utils.layout.InnerScaling attribute), 105

hanko.sign.ades.report.AdESIndeterminate
attribute), 136

sig_content_identifier() (py-
hanko.sign.validation.dss.DocumentSecurityStore
static method), 220

SIG_CRYPTO_FAILURE (py-
hanko.sign.ades.report.AdESFailure *attribute*),
 136

SIG_DETAILS_DEFAULT_TEMPLATE (in module py-
hanko.sign.signers.constants), 182

sig_field (pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature
attribute), 232

sig_field_name (pyhanko.sign.fields.SigFieldSpec *at-*
tribute), 154

sig_object (pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature
attribute), 232

sig_object_type (py-
hanko.sign.validation.pdf_embedded.EmbeddedPdfSignature
property), 232

sig_placeholder (py-
hanko.sign.signers.cms_embedder.SigObjSetup
attribute), 174

SigAppearanceSetup (class in py-
hanko.sign.signers.cms_embedder), 175

SigCertConstraintFlags (class in py-
hanko.sign.fields), 159

SigCertConstraints (class in pyhanko.sign.fields),
 156

SigCertKeyUsage (class in pyhanko.sign.fields), 160

SigDSSPlacementPreference (class in py-
hanko.sign.signers.pdf_signer), 202

SigFieldCreationRule (class in py-
hanko.sign.diff_analysis.rules.form_field_rules),
 149

SigFieldModificationRule (class in py-
hanko.sign.diff_analysis.rules.form_field_rules),
 149

SigFieldSpec (class in pyhanko.sign.fields), 153

SigIOSetup (class in py-
hanko.sign.signers.cms_embedder), 175

SigMDPSetup (class in py-
hanko.sign.signers.cms_embedder), 174

sign() (pyhanko.sign.signers.pdf_cms.Signer *method*),
 192

sign_general_data() (py-
hanko.sign.signers.pdf_cms.Signer *method*),
 194

sign_pdf() (in module pyhanko.sign.signers.functions),
 183

sign_pdf() (pyhanko.sign.signers.pdf_signer.PdfSigner
method), 209

sign_prescribed_attributes() (py-
hanko.sign.signers.pdf_cms.Signer *method*),
 193

sign_raw() (pyhanko.sign.signers.pdf_cms.SimpleSigner
method), 196

signature_mechanism (py-
hanko.sign.signers.pdf_cms.Signer *attribute*),
 188

signature_policy_identifier (py-
hanko.sign.ades.api.CAdESSignedAttrSpec
attribute), 133

SIGNATURE_POLICY_NOT_AVAILABLE (py-
hanko.sign.ades.report.AdESIndeterminate
attribute), 136

SignatureCoverageLevel (class in py-
hanko.sign.validation.status), 242

SignatureFormField (class in pyhanko.sign.fields),
 162

SignatureObject (class in py-
hanko.sign.signers.pdf_byterange), 186

SignaturePolicyDocument (class in py-
hanko.sign.ades.cades_asn1), 135

SignaturePolicyId (class in py-
hanko.sign.ades.cades_asn1), 134

SignaturePolicyIdentifier (class in py-
hanko.sign.ades.cades_asn1), 135

SignaturePolicyStore (class in py-
hanko.sign.ades.cades_asn1), 135

SignatureStatus (class in py-
hanko.sign.validation.status), 238

SignatureValidationError, 224

signed_attr_prov_spec (py-
hanko.sign.signers.pdf_cms.ExternalSigner
attribute), 197

signed_attr_providers() (py-
hanko.sign.attributes.SignedAttributeProviderSpec
method), 137

signed_attrs() (pyhanko.sign.signers.pdf_cms.Signer
method), 189

signed_data (pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature
attribute), 232

SIGNED_DATA_NOT_FOUND (py-
hanko.sign.ades.report.AdESIndeterminate
attribute), 137

