
pyHanko

Release 0.1.0

Matthias Valvekens

Dec 30, 2020

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	7
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	10
1.3	Stamping PDF files	12
1.4	Configuration options	12
1.4.1	Config file location	12
1.4.2	Configuration options	12
2	Library (SDK) user's guide	15
2.1	Reading and writing PDF files	15
2.1.1	Reading files	15
2.1.2	Modifying files	16
2.2	Signature fields	17
2.2.1	General API design	17
2.2.2	Positioning	17
2.2.3	Seed value settings	18
2.2.4	Document modification policy settings	19
2.3	Signing functionality	21
2.3.1	General API design	21
2.3.2	A simple example	21
2.3.3	Timestamp handling	23
2.3.4	Extending <code>Signer</code>	24
2.4	Validation functionality	24
2.4.1	General API design	24
2.4.2	Accessing signatures in a document	25
2.4.3	Validating a PDF signature	25
2.4.4	Long-term verifiability checking	25
2.4.5	Probing different aspects of the validity of a signature	27
2.5	The <code>pdf-utils</code> package	27
2.5.1	Background and future perspectives	27
2.5.2	PDF object model	28
2.5.3	PDF content abstractions	28

3	API reference	29
3.1	pyhanko package	29
3.1.1	Subpackages	29
3.1.2	Submodules	82
3.1.3	Module contents	87
4	Known issues	89
5	Licenses	91
5.1	pyHanko License	91
5.2	Original PyPDF2 license	91
6	Indices and tables	93
	Python Module Index	95
	Index	97

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*.

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 devices¹.

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 made².

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

¹ The PKCS#11 functionality is currently only exposed in the CLI for Belgian eID cards, but it should be reasonably easy to write an implementation that works for any (RSA-based) PKCS#11 device. That being said, my experience with general PKCS#11 is limited, and I'm not 100% sure about current best practices for generic PKCS#11 clients (key selection, key-certificate pairing, ...). Additionally, the PKCS#11 module doesn't support ECC-based certificates yet, because I currently don't have the means to test those. The newest generation of BelD cards (Applet v1.8, launched in 2020) will therefore probably take a while to be integrated into the CLI. Discussion and (properly motivated) pull requests are certainly welcome!

² 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.

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 are “legitimate” changes. What precisely constitutes a “legitimate” change depends on the signature’s modification policy, but is not rigorously defined in the standard³. 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: 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](#).

³ The author has it on good authority that a rigorous 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.

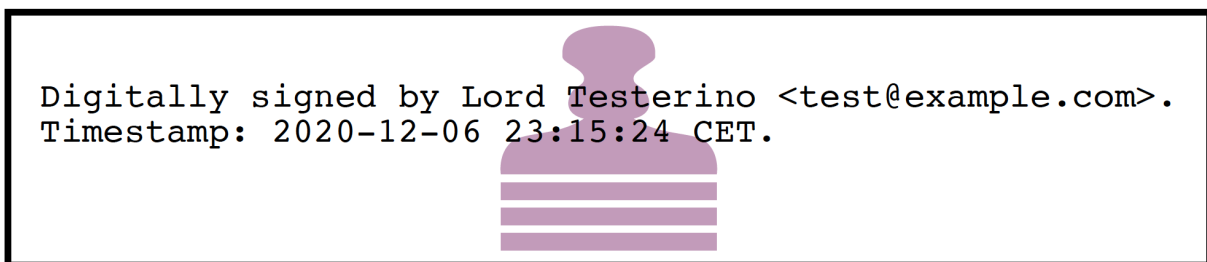


Fig. 1.1: The default appearance of a (visible) signature in pyHanko.

Signing a PDF file using a Belgian eID card

PyHanko also supports PKCS#11 devices to create signatures, although the CLI only exposes a wrapper for Belgian eID cards¹. This should work on all platforms supported by the eID middleware (tested on Linux, macOS and Windows).

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. 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 trust⁴ to use the ‘Signature’ certificate!

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 card⁵. 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;

⁴ 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.

⁵ The certificate’s serial number is in fact equal to the holder’s national registry number.

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 (yet) supported in pyHanko, and is also not guaranteed to be acceptable in all X.509 validation models.
 - 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'.

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.

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 will reissue their roots with the same key pair and a later expiration date⁶. Due to the way PKIX validation works, these certificates will typically work as drop-in replacements for the older ones.

⁶ Author’s note: it’s not clear to me whether this is good PKI practice, and how common it is.

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.

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.

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.

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:
    certvalidator:
      level: DEBUG
      output: certvalidator.log
    pyhanko.sign:
      level: DEBUG
```


The keys `root-level` and `root-ouput` 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 pender --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
```

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 either specify a path to a bitmap image, or 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). Any bitmap 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 text¹. If not specified, pyHanko will use a standard monospaced Courier font. See [TextBoxStyle](#) in the API reference for other customisable parameters.

¹ Custom font use is somewhat experimental, so please file an issue if you encounter problems. An appropriate subset of the font will always be embedded into the output file by pyHanko. The text rendering is currently fairly basic: pyHanko only takes character width into account, but ignores things like kerning pairs and ligatures. In particular, rendering of complex scripts (Myanmar, Indic scripts, ...) is not supported (but may be in the future). CJK fonts should work fine, though.

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](#).

(Under construction)

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

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.

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.

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 oscrypto import keys

franchising_ca = keys.parse_certificate(b'<certificate data goes here>')
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. While the signer *should* follow these restrictions, this doesn't always happen¹, so they shouldn't be relied upon if the signing of the document is out of your control.

¹ Right now, pyHanko doesn't yet reject such signatures, but it may in the future.

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.

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

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'
```

(continues on next page)

(continued from previous page)

```

)

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

    # do stuff with 'out'
    # ...

```

The `sign_pdf()` 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.

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*.

2.3.3 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` (which is actually a superclass of `PdfSigner`).

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
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'
)

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 size³. 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.

³ 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.

2.3.4 Extending Signer

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. 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 pyHanko¹, it should be sufficient to implement *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 matter².

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

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.

¹ ... and doesn’t require any cryptographic parameters. Signature mechanisms with parameters (such as RSA-PSS) are a bit more tricky. RSA-PSS is on the roadmap for pyHanko; once that has been implemented, using custom signature mechanisms with parameters should also become a bit more straightforward.

² 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.

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 `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 oscrypto import keys
from certvalidator import ValidationContext
from pyhanko.pdf_utils.reader import PdfFileReader
from pyhanko.sign.validation import validate_pdf_signature

root_cert = keys.parse_certificate(b'<certificate data goes here>')
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 oscrypto import keys
from pyhanko.pdf_utils.reader import PdfFileReader
from pyhanko.sign.validation import (
    validate_pdf_ltv_signature, RevocationInfoValidationType
)

root_cert = keys.parse_certificate(b'<certificate data goes here>')

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.

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.

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

2.4.5 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 two 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`.

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

Submodules

pyhanko.pdf_utils.barcodes module

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

Bases: *pyhanko.pdf_utils.content.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.

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 `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.content module

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

Bases: `enum.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() → *pyhanko.pdf_utils.generic.DictionaryObject*

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

class `pyhanko.pdf_utils.content.PdfContent` (*resources:* *Optional*[*pyhanko.pdf_utils.content.PdfResources*]
= *None*, *box:* *Optional*[*pyhanko.pdf_utils.layout.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:* *pyhanko.pdf_utils.content.ResourceType*, *name:* *pyhanko.pdf_utils.generic.NameObject*, *value:* *pyhanko.pdf_utils.generic.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*: `pyhanko.pdf_utils.content.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

Returns The `PdfResources` instance associated with this content fragment.

render () → bytes

Compile the content to graphics operators.

as_form_xobject () → `pyhanko.pdf_utils.generic.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[pyhanko.pdf_utils.content.PdfResources]
                                           = None, box: Optional[pyhanko.pdf_utils.layout.BoxConstraints]
                                           = None)
```

Bases: `pyhanko.pdf_utils.content.PdfContent`

Raw byte sequence to be used as PDF content.

render () → bytes

Compile the content to graphics operators.

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 `/Crypt` and `/LZWDecode` are missing.

class `pyhanko.pdf_utils.filters.Decoder`

Bases: `object`

General filter/decoder interface.

classmethod **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.

classmethod 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: `pyhanko.pdf_utils.filters.Decoder`

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

classmethod 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.

classmethod 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.ASCIIHexDecode`

Bases: `pyhanko.pdf_utils.filters.Decoder`

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

classmethod 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.

classmethod 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: `pyhanko.pdf_utils.filters.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.

classmethod `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.

classmethod `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.DECODERS = {'/A85': <class 'pyhanko.pdf_utils.filters.ASCII85Decode'>}`
Dictionary mapping decoder names to implementations.

pyhanko.pdf_utils.font module

Basic support for font handling & subsetting.