SignedAssertion (class in py-
hanko.sign.ades.cades_asn1), 135

SignedAssertionId (class in py-
hanko.sign.ades.cades_asn1), 135

SignedAssertions (class in py-
hanko.sign.ades.cades_asn1), 135

SignedAttributeProviderSpec (class in py-
hanko.sign.attributes), 137

SignedDataCerts (class in pyhanko.sign.general), 167

Signer (class in pyhanko.sign.signers.pdf_cms), 187

signer_attributes (py-
hanko.sign.ades.api.CAdESSignedAttrSpec
attribute), 133

signer_cert (*pyhanko.sign.general.SignedDataCertificate* attribute), 167
signer_cert (*pyhanko.sign.validation.pdf_embedded.EmbeddedCertificate* property), 232
SIGNER_DISCRETION (*pyhanko.sign.fields.SeedLockDocument* attribute), 160
signer_info() (*pyhanko.sign.signers.pdf_cms.Signer* method), 188
signer_key_usage (*pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata* attribute), 200
signer_path (*pyhanko.sign.signers.pdf_signer.PreSignValidation* attribute), 214
signer_reported_dt (*pyhanko.sign.validation.status.StandardCMSSignatureStatus* attribute), 241
SignerAttributesV2 (class in *pyhanko.sign.ades.cades_asn1*), 135
SignerAttrSpec (class in *pyhanko.sign.ades.api*), 133
signing_cert (*pyhanko.sign.pkcs11.PKCS11Signer* property), 171
signing_cert (*pyhanko.sign.signers.csc_signer.CSCCredential* attribute), 179
signing_cert (*pyhanko.sign.signers.pdf_cms.Signer* attribute), 187
signing_cert (*pyhanko.sign.validation.status.SignatureStatus* attribute), 238
signing_certificate (*pyhanko.config.PKCS11SignatureConfig* attribute), 252
signing_key (*pyhanko.sign.signers.pdf_cms.SimpleSigner* attribute), 195
SigningCertificateV2Provider (class in *pyhanko.sign.attributes*), 138
SigningError, 167
SigningTimeProvider (class in *pyhanko.sign.attributes*), 137
SigObjSetup (class in *pyhanko.sign.signers.cms_embedder*), 174
SigPolicyQualifierId (class in *pyhanko.sign.ades.cades_asn1*), 134
SigPolicyQualifierInfo (class in *pyhanko.sign.ades.cades_asn1*), 134
SigPolicyQualifierInfos (class in *pyhanko.sign.ades.cades_asn1*), 134
SigSeedSubFilter (class in *pyhanko.sign.fields*), 160
SigSeedValFlags (class in *pyhanko.sign.fields*), 155
SigSeedValueSpec (class in *pyhanko.sign.fields*), 157
SigSeedValueValidationError, 224
simple_cms_attribute() (in module *pyhanko.sign.general*), 165
SimpleBoxLayoutRule (class in *pyhanko.pdf_utils.layout*), 105
SimpleCertificateStore (class in *pyhanko.sign.general*), 166
SimplePKCS11KeyDecrypter (class in *pyhanko.pdf_utils.crypt*), 73
SimpleFontEngine (class in *pyhanko.pdf_utils.font.basic*), 85
SimpleFontEngineFactory (class in *pyhanko.pdf_utils.font.basic*), 85
SimpleFontMeta (class in *pyhanko.pdf_utils.font.basic*), 85
SimpleSigner (class in *pyhanko.sign.signers.pdf_cms*), 195
SimpleStamp (*pyhanko.pdf_utils.misc.ConsList* static method), 114
Singleton (class in *pyhanko.pdf_utils.misc*), 114
SignatureStatus (*pyhanko.pdf_utils.xref.XRefSectionMetaInfo* attribute), 130
SKIP (*pyhanko.config.PKCS11PinEntryMode* attribute), 251
skip_if_unneeded (*pyhanko.sign.signers.pdf_signer.GeneralDSSContentSettings* attribute), 202
skip_over_comment() (in module *pyhanko.pdf_utils.misc*), 113
skip_over_whitespace() (in module *pyhanko.pdf_utils.misc*), 113
slot_no (*pyhanko.config.PKCS11SignatureConfig* attribute), 252
SPDocSpecification (class in *pyhanko.sign.ades.cades_asn1*), 134
SPUserNotice (class in *pyhanko.sign.ades.cades_asn1*), 134
STAMP_ART_CONTENT (in module *pyhanko.stamp*), 260
stamp_styles (*pyhanko.config.CLIconfig* attribute), 248
stamp_text (*pyhanko.stamp.QRStampStyle* attribute), 256
stamp_text (*pyhanko.stamp.TextStampStyle* attribute), 255
STANDARD (*pyhanko.pdf_utils.xref.XRefSectionType* attribute), 130
STANDARD (*pyhanko.pdf_utils.xref.XRefType* attribute), 129
standard_filters() (*pyhanko.pdf_utils.crypt.CryptFilterConfiguration* method), 67
StandardAESCryptFilter (class in *pyhanko.pdf_utils.crypt*), 73
StandardCMSSignatureStatus (class in *pyhanko.sign.validation.status*), 240
StandardCryptFilter (class in *pyhanko.pdf_utils.crypt*), 68
StandardDiffPolicy (class in *pyhanko.sign.diff_analysis.policies*), 144