This module relies on `fontTools` for OTF parsing and subsetting.

Warning: If/when support is added for more advanced typographical features, the general `FontEngine` interface might change.

class `pyhanko.pdf_utils.font.FontEngine`

Bases: `object`

General interface for glyph lookups and font metrics.

measure (*txt: str*) → float

Measure the length of a string in em units.

Parameters **txt** – String to measure.

Returns A length in em units.

render (*txt: str*)

Render a string to a format suitable for inclusion in a content stream.

Parameters *txt* – String to render.

Returns A string.

as_resource () → *pyhanko.pdf_utils.generic.DictionaryObject*

Convert a *FontEngine* to a PDF object suitable for embedding inside a resource dictionary.

Returns A PDF dictionary.

class *pyhanko.pdf_utils.font.SimpleFontEngine* (*name, avg_width*)

Bases: *pyhanko.pdf_utils.font.FontEngine*

Simplistic font engine that only works with PDF standard fonts, and does not care about font metrics. Best used with monospaced fonts such as Courier.

static **default_engine** ()

Returns A *FontEngine* instance representing the Courier standard font.

render (*txt*)

Render a string to a format suitable for inclusion in a content stream.

Parameters *txt* – String to render.

Returns A string.

measure (*txt*)

Measure the length of a string in em units.

Parameters *txt* – String to measure.

Returns A length in em units.

as_resource ()

Convert a *FontEngine* to a PDF object suitable for embedding inside a resource dictionary.

Returns A PDF dictionary.

class *pyhanko.pdf_utils.font.GlyphAccumulator* (*tt: fontTools.ttLib.ttFont.TTFont*)

Bases: *pyhanko.pdf_utils.font.FontEngine*

Utility to collect & measure glyphs from TrueType fonts.

Warning: This utility class ignores all positioning & substitution information in the font file, other than glyph width/height. In particular, features such as kerning, ligatures, complex script support and regional substitution will not work out of the box.

Warning: This functionality was only really tested with CID-keyed fonts that have a CFF table. This is good enough to offer basic support for CJK scripts, but as I am not an OTF expert, more testing is necessary.

feed_string (*txt*)

Feed a string to this glyph accumulator.

Parameters *txt* – String to encode/measure. The glyphs used to render the string are marked for inclusion in the font subset associated with this glyph accumulator.

Returns Returns the CID-encoded version of the string passed in, and an estimate of the width in em units. The width computation ignores kerning, but takes the width of all characters into account.

render (*txt*)

Render a string to a format suitable for inclusion in a content stream.

Parameters *txt* – String to render.

Returns A string.

measure (*txt*)

Measure the length of a string in em units.

Parameters *txt* – String to measure.

Returns A length in em units.

embed_subset (*writer*: `pyhanko.pdf_utils.writer.BasePdfFileWriter`, *obj_stream*=None)

Embed a subset of this glyph accumulator's font into the provided PDF writer. 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.

Parameters

- **writer** – A PDF writer.
- **obj_stream** – If provided, write all relevant objects to the provided *obj_stream*. If None (the default), they will simply be written to the file as top-level objects.

Returns A reference to the embedded `/Font` object.

as_resource ()

Convert a `FontEngine` to a PDF object suitable for embedding inside a resource dictionary.

Returns A PDF dictionary.

class `pyhanko.pdf_utils.font.GlyphAccumulatorFactory` (*font_file*: str)

Bases: `object`

Stateless callable helper class to instantiate `GlyphAccumulator` objects.

font_file: str

Path to the OTF/TTF font to load.

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() → [pyhanko.pdf_utils.generic.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: [pyhanko.pdf_utils.generic.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() → [pyhanko.pdf_utils.generic.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: [pyhanko.pdf_utils.generic.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](#).

get_object() → *pyhanko.pdf_utils.generic.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: *pyhanko.pdf_utils.generic.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() → *pyhanko.pdf_utils.generic.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, encryption_key)
Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

class *pyhanko.pdf_utils.generic.IndirectObject* (*idnum, generation, pdf*)
Bases: *pyhanko.pdf_utils.generic.PdfObject*, *pyhanko.pdf_utils.generic.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

Returns the object ID of this reference.

property generation

Returns the generation number of this reference.

write_to_stream(stream, encryption_key)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

static read_from_stream(stream, container_ref: pyhanko.pdf_utils.generic.Dereferenceable)

class pyhanko.pdf_utils.generic.NullObject

Bases: *pyhanko.pdf_utils.generic.PdfObject*

PDF *null* object.

All instances are treated as equal and falsy.

write_to_stream(stream, encryption_key)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

static read_from_stream(stream)

class pyhanko.pdf_utils.generic.BooleanObject(value)

Bases: *pyhanko.pdf_utils.generic.PdfObject*

PDF boolean value.

write_to_stream(stream, encryption_key)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

static read_from_stream(stream)

class pyhanko.pdf_utils.generic.FloatObject(value='0', context=None)

Bases: *decimal.Decimal*, *pyhanko.pdf_utils.generic.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*, *encryption_key*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

class pyhanko.pdf_utils.generic.**NumberObject**(*value*)

Bases: int, *pyhanko.pdf_utils.generic.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*, *encryption_key*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

static read_from_stream(*stream*)

class pyhanko.pdf_utils.generic.**ByteStringObject**

Bases: bytes, *pyhanko.pdf_utils.generic.PdfObject*

PDF bytestring class.

property original_bytes

For compatibility with *TextStringObject.original_bytes*

write_to_stream(*stream*, *encryption_key*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

class pyhanko.pdf_utils.generic.**TextStringObject**

Bases: str, *pyhanko.pdf_utils.generic.PdfObject*

PDF text string object.

autodetect_pdfdocencoding = **False**

If True, this string was determined to be encoded in PDFDoc encoding.

autodetect_utf16 = **False**

If True, this string was determined to be encoded in UTF16-BE encoding.

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*, *encryption_key*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

class pyhanko.pdf_utils.generic.**NameObject**

Bases: `str`, `pyhanko.pdf_utils.generic.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+| [\\(\\) <>\\[\\] {}/%] ')`

write_to_stream (*stream*, *encryption_key*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

static read_from_stream (*stream*, *strict=True*)

class pyhanko.pdf_utils.generic.**ArrayObject** (*iterable=()*, */*)

Bases: `list`, `pyhanko.pdf_utils.generic.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=True*)

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** – If `False`, instances of `DecryptedObjectProxy` will be returned as-is. If `True`, they will be decrypted. Default `True`.

Returns A `PdfObject`.

write_to_stream (*stream*, *encryption_key*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

```
static read_from_stream(stream, container_ref)
```

```
class pyhanko.pdf_utils.generic.DictionaryObject(dict_data=None)
```

Bases: dict, [pyhanko.pdf_utils.generic.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=True)
```

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** – If False, instances of [DecryptedObjectProxy](#) will be returned as-is. If True, they will be decrypted. Default True.

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.

```
write_to_stream(stream, encryption_key)
```

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

```
static read_from_stream(stream, container_ref: pyhanko.pdf\_utils.generic.Dereferenceable)
```

```
class pyhanko.pdf_utils.generic.StreamObject(dict_data=None, stream_data=None, encoded_data=None)
```

Bases: [pyhanko.pdf_utils.generic.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 that 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.

```
strip_filters()
```

Ensure the stream is decoded, and remove any filters.

property data

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

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.

write_to_stream (*stream*, *encryption_key*)

Abstract method to render this object to an output stream.

Parameters

- **stream** – An output stream.
- **encryption_key** – Key to encrypt the object with.

```
pyhanko.pdf_utils.generic.read_object (stream, container_ref:
                                         hanko.pdf_utils.generic.Dereferenceable) → py-
                                         hanko.pdf_utils.generic.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.

Returns A *PdfObject*.

```
pyhanko.pdf_utils.generic.pdf_name
```

alias of `pyhanko.pdf_utils.generic.NameObject`

```
pyhanko.pdf_utils.generic.pdf_string (string) → Union[pyhanko.pdf_utils.generic.ByteStringObject,
                                                         pyhanko.pdf_utils.generic.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.datetime) → py-  
                                     hanko.pdf_utils.generic.TextStringObject
```

Convert a datetime object into a PDF string. This function supports both timezone-aware and naive datetimes.

Parameters **dt** – The datetime object to convert.

Returns A *TextStringObject* representing the datetime passed in.

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: PIL.Image.Image, writer: py-  
                                     hanko.pdf_utils.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[PIL.Image.Image, str], writer: Op-  
                                          tional[pyhanko.pdf_utils.writer.BasePdfFileWriter]  
                                          = None, resources: Op-  
                                          tional[pyhanko.pdf_utils.content.PdfResources]  
                                          = None, name: Optional[str] =  
                                          None, opacity=None, box: Op-  
                                          tional[pyhanko.pdf_utils.layout.BoxConstraints] =  
                                          None)
```

Bases: *pyhanko.pdf_utils.content.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

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*)
Bases: `pyhanko.pdf_utils.writer.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.

get_object (*ido*)

Retrieve the object associated with the provided reference from this PDF handler.

Parameters *ref* – An instance of `generic.Reference`.

Returns A PDF object.

mark_update (*obj_ref*: `Union[pyhanko.pdf_utils.generic.Reference, pyhanko.pdf_utils.generic.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*: `pyhanko.pdf_utils.generic.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 ()

write (*stream*)

Write the contents of this PDF writer to a stream.

Parameters *stream* – A writable output stream.

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 RC4-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.

Danger: One should also be aware that the 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.

Parameters `user_pwd` – The original file’s user password.

Raises `PdfReadError` – Raised when there is a problem decrypting the file.

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*: `pyhanko.pdf_utils.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.

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.BoxSpecificationError`

Bases: `ValueError`

Raised when a box constraint is over/underspecified.

class `pyhanko.pdf_utils.layout.BoxConstraints` (*width=None, height=None, aspect_ratio: Optional[fractions.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`

Returns The width of the box.

Raises `BoxSpecificationError` – if the box’s width could not be determined.

property `width_defined`

Returns `True` if the box currently has a well-defined width, `False` otherwise.

property `height`

Returns The height of the box.

Raises `BoxSpecificationError` – if the box’s height could not be determined.

property `height_defined`

Returns `True` if the box currently has a well-defined height, `False` otherwise.

property `aspect_ratio`

Returns The aspect ratio of the box.

Raises `BoxSpecificationError` – if the box’s aspect ratio could not be determined.

property `aspect_ratio_defined`

Returns `True` if the box currently has a well-defined aspect ratio, `False` otherwise.

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.

exception `pyhanko.pdf_utils.misc.PdfError`

Bases: `Exception`

exception `pyhanko.pdf_utils.misc.PdfReadError`

Bases: `pyhanko.pdf_utils.misc.PdfError`

exception `pyhanko.pdf_utils.misc.PdfWriteError`

Bases: `pyhanko.pdf_utils.misc.PdfError`

exception `pyhanko.pdf_utils.misc.PdfStreamError`

Bases: `pyhanko.pdf_utils.misc.PdfReadError`

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: `pyhanko.pdf_utils.rw_common.PdfHandler`

Class implementing functionality to read a PDF file and cache certain data about it.

last_startxref = `None`

has_xref_stream = `False`

property `root_ref`

Returns A reference to the document catalog of this PDF handler.

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`

Returns The total number of revisions made to this file.

get_object (*ref*, *revision=None*, *never_decrypt=False*, *transparent_decrypt=True*)

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.

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: bytes*) → int

When using an encrypted PDF file with the PDF legacy RC4-based 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.

Supplying either user or owner password will work.

Danger: One should also be aware that the 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.

Parameters password – The password to match.

Returns 0 if the password failed, 1 if the password matched the user password, and 2 if the password matched the owner password.

Raises NotImplementedError – Raised if the document uses an unsupported encryption method.

property encrypted

Returns True if a document is encrypted, False otherwise.

get_historical_resolver (*revision: int*) → *pyhanko.pdf_utils.reader.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 signatures embedded in this document, in signing order; see *EmbeddedPdfSignature*.

class *pyhanko.pdf_utils.reader.HistoricalResolver* (*reader: pyhanko.pdf_utils.reader.PdfFileReader, revision*)

Bases: *pyhanko.pdf_utils.rw_common.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.

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.

get_object (*ref: pyhanko.pdf_utils.generic.Reference*)

Retrieve the object associated with the provided reference from this PDF handler.

Parameters ref – An instance of *generic.Reference*.

Returns A PDF object.

property root_ref

Returns A reference to the document catalog of this PDF handler.

collect_dependencies (*obj*: `pyhanko.pdf_utils.generic.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.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 querying objects in PDF readers and writers alike.

get_object (*ref*: `pyhanko.pdf_utils.generic.Reference`)
Retrieve the object associated with the provided reference from this PDF handler.

Parameters **ref** – An instance of *generic.Reference*.

Returns A PDF object.

property `root_ref`

Returns A reference to the document catalog of this PDF handler.

property `root`

Returns The document catalog of this PDF handler.

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.

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.

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: Union[pyhanko.pdf_utils.font.FontEngine,
                                                Callable[[], pyhanko.pdf_utils.font.FontEngine]]
                                         = <factory>, font_size: int = 10, leading: Optional[int] = None)
```

Bases: `pyhanko.pdf_utils.config_utils.ConfigurableMixin`

Container for basic text styling settings.

font: Union[`pyhanko.pdf_utils.font.FontEngine`, `Callable[]`, `pyhanko.pdf_utils.font.Fon`
The `FontEngine` to be used for this text style. Defaults to Courier (as a non-embedded standard font).

Caution: Not all `FontEngine` implementations are reusable and/or stateless! When reusability is a requirement, passing a no-argument callable that produces `FontEngine` objects of the appropriate type might help (see `GlyphAccumulatorFactory`).

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: Union[pyhanko.pdf_utils.font.FontEngine,
                                                Callable[[], py-
                                                hanko.pdf_utils.font.FontEngine]] = <factory>,
                                         font_size: int = 10, leading: Optional[int] =
                                         None, text_sep: int = 10, border_width: int = 0,
                                         vertical_center: bool = True)
```

Bases: `pyhanko.pdf_utils.text.TextStyle`

Extension of `TextStyle` for use in text boxes.

text_sep: int = 10
Separation of text from the box's border, in user units.

border_width: int = 0
Border width, if applicable.

vertical_center: bool = True
Attempt to vertically center text if the box's height is fixed.

```
class pyhanko.pdf_utils.text.TextBox (style: pyhanko.pdf_utils.text.TextBoxStyle, resources:
                                         Optional[pyhanko.pdf_utils.content.PdfResources]
                                         = None, box: Optional[pyhanko.pdf_utils.layout.BoxConstraints]
                                         = None, writer=None, font_name='F1')
```

Bases: `pyhanko.pdf_utils.content.PdfContent`

Implementation of a text box that implements the `PdfContent` interface.

Note: Text boxes currently don't offer automatic word wrapping.

wrap_string (*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.

get_text_height ()

Returns The text height in user units.

text_x ()

Returns The x-position where the text will be painted.

text_y ()

Returns The y-position where the text will be painted.

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.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.

Warning: The usefulness of object streams is somewhat stymied by the fact that PDF stream objects cannot be embedded into object streams for syntactical reasons.

add_object (*idnum*: int, *obj*: pyhanko.pdf_utils.generic.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 () → pyhanko.pdf_utils.generic.StreamObject

Render the object stream to a PDF stream object

Returns An instance of *StreamObject*.

class pyhanko.pdf_utils.writer.BasePdfFileWriter (*root*, *info*, *document_id*,
obj_id_start=0, stream_xrefs=True)

Bases: pyhanko.pdf_utils.rw_common.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).

mark_update (*obj_ref*: Union[pyhanko.pdf_utils.generic.Reference, pyhanko.pdf_utils.generic.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*: pyhanko.pdf_utils.generic.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

Returns A reference to the document catalog of this PDF handler.

get_object (*ido*)

Retrieve the object associated with the provided reference from this PDF handler.

Parameters *ref* – An instance of *generic.Reference*.

Returns A PDF object.

allocate_placeholder() → *pyhanko.pdf_utils.generic.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*[*pyhanko.pdf_utils.writer.ObjectStream*] = *None*, *idnum*=*None*) → *pyhanko.pdf_utils.generic.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.

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*: *pyhanko.pdf_utils.generic.PdfObject*, *obj_stream*: *Optional*[*pyhanko.pdf_utils.writer.ObjectStream*] = *None*) → *Optional*[*pyhanko.pdf_utils.generic.PdfObject*]

Deep-copy an object into this writer, dealing with resolving indirect references in the process.

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*: [pyhanko.pdf_utils.rw_common.PdfHandler](#), *page_ix*=0, *content_stream*=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)
- **content_stream** – Index of the page’s content stream to copy, if multiple are present (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.

class [pyhanko.pdf_utils.writer.PageObject](#) (*contents*, *media_box*, *resources*=None)

Bases: [pyhanko.pdf_utils.generic.DictionaryObject](#)

Subclass of [DictionaryObject](#) that handles some of the initialisation boilerplate for page objects.

class [pyhanko.pdf_utils.writer.PdfFileWriter](#) (*stream_xrefs*=True)

Bases: [pyhanko.pdf_utils.writer.BasePdfFileWriter](#)

Class to write new PDF files.

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**[[pyhanko.pdf_utils.writer.ObjectStream](#)]

[pyhanko.pdf_utils.writer.init_xobject_dictionary](#) (*command_stream*: bytes, *box_width*, *box_height*, *resources*: *Optional*[[pyhanko.pdf_utils.generic.DictionaryObject](#)] = None) → [pyhanko.pdf_utils.generic.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.

Module contents

pyhanko.sign package

Submodules

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) → pkcs11.types.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: pkcs11.types.Session, cert_label: str, ca_chain=None, key_label=None)`
Bases: `pyhanko.sign.pkcs11.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.

signature_mechanism: `asn1crypto.algos.SignedDigestAlgorithm`
The (cryptographic) signature mechanism to use.

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)`

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).

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.

seed_value_dict: `pyhanko.sign.fields.SigSeedValueSpec = None`
Specification for the seed value dictionary, if applicable.

field_mdp_spec: `pyhanko.sign.fields.FieldMDPSpec = None`
Specification for the field lock dictionary, if applicable.

doc_mdp_update_value: `pyhanko.sign.fields.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.

format_lock_dictionary() → `Optional[pyhanko.pdf_utils.generic.DictionaryObject]`

class `pyhanko.sign.fields.SigSeedValFlags` (*value*)
Bases: `enum.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: While this enum records values for all flags, not all corresponding constraint types have been implemented yet.

FILTER = 1

SUBFILTER = 2

V = 4

REASONS = 8

LEGAL_ATTESTATION = 16

ADD_REV_INFO = 32

DIGEST_METHOD = 64

LOCK_DOCUMENT = 128

APPEARANCE_FILTER = 256

UNSUPPORTED = 400

Flags for which the corresponding constraint is unsupported.