StandardRC4CryptFilter (class in pyhanko.pdf_utils.crypt), 73
 StandardSecurityHandler (class in pyhanko.pdf_utils.crypt), 61
 StandardSecuritySettingsRevision (class in pyhanko.pdf_utils.crypt), 65
 start_location (pyhanko.pdf_utils.xref.XRefSectionMetaInfo attribute), 130
 StaticContentStamp (class in pyhanko.stamp), 259
 StaticStampStyle (class in pyhanko.stamp), 256
 status (pyhanko.pdf_utils.crypt.AuthResult attribute), 65
 status (pyhanko.sign.adess.report.AdESSubIndic property), 136
 STDERR (pyhanko.config.StdLogOutput attribute), 248
 StdLogOutput (class in pyhanko.config), 248
 STDOUT (pyhanko.config.StdLogOutput attribute), 248
 STREAM (pyhanko.pdf_utils.xref.XRefSectionType attribute), 130
 stream_filter_name (pyhanko.pdf_utils.crypt.CryptFilterConfiguration property), 67
 stream_ref (pyhanko.pdf_utils.xref.XRefSectionMetaInfo attribute), 130
 stream_xrefs (pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter attribute), 101
 stream_xrefs (pyhanko.pdf_utils.writer.BasePdfFileWriter attribute), 121
 stream_xrefs (pyhanko.pdf_utils.writer.PdfFileWriter attribute), 126
 StreamObject (class in pyhanko.pdf_utils.generic), 94
 STRETCH_FILL (pyhanko.pdf_utils.layout.InnerScaling attribute), 105
 STRETCH_TO_FIT (pyhanko.pdf_utils.layout.InnerScaling attribute), 105
 string_filter_name (pyhanko.pdf_utils.crypt.CryptFilterConfiguration property), 67
 StringWithLanguage (class in pyhanko.pdf_utils.misc), 112
 strip_filters() (pyhanko.pdf_utils.generic.StreamObject method), 95
 style (pyhanko.sign.signers.cms_embedder.SigAppearanceSetup attribute), 175
 SUBFILTER (pyhanko.sign.fields.SigSeedValFlags attribute), 155
 subfilter (pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute), 200
 subfilters (pyhanko.sign.fields.SigSeedValueSpec attribute), 157
 subject (pyhanko.pdf_utils.metadata.model.DocumentMetadata attribute), 107
 SUBJECT (pyhanko.sign.fields.SigCertConstraintFlags attribute), 159
 SUBJECT_DN (pyhanko.sign.fields.SigCertConstraintFlags attribute), 159
 subject_dn (pyhanko.sign.fields.SigCertConstraints attribute), 156
 subject_name (pyhanko.sign.signers.pdf_cms.Signer property), 188
 subjects (pyhanko.sign.fields.SigCertConstraints attribute), 156
 subsets (pyhanko.pdf_utils.font.api.FontSubsetCollection attribute), 84
 substitute_margins() (pyhanko.pdf_utils.layout.SimpleBoxLayoutRule method), 105
 subsumed_by (pyhanko.pdf_utils.extensions.DeveloperExtension attribute), 80
 subsumes (pyhanko.pdf_utils.extensions.DeveloperExtension attribute), 80
 summarise_integrity_info() (pyhanko.sign.validation.pdf_embedded.EmbeddedPdfSignature method), 233
 summary() (pyhanko.sign.validation.status.SignatureStatus method), 238
 summary_fields() (pyhanko.sign.validation.status.PdfSignatureStatus method), 244
 summary_fields() (pyhanko.sign.validation.status.SignatureStatus method), 238
 summary_fields() (pyhanko.sign.validation.status.StandardCMSSignatureStatus method), 242
 supply_dss_in_writer() (pyhanko.sign.validation.dss.DocumentSecurityStore class method), 221
 support_generic_subfilters() (pyhanko.pdf_utils.crypt.PubKeySecurityHandler class method), 63
 support_generic_subfilters() (pyhanko.pdf_utils.crypt.SecurityHandler class method), 59
 supported_mechanisms (pyhanko.sign.signers.csc_signer.CSCCredencialInfo attribute), 179
 SuspiciousModification, 146
 sv_dict_version (pyhanko.sign.fields.SigSeedValueSpec attribute), 158