```
class pyhanko.sign.fields.SigCertConstraints (flags:                                py-
                                             hanko.sign.fields.SigCertConstraintFlags
                                             =                                <SigCertConstraint-
                                             Flags.0: 0>, subjects: Optional[List[asn1crypto.x509.Certificate]]
                                             = None, subject_dn: Optional[asn1crypto.x509.Name]
                                             = None, issuers: Optional[List[asn1crypto.x509.Certificate]]
                                             = None, info_url: Optional[str] = None, url_type: py-
                                             hanko.pdf_utils.generic.NameObject =
                                             '/Browser')
```

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: `pyhanko.sign.fields.SigCertConstraintFlags = 0`

Enforcement flags. By default, all entries are optional.

subjects: `List[asn1crypto.x509.Certificate] = None`

Explicit list of certificates that can be used to sign a signature field.

subject_dn: `asn1crypto.x509.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[asn1crypto.x509.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 signature cannot be found.

url_type: `pyhanko.pdf_utils.generic.NameObject = '/Browser'`

Handler that should be used to open `info_url`. `/Browser` is the only implementation-independent value.

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:* `asn1crypto.x509.Certificate`, *validation_path:* `Optional[certvalidator.path.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.0:    0>, reasons:
                                             Optional[List[str]]    =    None, times-
                                             tamp_server_url:    Optional[str]    =    None,
                                             timestamp_required: bool = False, cert: Op-
                                             tional[pyhanko.sign.fields.SigCertConstraints]
                                             =    None, subfilters:    Op-
                                             tional[List[pyhanko.sign.fields.SigSeedSubFilter]]
                                             =    None, digest_methods: Optional[List[str]]
                                             =    None, add_rev_info:    Optional[bool]    =
                                             None)
```

Bases: object

Python representation of a PDF seed value dictionary.

flags: *pyhanko.sign.fields.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: *pyhanko.sign.fields.SigCertConstraints* = None
Constraints on the signer's certificate.

subfilters: List[*pyhanko.sign.fields.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.

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: `enum.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

ISSUER = 2

OID = 4

SUBJECT_DN = 8

RESERVED = 16

KEY_USAGE = 32

URL = 64

class `pyhanko.sign.fields.SigSeedSubFilter`(*value*)

Bases: `enum.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.MDPPerm`(*value*)

Bases: `pyhanko.pdf_utils.misc.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.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: pyhanko.sign.fields.FieldMDPAction, fields:
                                         Optional[List[str]] = None)
    Bases: object

    /FieldMDP policy description.

    This class models both field lock dictionaries and /FieldMDP transformation parameters.

    action: pyhanko.sign.fields.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() → pyhanko.pdf_utils.generic.DictionaryObject
        Render this /FieldMDP policy description as a PDF dictionary.

        Returns A DictionaryObject.

    as_transform_params() → pyhanko.pdf_utils.generic.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() → pyhanko.pdf_utils.generic.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) → pyhanko.sign.fields.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, include_on_page,
*, writer, sig_object_ref=None,
box=None, appearances: Optional[pyhanko.stamp.AnnotAppearances]
= None)
```

Bases: `pyhanko.pdf_utils.generic.DictionaryObject`

```
pyhanko.sign.fields.enumerate_sig_fields (handler: pyhanko.pdf_utils.rw_common.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: py-
hanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter,
sig_field_spec: py-
hanko.sign.fields.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.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`.

```
class pyhanko.sign.general.SignatureStatus (intact: bool, valid: bool, trusted:
bool, revoked: bool, signing_cert:
asn1crypto.x509.Certificate,
pkcs7_signature_mechanism: str,
md_algorithm: str, validation_path: cert-
validator.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.

trusted: bool

Reports whether the signer's certificate is trusted w.r.t. the currently relevant validation context and key usage requirements.

revoked: `bool`

Reports whether the signer's certificate has been revoked or not. If this field is `True`, then obviously `trusted` will be `False`.

signing_cert: `asn1crypto.x509.Certificate`

Contains the certificate of the signer, as embedded in the CMS object.

pkcs7_signature_mechanism: `str`

PKCS7 signature mechanism used.

md_algorithm: `str`

Message digest algorithm used.

validation_path: `certvalidator.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 usage extensions are required to be present on the signer's certificate.

extd_key_usage: `ClassVar[Set[str]] = {}`

Class property indicating which extended key usage extensions are required to be present on the signer's certificate.

summary_fields()

summary()

Provide a textual but machine-parsable summary of the validity.

classmethod validate_cert_usage (*validator: certvalidator.CertificateValidator*)

`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.

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 `KeyError` – Raised when no such type entry could be found in the `cms.CMSAttributes` object.

class `pyhanko.sign.general.CertificateStore`

Bases: `object`

Bare-bones interface for modelling a collection of certificates.

register (*cert: asn1crypto.x509.Certificate*)

Add a certificate to the collection.

Parameters **cert** – The certificate to add.

register_multiple (*certs*)
Register multiple certificates.

Parameters **certs** – Certificates to register.

class pyhanko.sign.general.**SimpleCertificateStore**
Bases: *pyhanko.sign.general.CertificateStore*

Unopinionated replacement for certvalidator’s CertificateRegistry in cases where we explicitly don’t care about whether the certs are trusted or not.

register (*cert: asn1crypto.x509.Certificate*)
Add a certificate to the collection.

Parameters **cert** – The certificate to add.

exception pyhanko.sign.general.**SigningError**
Bases: *ValueError*

Error encountered while signing a file.

exception pyhanko.sign.general.**UnacceptableSignerError**
Bases: *pyhanko.sign.general.SigningError*

Error raised when a signer was judged unacceptable.

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: pkcs11.types.Session, cert_label: str, ca_chain=None, key_label=None*)
Bases: *pyhanko.sign.signers.Signer*

Signer implementation for PKCS11 devices.

Note: this class only supports the “RSA with PKCS#1 v1.5” scheme. In particular, there’s no ECDSA support (yet).

signature_mechanism: *asn1crypto.algos.SignedDigestAlgorithm*
The (cryptographic) signature mechanism to use.

property **cert_registry**

property **signing_cert**

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.

pyhanko.sign.signers module

```
class pyhanko.sign.signers.PdfSignatureMetadata (field_name: Optional[str] = None,
                                                md_algorithm: Optional[str] =
None, location: Optional[str] =
None, reason: Optional[str] =
None, name: Optional[str] = None,
certify: bool = False, subfilter: Op-
tional[pyhanko.sign.fields.SigSeedSubFilter]
= None, embed_validation_info: bool
= False, use_pades_lta: bool = False,
timestamp_field_name: Optional[str]
= None, validation_context: Op-
tional[certvalidator.context.ValidationContext]
= None, docmdp_permissions: py-
hanko.sign.fields.MDPPerm = <MDP-
Perm.FILL_FORMS: 2>)
```

Bases: object

Specification for a PDF signature.

field_name: 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: str = None

The name of the digest algorithm to use. It should be supported by `hashlib`.

If `None`, this will ordinarily default to the value of `DEFAULT_MD`, unless a seed value dictionary and/or a prior certification signature happen to be available.

location: str = None

Location of signing.

reason: str = None

Reason for signing (textual).

name: 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: *pyhanko.sign.fields.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: 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: certvalidator.context.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: pyhanko.sign.fields.MDPPerm = 2`

Indicates the document modification policy that will be in force after this signature is created.

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.

`class pyhanko.sign.signers.Signer`

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`).

`signing_cert: asn1crypto.x509.Certificate`

The certificate that will be used to create the signature.

`cert_registry: pyhanko.sign.general.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: asn1crypto.algos.SignedDigestAlgorithm`

The (cryptographic) signature mechanism to use.

`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.

property subject_name

Returns The subject's common name as a string, extracted from *signing_cert*.

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.

Returns A CMS attribute containing the relevant data.

signed_attrs (*data_digest*: *bytes*, *timestamp*: *Optional[datetime.datetime] = None*, *revocation_info*=*None*, *use_pades*=*False*)
Format the signed attributes for a CMS signature.

Parameters

- **data_digest** – Raw digest of the data to be signed.
- **timestamp** – Current timestamp (ignored when *use_pades* is *True*).
- **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.

Returns An `asn1crypto.cms.CMSAttributes` object.

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.

sign (*data_digest*: *bytes*, *digest_algorithm*: *str*, *timestamp*: *Optional[datetime.datetime] = None*, *dry_run*=*False*, *revocation_info*=*None*, *use_pades*=*False*, *timestampper*=*None*) → `asn1crypto.cms.ContentInfo`
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** – Current timestamp (ignored when `use_pades` is `True`).
- **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.

Returns An `ContentInfo` object.

```
class pyhanko.sign.signers.SimpleSigner (signing_cert:      asn1crypto.x509.Certificate,
                                          signing_key:      asn1crypto.keys.PrivateKeyInfo,
                                          cert_registry:    pyhanko.sign.general.CertificateStore,
                                          signature_mechanism: Optional[asn1crypto.algos.SignedDigestAlgorithm]
                                          = None)
```

Bases: `pyhanko.sign.signers.Signer`

Simple signer implementation where the key material is available in local memory.

signing_key: `asn1crypto.keys.PrivateKeyInfo`
Private key associated with the certificate in `signing_cert`.

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.

```
classmethod load_pkcs12 (pfx_file,      ca_chain_files=None,      passphrase=None,      signature_mechanism=None)
```

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.
- **passphrase** – Passphrase to decrypt the PKCS#12 archive, if required.
- **signature_mechanism** – Override the signature mechanism to use.

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)
```

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.

Returns A *SimpleSigner* object initialised with key material loaded from the files provided.

```
class pyhanko.sign.signers.PdfTimeStamper(timestamper: pyhanko.sign.timestamps.TimeStamper)
```

Bases: object

Class to encapsulate the process of appending document timestamps to PDF files.

```
generate_timestamp_field_name() → str
```

Generate a unique name for a document timestamp field using uuid.

Returns The field name, as a (Python) string.

```
timestamp_pdf(pdf_out: pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter,
               md_algorithm, validation_context, bytes_reserved=None, validation_paths=None,
               in_place=False, timestamper: Optional[pyhanko.sign.timestamps.TimeStamper] =
               None)
```

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 `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.
- **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.
- **in_place** – Sign the input in-place. If `False`, write output to a `BytesIO` object.

- **timestamp** – Override the default *TimeStamper* associated with this *PdfTimeStamper*.

Returns The output stream containing the signed output.

update_archival_timestamp_chain (*reader*: `pyhanko.pdf_utils.reader.PdfFileReader`, *validation_context*, *in_place*=*True*)

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** – `certvalidator.ValidationContext` object to validate the last timestamp.
- **in_place** – Sign the input in-place. If *False*, write output to a `BytesIO` object.

Returns The output stream containing the signed output.

```
class pyhanko.sign.signers.PdfSigner (signature_meta: py-
                                     hanko.sign.signers.PdfSignatureMetadata, signer:
                                     pyhanko.sign.signers.Signer, *, timestamp: Op-
                                     tional[pyhanko.sign.timestamps.TimeStamper]
                                     = None, stamp_style: Op-
                                     tional[pyhanko.stamp.TextStampStyle]
                                     = None, new_field_spec: Op-
                                     tional[pyhanko.sign.fields.SigFieldSpec] = None)
```

Bases: `pyhanko.sign.signers.PdfTimeStamper`

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.
- **timestamp** – *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 `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*.

generate_timestamp_field_name () → str

Look up the timestamp field name in the *PdfSignatureMetadata* object associated with this *PdfSigner*. If not specified, generate a unique field name using `uuid`.

Returns The field name, as a (Python) string.

sign_pdf (*pdf_out*: `pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter`, *existing_fields_only*=*False*, *bytes_reserved*=*None*, *in_place*=*False*, *appearance_text_params*=*None*)

Sign a PDF file using the provided output writer.

Parameters

- **pdf_out** – 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.
- **in_place** – Sign the input in-place. If `False`, write output to a `BytesIO` object.
- **appearance_text_params** – Dictionary with text parameters that will be passed to the signature appearance constructor (if applicable).

Returns The output stream containing the signed data.

```
pyhanko.sign.signers.sign_pdf(pdf_out: pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter,
                               signature_meta: pyhanko.sign.signers.PdfSignatureMetadata,
                               signer: pyhanko.sign.signers.Signer, timestamp: Optional[pyhanko.sign.timestamps.TimeStamper] = None,
                               new_field_spec: Optional[pyhanko.sign.fields.SigFieldSpec] = None, existing_fields_only=False, bytes_reserved=None,
                               in_place=False)
```

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`.

Returns The output stream containing the signed output.

```
pyhanko.sign.signers.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.signers.DEFAULT_MD = 'sha256'
```

Default message digest algorithm used when computing digests for use in signatures.

```
pyhanko.sign.signers.DEFAULT_SIGNING_STAMP_STYLE = TextStampStyle(text_box_style=TextBoxStyle)
```

Default stamp style used for visible signatures.

pyhanko.sign.timestamps 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.TimestampSignatureStatus (intact: bool, valid: bool, trusted: bool, re-
voked: bool, signing_cert: asn1crypto.x509.Certificate,
pkcs7_signature_mechanism: str, md_algorithm: str,
validation_path: certvalidator.path.ValidationPath,
timestamp: datetime.datetime)
```

Bases: `pyhanko.sign.general.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[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.datetime`

Value of the timestamp token as a datetime object.

class pyhanko.sign.timestamps.TimeStamper

Bases: `object`

Class to make [RFC 3161](#) timestamp requests.

dummy_response (*md_algorithm*) → `asn1crypto.cms.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.

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 A generator producing validation paths.

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.

request_tsa_response (*req: `asn1crypto.tsp.TimeStampReq`*) → `asn1crypto.tsp.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.

timestamp (*message_digest, md_algorithm*) → `asn1crypto.cms.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.

class `pyhanko.sign.timestamps.HTTPTimeStamper` (*url, https=False, timeout=5, auth=None, headers=None*)

Bases: `pyhanko.sign.timestamps.TimeStamper`

Standard HTTP-based timestamp client.

request_headers () → dict
Format the HTTP request headers.

Returns Header dictionary.

timestamp (*message_digest, md_algorithm*) → `asn1crypto.cms.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.

request_tsa_response (*req: `asn1crypto.tsp.TimeStampReq`*) → `asn1crypto.tsp.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.

exception `pyhanko.sign.timestamps.TimestampRequestError`

Bases: `OSError`

Raised when an error occurs while requesting a timestamp.

pyhanko.sign.validation module

class `pyhanko.sign.validation.SignatureCoverageLevel` (*value*)

Bases: `pyhanko.pdf_utils.misc.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.ModificationLevel` (*value*)

Bases: `pyhanko.pdf_utils.misc.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 signature long term archival (LTA) updates. 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, 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.