T

tail (pyhanko.pdf_utils.misc.ConsList attribute), 113
 text_box_style (pyhanko.stamp.TextStampStyle attribute), 255

text_params (pyhanko.sign.signers.cms_embedder.SigAppearanceSettings attribute), 175
 text_stamp_file() (in module pyhanko.stamp), 259
 TextBox (class in pyhanko.pdf_utils.text), 120
 TextBoxStyle (class in pyhanko.pdf_utils.text), 120
 TextStamp (class in pyhanko.stamp), 258
 TextStampStyle (class in pyhanko.stamp), 254
 TextStringEncoding (class in pyhanko.pdf_utils.generic), 97
 TextStringObject (class in pyhanko.pdf_utils.generic), 92
 TextStyle (class in pyhanko.pdf_utils.text), 120
 tight_size_estimates (pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute), 201
 tight_size_estimates (pyhanko.sign.signers.pdf_signer.PostSignInstructions attribute), 215
 time_tolerance (pyhanko.config.CLIConfig attribute), 248
 timestamp (pyhanko.sign.signers.cms_embedder.SigAppearanceSettings attribute), 175
 timestamp (pyhanko.sign.validation.status.TimestampSignatureStatus attribute), 239
 timestamp_content (pyhanko.sign.ades.api.CAdESSignedAttrSpec attribute), 133
 timestamp_field_name (pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata attribute), 200
 timestamp_field_name (pyhanko.sign.signers.pdf_signer.PostSignInstructions attribute), 215
 timestamp_format (pyhanko.stamp.TextStampStyle attribute), 255
 timestamp_md_algorithm (pyhanko.sign.signers.pdf_signer.PostSignInstructions attribute), 215
 TIMESTAMP_ORDER_FAILURE (pyhanko.sign.ades.report.AdESIndeterminate attribute), 136
 timestamp_pdf() (pyhanko.sign.signers.pdf_signer.PdfTimeStamper method), 203
 timestamp_required (pyhanko.sign.fields.SigSeedValueSpec attribute), 157
 timestamp_server_url (pyhanko.sign.fields.SigSeedValueSpec attribute), 157
 timestamp_validity (pyhanko.sign.validation.status.StandardCMSSignatureStatus attribute), 241
 TimestampDSSContentSettings (class in pyhanko.sign.signers.pdf_signer), 202
 TimeStamper (class in pyhanko.sign.timestamps.api), 216
 timestamp (pyhanko.sign.signers.pdf_signer.PostSignInstructions attribute), 215
 TimestampRequestError, 220
 TimestampSignatureStatus (class in pyhanko.sign.validation.status), 239
 title (pyhanko.pdf_utils.metadata.model.DocumentMetadata attribute), 107
 TOGETHER_WITH_NEXT_TS (pyhanko.sign.signers.pdf_signer.SigDSSPlacementPreference attribute), 203
 TOGETHER_WITH_SIGNATURE (pyhanko.sign.signers.pdf_signer.SigDSSPlacementPreference attribute), 203
 token_criteria (pyhanko.config.PKCS11SignatureConfig attribute), 252
 TokenCriteria (class in pyhanko.config), 250
 top (pyhanko.pdf_utils.layout.Margins attribute), 103
 total_revisions (pyhanko.pdf_utils.reader.PdfFileReader property), 114
 total_revisions (pyhanko.pdf_utils.xref.XRefCache property), 127
 trailer_view (pyhanko.pdf_utils.reader.HistoricalResolver property), 117
 trailer_view (pyhanko.pdf_utils.reader.PdfFileReader property), 114
 trailer_view (pyhanko.pdf_utils.rw_common.PdfHandler property), 119
 trailer_view (pyhanko.pdf_utils.writer.BasePdfFileWriter property), 123
 TrailerDictionary (class in pyhanko.pdf_utils.xref), 130
 TrailerReference (class in pyhanko.pdf_utils.generic), 89
 TRANSPARENT (pyhanko.pdf_utils.generic.EncryptedObjAccess attribute), 97
 trust_problem_indic (pyhanko.sign.validation.status.SignatureStatus attribute), 238
 trusted (pyhanko.sign.validation.status.SignatureStatus property), 238
 TRY_LATER (pyhanko.sign.ades.report.AdESIndeterminate attribute), 136
 try_resolve() (pyhanko.pdf_utils.xref.XRefSectionData method), 130
 ts_validation_paths (pyhanko.sign.signers.pdf_signer.PreSignValidationStatus attribute), 214
 TSProvider (class in pyhanko.sign.attributes), 139

U

- `uf_related_files` (`pyhanko.pdf_utils.embed.FileSpec` attribute), 78
- `UnacceptableSignerError`, 167
- `UNCLEAR` (`pyhanko.sign.validation.status.SignatureCoverageLevel` attribute), 242
- `uniform()` (`pyhanko.pdf_utils.layout.Margins` class method), 104
- `unknown_attrs_present` (`pyhanko.sign.validation.status.CADESSignerAttributeAssertions` attribute), 240
- `UNORDERED` (`pyhanko.pdf_utils.metadata.model.XmpArrayType` attribute), 109
- `unordered()` (`pyhanko.pdf_utils.metadata.model.XmpArrayType` class method), 110
- `unsigned_attr_spec` (`pyhanko.sign.signers.pdf_cms.ExternalSigner` attribute), 197
- `unsigned_attr_providers()` (`pyhanko.sign.attributes.UnsignedAttributeProviderSpec` method), 137
- `unsigned_attrs()` (`pyhanko.sign.signers.pdf_cms.Signer` method), 189
- `UnsignedAttributeProviderSpec` (class in `pyhanko.sign.attributes`), 137
- `UNSUPPORTED` (`pyhanko.sign.fields.SigCertConstraintFlags` attribute), 160
- `update_archival_timestamp_chain()` (`pyhanko.sign.signers.pdf_signer.PdfTimeStamper` method), 205
- `update_before_ts` (`pyhanko.sign.signers.pdf_signer.TimestampDSSConventions` attribute), 202
- `update_container()` (`pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter` method), 100
- `update_container()` (`pyhanko.pdf_utils.writer.BasePdfFileWriter` method), 122
- `update_root()` (`pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter` method), 100
- `update_root()` (`pyhanko.pdf_utils.writer.BasePdfFileWriter` method), 122
- `updated_ref` (`pyhanko.sign.diff_analysis.rules_api.ReferenceUpdate` attribute), 153
- `url` (`pyhanko.pdf_utils.extensions.DeveloperExtension` attribute), 80
- `URL` (`pyhanko.sign.fields.SigCertConstraintFlags` attribute), 159
- `url_type` (`pyhanko.sign.fields.SigCertConstraints` attribute), 156
- `use_pades_lta` (`pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata` attribute), 200
- `USER` (`pyhanko.pdf_utils.crypt.AuthStatus` attribute), 65
- `user_pin` (`pyhanko.config.PKCS11SignatureConfig` attribute), 252
- `uses_complex_positioning` (`pyhanko.pdf_utils.font.api.FontEngine` property), 84
- `uses_complex_positioning` (`pyhanko.pdf_utils.font.basic.SimpleFontEngine` property), 85
- `UTF16BE` (`pyhanko.pdf_utils.generic.TextStringEncoding` attribute), 97
- `UTF16LE` (`pyhanko.pdf_utils.generic.TextStringEncoding` attribute), 97
- `UTF8` (`pyhanko.pdf_utils.generic.TextStringEncoding` attribute), 97