```
class pyhanko.sign.validation.PdfSignatureStatus (intact: bool, valid: bool, trusted:
    bool, revoked: bool, signing_cert:
    asn1crypto.x509.Certificate,
    pkcs7_signature_mechanism: str,
    md_algorithm: str, validation_path:
    certvalidator.path.ValidationPath,
    coverage: py-
    hanko.sign.validation.SignatureCoverageLevel,
    modification_level: py-
    hanko.sign.validation.ModificationLevel,
    seed_value_ok: bool, docmdp_ok:
    bool, signer_reported_dt: Op-
    tional[datetime.datetime] =
    None, timestamp_validity: Op-
    tional[pyhanko.sign.timestamps.TimestampSignatureStatus]
    = None)
```

Bases: *pyhanko.sign.general.SignatureStatus*

Class to indicate the validation status of a PDF signature.

coverage: *pyhanko.sign.validation.SignatureCoverageLevel*

Indicates how much of the document is covered by the signature.

modification_level: *pyhanko.sign.validation.ModificationLevel*

Indicates the degree to which the document was modified after the signature was applied.

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*.

docmdp_ok: `bool`

Indicates whether the signature's *modification_level* is in line with the document signature policy in force.

signer_reported_dt: `Optional[datetime.datetime] = None`

Signer-reported signing time, if present in the signature.

Generally speaking, this timestamp should not be taken as fact.

timestamp_validity: `Optional[pyhanko.sign.timestamps.TimestampSignatureStatus] = None`

Validation status of the timestamp token embedded in this signature, if present.

property bottom_line

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.

summary_fields()

pretty_print_details()

class `pyhanko.sign.validation.EmbeddedPdfSignature` (*reader:* `pyhanko.pdf_utils.reader.PdfFileReader`,
sig_field: `pyhanko.pdf_utils.generic.DictionaryObject`)

Bases: `object`

Class modelling a signature embedded in a PDF document.

sig_object: `pyhanko.pdf_utils.generic.DictionaryObject`

The signature dictionary.

sig_field: `pyhanko.pdf_utils.generic.DictionaryObject`

The field dictionary of the form field containing the signature.

signed_data: `asn1crypto.cms.SignedData`

CMS signed data in the signature.

signer_cert: `asn1crypto.x509.Certificate`

Certificate of the signer.

property field_name

Returns Name of the signature field.

property self_reported_timestamp

Returns The signing time as reported by the signer, if embedded in the signature's signed attributes.

property external_timestamp_data

Returns The signed data component of the timestamp token embedded in this signature, if present.

compute_integrity_info (*skip_diff=False*)

Compute the various integrity indicators of this signature.

Parameters *skip_diff* – If `True`, skip the (rather expensive) modification level evaluation.

summarise_integrity_info () → dict

Compile the integrity information for this signature into a dictionary that can later be passed to *PdfSignatureStatus* as kwargs.

Returns A kwargs dictionary.

property docmdp_level

Returns

The document modification policy required by this signature.

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

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 () → *pyhanko.sign.validation.SignatureCoverageLevel*

Internal method used to evaluate the coverage level of a signature.

Returns The coverage level of the signature.

evaluate_modifications () → *pyhanko.sign.validation.ModificationLevel*

Internal method used to evaluate the modification level of a signature.

Returns The modification level of the signature.

class *pyhanko.sign.validation.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

class `pyhanko.sign.validation.RevocationInfoValidationType` (*value*)

Bases: `enum.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** ()

class `pyhanko.sign.validation.VRI` (*certs: set = <factory>, ocsp: 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.

ocsp: set
Relevant OCSP responses.

crls: set
Relevant CRLs.

as_pdf_object () → *pyhanko.pdf_utils.generic.DictionaryObject*

Returns A PDF dictionary representing this VRI entry.

class `pyhanko.sign.validation.DocumentSecurityStore` (*writer, certs=None, ocsp=None, crls=None, vri_entries=None, backing_pdf_object=None*)

Bases: `object`

Representation of a DSS in Python.

static **sig_content_identifier** (*contents*) → *pyhanko.pdf_utils.generic.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, paths, validation_context*)

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*)
- **paths** – Validation paths to add.
- **validation_context** – Validation context to source CRLs and OCSP responses from.

as_pdf_object ()

Convert the *DocumentSecurityStore* object to a python dictionary. This method also handles DSS updates.

Returns A PDF object representing this DSS.

as_validation_context (*validation_context_kwargs, include_revinfo=True*) → *certvalidator.context.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: pyhanko.pdf_utils.rw_common.PdfHandler*) → *pyhanko.sign.validation.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 add_dss (*output_stream, sig_contents, paths, validation_context*)

Add or update a DSS, and add the new information to a specific VRI. This will be done 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)
- **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.

pyhanko.sign.validation.apply_adobe_revocation_info (*signer_info: asn1crypto.cms.SignerInfo, validation_context_kwargs=None*) → *certvalidator.context.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.read_certification_data (reader: py-  
hanko.pdf_utils.reader.PdfFileReader)  
→ Optional[pyhanko.sign.validation.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.

```
pyhanko.sign.validation.validate_pdf_ltv_signature (embedded_sig: py-  
hanko.sign.validation.EmbeddedPdfSignature,  
validation_type: py-  
hanko.sign.validation.RevocationInfoValidationType,  
validation_context_kwargs=None, bootstrap_validation_context=None,  
force_revinfo=False) → py-  
hanko.sign.validation.PdfSignatureStatus
```

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 *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.

Returns The status of the signature.

```
pyhanko.sign.validation.validate_pdf_signature (embedded_sig: py-  
hanko.sign.validation.EmbeddedPdfSignature,  
signer_validation_context: Optional[certvalidator.context.ValidationContext]  
= None, ts_validation_context: Optional[certvalidator.context.ValidationContext]  
= None) → py-  
hanko.sign.validation.PdfSignatureStatus
```

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*).

Returns The status of the PDF signature in question.

```

pyhanko.sign.validation.validate_cms_signature(signed_data:
                                              asn1crypto.cms.SignedData,
                                              status_cls: Type[StatusType] = <class 'py-
hanko.sign.general.SignatureStatus'>,
                                              raw_digest: Optional[bytes] =
None,
                                              validation_context: Op-
tional[certvalidator.context.ValidationContext]
= None,
                                              status_kwargs: Optional[dict]
= None,
                                              externally_invalid=False)

```

Validate a detached 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.
- **`externally_invalid`** – If `True`, there is an external reason why the signature cannot be valid, but the remaining validation logic still has to be run.

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)

exception `pyhanko.sign.validation.ValidationInfoReadingError`

Bases: `ValueError`

Error reading validation info.

exception `pyhanko.sign.validation.SignatureValidationError`

Bases: `ValueError`

Error validating a signature.

exception `pyhanko.sign.validation.SigSeedValueValidationError`

Bases: `pyhanko.sign.validation.SignatureValidationError`

Error validating a signature’s seed value constraints.

exception `pyhanko.sign.validation.SuspiciousModification`

Bases: `ValueError`

Error indicating a suspicious modification

Module contents

3.1.2 Submodules

pyhanko.cli module

Warning: This is the reference page for the (as of now largely undocumented) `pyhanko.cli` module that backs pyHanko’s command-line interface. You probably don’t want/need to call this module’s functions from your own python code.

Chances are that you were looking for the [CLI user guide](#) or the [library user guide](#) instead.

pyhanko.config module

```
class pyhanko.config.StdLogOutput (value)
    Bases: enum.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[pyhanko.config.StdLogOutput, str]
        Name of the output file, or a standard one.

    static parse_output_spec (spec) → Union[pyhanko.config.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, log_config: Dict[Union[str,
NoneType], pyhanko.config.LogConfig])
    Bases: object

    validation_contexts: Dict[str, dict]

    stamp_styles: Dict[str, dict]

    default_validation_context: str

    default_stamp_style: str

    time_tolerance: datetime.timedelta

    log_config: Dict[Optional[str], pyhanko.config.LogConfig]

    get_validation_context (name=None, as_dict=False)

    get_stamp_style (name=None) → pyhanko.stamp.TextStampStyle

pyhanko.config.init_validation_context_kwargs (trust, trust_replace, other_certs,
time_tolerance=None)

pyhanko.config.parse_trust_config (trust_config, time_tolerance) → dict
```

```
pyhanko.config.parse_logging_config(log_config_spec) → Dict[Optional[str], py-
                                         hanko.config.LogConfig]
pyhanko.config.parse_cli_config(yaml_str)
```

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: pyhanko.pdf_utils.generic.IndirectObject,
                                         rollover: Optional[pyhanko.pdf_utils.generic.IndirectObject]
                                         = None, down: Optional[pyhanko.pdf_utils.generic.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 () → *pyhanko.pdf_utils.generic.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.TextStampStyle (text_box_style: pyhanko.pdf_utils.text.TextBoxStyle =
                                         TextBoxStyle(font=<pyhanko.pdf_utils.font.SimpleFontEngine
                                         object>, font_size=10, leading=None, text_sep=10, bor-
                                         der_width=0, vertical_center=True), border_width: int
                                         = 3, stamp_text: str = '%(ts)s', timestamp_format: str
                                         = '%Y-%m-%d %H:%M:%S %Z', background: Op-
                                         tional[pyhanko.pdf_utils.content.PdfContent] = None,
                                         background_opacity: float = 0.6)
```

Bases: *pyhanko.pdf_utils.config_utils.ConfigurableMixin*

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: *pyhanko.pdf_utils.text.TextBoxStyle* = *TextBoxStyle*(font=<pyhanko.pdf_...
 The text box style for the internal text box used.

border_width: `int = 3`

Border width in user units (for the stamp, not the text box).