V

- `V` (`pyhanko.sign.fields.SigSeedValFlags` attribute), 155
- `valid` (`pyhanko.sign.validation.status.CADESSignerAttributeAssertions` property), 240
- `valid` (`pyhanko.sign.validation.status.SignatureStatus` attribute), 238
- `valid_when_certifying` (`pyhanko.sign.diff_analysis.form_rules_api.FormUpdate` attribute), 143
- `valid_when_locked` (`pyhanko.sign.diff_analysis.form_rules_api.FormUpdate` attribute), 143
- `validate()` (`pyhanko.sign.validation.settings.KeyUsageConstraints` method), 237
- `validate_algorithm_protection()` (in module `pyhanko.sign.validation.generic_cms`), 229
- `validate_cert_usage()` (`pyhanko.sign.validation.status.SignatureStatus` class method), 238
- `validate_cms_signature()` (in module `pyhanko.sign.validation`), 245
- `validate_detached_cms()` (in module `pyhanko.sign.validation`), 246
- `validate_pdf_ltv_signature()` (in module `pyhanko.sign.validation`), 246
- `validate_pdf_signature()` (in module `pyhanko.sign.validation`), 244
- `validate_pdf_timestamp()` (in module `pyhanko.sign.validation`), 246
- `validate_raw()` (in module `pyhanko.sign.validation.utils`), 244
- `validate_sig_integrity()` (in module `pyhanko.sign.validation.generic_cms`), 225
- `validate_tst_signed_data()` (in module `pyhanko.sign.validation.generic_cms`), 227
- `validation_context` (`pyhanko.sign.signers.pdf_signer.PdfSignatureMetadata` attribute), 200

validation_contexts (pyhanko.config.CLIFConfig attribute), 248
 validation_info (pyhanko.sign.signers.pdf_signer.PostSignInstructions attribute), 215
 validation_path (pyhanko.sign.validation.status.SignatureStatus attribute), 238
 validation_paths (pyhanko.sign.signers.pdf_signer.PreSignValidationStatus attribute), 214
 validation_paths() (pyhanko.sign.timestamps.api.TimeStamper method), 216
 validation_results (pyhanko.sign.validation.status.CertifiedAttributeInfo attribute), 239
 ValidationInfoReadingError, 224
 value (pyhanko.pdf_utils.metadata.model.XmpValue attribute), 109
 value (pyhanko.pdf_utils.misc.StringWithLanguage attribute), 112
 ValueErrorWithMessage, 170
 VENDOR (in module pyhanko.pdf_utils.metadata.model), 108
 vertical_text (pyhanko.pdf_utils.text.TextBoxStyle attribute), 120
 view_over() (pyhanko.pdf_utils.metadata.model.DocumentMetadata method), 108
 visible_sig_settings (pyhanko.sign.fields.SigFieldSpec attribute), 155
 VisibleSigSettings (class in pyhanko.sign.fields), 164
 VRI (class in pyhanko.sign.validation.dss), 220
W
 WeakHashAlgorithmError, 224
 WhitelistRule (class in pyhanko.sign.diff_analysis.rules_api), 152
 width (pyhanko.pdf_utils.layout.BoxConstraints property), 102
 width_defined (pyhanko.pdf_utils.layout.BoxConstraints property), 102
 widths (pyhanko.pdf_utils.font.basic.SimpleFontMeta attribute), 86
 wrap_encrypted_payload() (in module pyhanko.pdf_utils.embed), 78
 write() (pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter method), 101
 write() (pyhanko.pdf_utils.writer.BasePdfFileWriter method), 123
 write_cms() (pyhanko.sign.signers.cms_embedder.PdfCMSEmbedder method), 173
 write_in_place() (pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter method), 101
 write_to_stream() (pyhanko.pdf_utils.embed.EmbeddedFileObject method), 76
 write_to_stream() (pyhanko.pdf_utils.generic.ArrayObject method), 93
 write_to_stream() (pyhanko.pdf_utils.generic.BooleanObject method), 91
 write_to_stream() (pyhanko.pdf_utils.generic.ByteStringObject method), 92
 write_to_stream() (pyhanko.pdf_utils.generic.DecryptedObjectProxy method), 98
 write_to_stream() (pyhanko.pdf_utils.generic.DictionaryObject method), 94
 write_to_stream() (pyhanko.pdf_utils.generic.FloatObject method), 91
 write_to_stream() (pyhanko.pdf_utils.generic.IndirectObject method), 90
 write_to_stream() (pyhanko.pdf_utils.generic.NameObject method), 93
 write_to_stream() (pyhanko.pdf_utils.generic.NullObject method), 91
 write_to_stream() (pyhanko.pdf_utils.generic.NumberObject method), 92
 write_to_stream() (pyhanko.pdf_utils.generic.PdfObject method), 90
 write_to_stream() (pyhanko.pdf_utils.generic.StreamObject method), 96
 write_to_stream() (pyhanko.pdf_utils.generic.TextStringObject method), 92
 write_to_stream() (pyhanko.pdf_utils.xref.TrailerDictionary method), 131
 write_to_stream() (pyhanko.pdf_utils.xref.XRefStream method), 132
 write_xref_table() (in module pyhanko.pdf_utils.xref), 131
 writer (pyhanko.pdf_utils.content.PdfContent attribute),