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.

background: `pyhanko.pdf_utils.content.PdfContent = None`

`PdfContent` instance that will be used to render the stamp's background.

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*)

The implementation of `process_entries()` calls `TextBoxStyle.from_config()` to parse the `text_box_style` configuration entry, if present.

Then, it processes the background specified. 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`.

See `ConfigurableMixin.process_entries()` for general documentation about this method.

```
class pyhanko.stamp.QRStampStyle(text_box_style: pyhanko.pdf_utils.text.TextBoxStyle =
    TextBoxStyle(font=<pyhanko.pdf_utils.font.SimpleFontEngine
    object>, font_size=10, leading=None, text_sep=10, bor-
    der_width=0, vertical_center=True), border_width: int
    = 3, stamp_text: str = 'Digital version available at\nthis
    url: %(url)s\nTimestamp: %(ts)s', timestamp_format:
    str = '%Y-%m-%d %H:%M:%S %Z', background: Op-
    tional[pyhanko.pdf_utils.content.PdfContent] = None, back-
    ground_opacity: float = 0.6, innsep: int = 3, stamp_qrsize:
    float = 0.25)
```

Bases: `pyhanko.stamp.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.

stamp_qrsize: `float = 0.25`

Indicates the proportion of the width of the stamp that should be taken up by the QR code.

`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).


```
class pyhanko.stamp.TextStamp (writer: pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter,
                                style,          text_params=None,          box:          Op-
                                tional[pyhanko.pdf_utils.layout.BoxConstraints] = None)
Bases: pyhanko.pdf_utils.content.PdfContent
```

Class that renders a text stamp as specified by an instance of *TextStampStyle*.

extra_commands () → list

Render extra graphics commands to be used after painting the inner text box, but before drawing the border.

Returns A list of bytes objects.

get_stamp_width () → int

Compute the stamp's total width.

Returns The width of the stamp in user units.

get_stamp_height () → int

Compute the stamp's total height.

Returns The height of the stamp in user units.

text_box_x () → int

Text box x-coordinate.

Returns The horizontal position of the internal text box's lower left corner inside the stamp's bounding box.

text_box_y ()

Text box y-coordinate.

Returns The horizontal position of the internal text box's lower left corner inside the stamp's bounding box.

get_default_text_params ()

Compute values for the default string interpolation parameters to be applied to the template string 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.

render ()

Compile the content to graphics operators.

register () → *pyhanko.pdf_utils.generic.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 *TextStamp* 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 () → *pyhanko.stamp.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.QRStamp* (*writer: pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter*,
 url: str, style: pyhanko.stamp.QRStampStyle, text_params=None,
 box: Optional[pyhanko.pdf_utils.layout.BoxConstraints] = None)

Bases: *pyhanko.stamp.TextStamp*

qr_default_width = 30

Default value for the QR code's width in user units. This value is only used if the stamp's bounding box does not have a defined width, in which case the *QRStampStyle.stamp_qrsize* attribute is unusable.

You can safely override this attribute if you so desire.

property qr_size

Compute the effective size of the QR code.

Returns The size of the QR code in user units.

extra_commands ()

Render extra graphics commands to be used after painting the inner text box, but before drawing the border.

Returns A list of bytes objects.

text_box_x ()

Text box x-coordinate.

Returns The horizontal position of the internal text box's lower left corner inside the stamp's bounding box.

get_stamp_height ()

Compute the stamp's total height.

Returns The height of the stamp in user units.

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 *TextStamp* 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.

pyhanko.stamp.text_stamp_file (*input_name: str, output_name: str, style: pyhanko.stamp.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: py-
                             hanko.stamp.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.

3.1.3 Module contents

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.
- 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.
- For now, only signatures using the “RSA with PKCS#1 v1.5” mechanism and ECDSA are effectively supported. Expanding this to include the full gamut of what *oscrypto* supports is on the roadmap.
- There is currently no 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.
- PKCS#11 support (esp. in the CLI) is limited to Belgian RSA-based cards, because my experience with (generic) PKCS#11 is very limited. Discussion and pull requests are certainly welcome, though!

LICENSES

5.1 pyHanko License

MIT License

Copyright (c) 2020 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.

5.2 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>

All rights reserved.

Redistribution and use in source and binary forms, with or without

(continues on next page)

(continued from previous page)

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`, 87
- `pyhanko.cli`, 82
- `pyhanko.config`, 82
- `pyhanko.pdf_utils`, 56
 - `pyhanko.pdf_utils.barcodes`, 29
 - `pyhanko.pdf_utils.config_utils`, 30
 - `pyhanko.pdf_utils.content`, 30
 - `pyhanko.pdf_utils.filters`, 32
 - `pyhanko.pdf_utils.font`, 34
 - `pyhanko.pdf_utils.generic`, 36
 - `pyhanko.pdf_utils.images`, 44
 - `pyhanko.pdf_utils.incremental_writer`, 45
 - `pyhanko.pdf_utils.layout`, 47
 - `pyhanko.pdf_utils.misc`, 47
 - `pyhanko.pdf_utils.reader`, 48
 - `pyhanko.pdf_utils.rw_common`, 50
 - `pyhanko.pdf_utils.text`, 51
 - `pyhanko.pdf_utils.writer`, 52
- `pyhanko.sign`, 82
 - `pyhanko.sign.beid`, 56
 - `pyhanko.sign.fields`, 56
 - `pyhanko.sign.general`, 62
 - `pyhanko.sign.pkcs11`, 64
 - `pyhanko.sign.signers`, 65
 - `pyhanko.sign.timestamps`, 72
 - `pyhanko.sign.validation`, 74
- `pyhanko.stamp`, 83

INDEX

A

- `action` (*pyhanko.sign.fields.FieldMDPSpec* attribute), 61
- `add_content_to_page()` (*pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter* method), 46
- `add_dss()` (*pyhanko.sign.validation.DocumentSecurityStore* class method), 79
- `add_object()` (*pyhanko.pdf_utils.writer.BasePdfFileWriter* method), 54
- `add_object()` (*pyhanko.pdf_utils.writer.ObjectStream* method), 53
- `ADD_REV_INFO` (*pyhanko.sign.fields.SigSeedValFlags* attribute), 57
- `add_rev_info` (*pyhanko.sign.fields.SigSeedValueSpec* attribute), 59
- `add_stream_to_page()` (*pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter* method), 46
- `ADOBE_PKCS7_DETACHED` (*pyhanko.sign.fields.SigSeedSubFilter* attribute), 60
- `ADOBE_STYLE` (*pyhanko.sign.validation.RevocationInfoValidationType* attribute), 78
- `ALL` (*pyhanko.sign.fields.FieldMDPAction* attribute), 61
- `allocate_placeholder()` (*pyhanko.pdf_utils.writer.BasePdfFileWriter* method), 53
- `AnnotAppearances` (class in *pyhanko.stamp*), 83
- `ANNOTATE` (*pyhanko.sign.fields.MDPPerm* attribute), 60
- `ANNOTATIONS` (*pyhanko.sign.validation.ModificationLevel* attribute), 75
- `APPEARANCE_FILTER` (*pyhanko.sign.fields.SigSeedValFlags* attribute), 57
- `append_signature_field()` (in module *pyhanko.sign.fields*), 62
- `apply()` (*pyhanko.stamp.QRStamp* method), 86
- `apply()` (*pyhanko.stamp.TextStamp* method), 85
- `apply_adobe_revocation_info()` (in module *pyhanko.sign.validation*), 79
- `apply_filter()` (*pyhanko.pdf_utils.generic.StreamObject* method), 43
- `ArrayObject` (class in *pyhanko.pdf_utils.generic*), 41
- `as_appearances()` (*pyhanko.stamp.TextStamp* method), 85
- `as_form_xobject()` (*pyhanko.pdf_utils.content.PdfContent* method), 32
- `as_numeric()` (*pyhanko.pdf_utils.generic.FloatObject* method), 39
- `as_numeric()` (*pyhanko.pdf_utils.generic.NumberObject* method), 40
- `as_pdf_object()` (*pyhanko.pdf_utils.content.PdfResources* method), 31
- `as_pdf_object()` (*pyhanko.pdf_utils.writer.ObjectStream* method), 53
- `as_pdf_object()` (*pyhanko.sign.fields.FieldMDPSpec* method), 61
- `as_pdf_object()` (*pyhanko.sign.fields.SigCertConstraints* method), 58
- `as_pdf_object()` (*pyhanko.sign.fields.SigSeedValueSpec* method), 59
- `as_pdf_object()` (*pyhanko.sign.validation.DocumentSecurityStore* method), 79
- `as_pdf_object()` (*pyhanko.sign.validation.VRI* method), 78
- `as_pdf_object()` (*pyhanko.stamp.AnnotAppearances* method), 83
- `as_resource()` (*pyhanko.pdf_utils.font.FontEngine* method), 35
- `as_resource()` (*pyhanko.pdf_utils.font.GlyphAccumulator* method), 36
- `as_resource()` (*pyhanko.pdf_utils.font.SimpleFontEngine* method), 36