56
 writing_direction (pyhanko.pdf_utils.font.opentype.GlyphAccumulator attribute), 87

X

X509AttributeInfo (class in pyhanko.sign.validation.status), 239
 x_advance (pyhanko.pdf_utils.font.api.ShapeResult attribute), 83
 x_align (pyhanko.pdf_utils.layout.SimpleBoxLayoutRule attribute), 105
 x_pos (pyhanko.pdf_utils.layout.Positioning attribute), 106
 x_scale (pyhanko.pdf_utils.layout.Positioning attribute), 106
 X_XMPMETA (in module pyhanko.pdf_utils.metadata.model), 111
 X_XMPMETA (in module pyhanko.pdf_utils.metadata.model), 111
 XML_LANG (in module pyhanko.pdf_utils.metadata.model), 110
 XMP_CREATEDATE (in module pyhanko.pdf_utils.metadata.model), 111
 XMP_CREATORTOOL (in module pyhanko.pdf_utils.metadata.model), 111
 xmp_extra (pyhanko.pdf_utils.metadata.model.DocumentMetadata attribute), 107
 XMP_MODDATE (in module pyhanko.pdf_utils.metadata.model), 111
 xmp_unmanaged (pyhanko.pdf_utils.metadata.model.DocumentMetadata attribute), 107
 XmpArray (class in pyhanko.pdf_utils.metadata.model), 109
 XmpArrayType (class in pyhanko.pdf_utils.metadata.model), 109
 XmpStructure (class in pyhanko.pdf_utils.metadata.model), 109
 XmpValue (class in pyhanko.pdf_utils.metadata.model), 109
 XOBJECT (pyhanko.pdf_utils.content.ResourceType attribute), 55
 xref_data (pyhanko.pdf_utils.xref.XRefSection attribute), 129
 xref_section_type (pyhanko.pdf_utils.xref.XRefSectionMetaInfo attribute), 130
 xref_type (pyhanko.pdf_utils.xref.XRefEntry attribute), 129
 XRefBuilder (class in pyhanko.pdf_utils.xref), 128
 XRefCache (class in pyhanko.pdf_utils.xref), 127
 XRefEntry (class in pyhanko.pdf_utils.xref), 129
 xrefs (pyhanko.pdf_utils.reader.PdfFileReader attribute), 114
 XRefSection (class in pyhanko.pdf_utils.xref), 129
 XRefSectionData (class in pyhanko.pdf_utils.xref), 129
 XRefSectionMetaInfo (class in pyhanko.pdf_utils.xref), 130
 XRefSectionType (class in pyhanko.pdf_utils.xref), 130
 XRefStream (class in pyhanko.pdf_utils.xref), 132
 XrefStreamRule (class in pyhanko.sign.diff_analysis.rules.file_structure_rules), 148
 XRefType (class in pyhanko.pdf_utils.xref), 128

Y

y_advance (pyhanko.pdf_utils.font.api.ShapeResult attribute), 83
 y_align (pyhanko.pdf_utils.layout.SimpleBoxLayoutRule attribute), 105
 y_pos (pyhanko.pdf_utils.layout.Positioning attribute), 106
 y_scale (pyhanko.pdf_utils.layout.Positioning attribute), 106