method), 35
 as_sig_field_lock() (*pyhanko.sign.fields.FieldMDPSpec method*), 61
 as_transform_params() (*pyhanko.sign.fields.FieldMDPSpec method*), 61
 as_tuple() (*pyhanko.sign.validation.RevocationInfoValidationType class method*), 78
 as_validation_context() (*pyhanko.sign.validation.DocumentSecurityStore method*), 79
 ASCII85Decode (*class in pyhanko.pdf_utils.filters*), 33
 ASCIIHexDecode (*class in pyhanko.pdf_utils.filters*), 33
 aspect_ratio() (*pyhanko.pdf_utils.layout.BoxConstraints property*), 47
 aspect_ratio_defined() (*pyhanko.pdf_utils.layout.BoxConstraints property*), 47
 author_sig() (*pyhanko.sign.validation.DocMDPInfo property*), 77
 autodetect_pdfdocencoding (*pyhanko.pdf_utils.generic.TextStringObject attribute*), 40
 autodetect_utf16 (*pyhanko.pdf_utils.generic.TextStringObject attribute*), 40

B

background (*pyhanko.stamp.TextStampStyle attribute*), 84
 background_opacity (*pyhanko.stamp.TextStampStyle attribute*), 84
 BarcodeBox (*class in pyhanko.pdf_utils.barcodes*), 29
 BasePdfFileWriter (*class in pyhanko.pdf_utils.writer*), 53
 BEIDSigner (*class in pyhanko.sign.beid*), 56
 BooleanObject (*class in pyhanko.pdf_utils.generic*), 39
 border_width (*pyhanko.pdf_utils.text.TextBoxStyle attribute*), 51
 border_width (*pyhanko.stamp.TextStampStyle attribute*), 83
 bottom_line() (*pyhanko.sign.validation.PdfSignatureStatus property*), 76
 box (*pyhanko.sign.fields.SigFieldSpec attribute*), 57
 BoxConstraints (*class in pyhanko.pdf_utils.layout*), 47
 BoxSpecificationError, 47
 build_timestamper() (*pyhanko.sign.fields.SigSeedValueSpec method*), 59

C

ByteDot (*pyhanko.pdf_utils.generic.NumberObject attribute*), 40
 ByteStringObject (*class in pyhanko.pdf_utils.generic*), 40
 cache_get_indirect_object() (*pyhanko.pdf_utils.reader.PdfFileReader method*), 48
 cache_indirect_object() (*pyhanko.pdf_utils.reader.PdfFileReader method*), 48
 cert (*pyhanko.sign.fields.SigSeedValueSpec attribute*), 59
 cert_registry (*pyhanko.sign.signers.Signer attribute*), 66
 cert_registry() (*pyhanko.sign.pkcs11.PKCS11Signer property*), 64
 CertificateStore (*class in pyhanko.sign.general*), 63
 certify (*pyhanko.sign.signers.PdfSignatureMetadata attribute*), 65
 certs (*pyhanko.sign.validation.VRI attribute*), 78
 CLIConfig (*class in pyhanko.config*), 82
 collect_dependencies() (*pyhanko.pdf_utils.reader.HistoricalResolver method*), 50
 COLOR_SPACE (*pyhanko.pdf_utils.content.ResourceType attribute*), 30
 compress() (*pyhanko.pdf_utils.generic.StreamObject method*), 43
 compute_digest() (*pyhanko.sign.validation.EmbeddedPdfSignature method*), 77
 compute_integrity_info() (*pyhanko.sign.validation.EmbeddedPdfSignature method*), 76
 compute_tst_digest() (*pyhanko.sign.validation.EmbeddedPdfSignature method*), 77
 ConfigurableMixin (*class in pyhanko.pdf_utils.config_utils*), 30
 ConfigurationError, 30
 container_ref (*pyhanko.pdf_utils.generic.PdfObject attribute*), 38
 content() (*pyhanko.pdf_utils.text.TextBox property*), 52
 content_lines() (*pyhanko.pdf_utils.text.TextBox property*), 52
 CONTIGUOUS_BLOCK_FROM_START (*pyhanko.sign.validation.SignatureCoverageLevel*

attribute), 74
 coverage (pyhanko.sign.validation.PdfSignatureStatus attribute), 75
 crls (pyhanko.sign.validation.VRI attribute), 78

D

data() (pyhanko.pdf_utils.generic.StreamObject property), 42
 decode() (pyhanko.pdf_utils.filters.ASCII85Decode class method), 33
 decode() (pyhanko.pdf_utils.filters.ASCIIHexDecode class method), 33
 decode() (pyhanko.pdf_utils.filters.Decoder class method), 32
 decode() (pyhanko.pdf_utils.filters.FlateDecode class method), 34
 Decoder (class in pyhanko.pdf_utils.filters), 32
 DECODERS (in module pyhanko.pdf_utils.filters), 34
 decrypt() (pyhanko.pdf_utils.reader.PdfFileReader method), 48
 default_engine() (pyhanko.pdf_utils.font.SimpleFontEngine static method), 35
 DEFAULT_MD (in module pyhanko.sign.signers), 71
 DEFAULT_SIGNING_STAMP_STYLE (in module pyhanko.sign.signers), 71
 default_stamp_style (pyhanko.config.CLIConfig attribute), 82
 default_validation_context (pyhanko.config.CLIConfig attribute), 82
 DELIMITER_PATTERN (pyhanko.pdf_utils.generic.NameObject attribute), 41
 Dereferenceable (class in pyhanko.pdf_utils.generic), 36
 DictionaryObject (class in pyhanko.pdf_utils.generic), 42
 DIGEST_METHOD (pyhanko.sign.fields.SigSeedValFlags attribute), 57
 digest_methods (pyhanko.sign.fields.SigSeedValueSpec attribute), 59
 doc_mdp_update_value (pyhanko.sign.fields.SigFieldSpec attribute), 57
 docmdp_level() (pyhanko.sign.validation.EmbeddedPdfSignature property), 77
 docmdp_ok (pyhanko.sign.validation.PdfSignatureStatus attribute), 76
 docmdp_permissions (pyhanko.sign.signers.PdfSignatureMetadata attribute), 66
 DocMDPInfo (class in pyhanko.sign.validation), 77

DocumentSecurityStore (class in pyhanko.sign.validation), 78
 dummy_response() (pyhanko.sign.timestamps.TimeStamper method), 72

E

embed_subset() (pyhanko.pdf_utils.font.GlyphAccumulator method), 36
 embed_validation_info (pyhanko.sign.signers.PdfSignatureMetadata attribute), 65
 embedded_signatures() (pyhanko.pdf_utils.reader.PdfFileReader property), 49
 EmbeddedPdfSignature (class in pyhanko.sign.validation), 76
 encode() (pyhanko.pdf_utils.filters.ASCII85Decode class method), 33
 encode() (pyhanko.pdf_utils.filters.ASCIIHexDecode class method), 33
 encode() (pyhanko.pdf_utils.filters.Decoder class method), 32
 encode() (pyhanko.pdf_utils.filters.FlateDecode class method), 34
 encoded_data() (pyhanko.pdf_utils.generic.StreamObject property), 42
 encrypt() (pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter method), 45
 encrypted() (pyhanko.pdf_utils.reader.PdfFileReader property), 49
 ENTIRE_FILE (pyhanko.sign.validation.SignatureCoverageLevel attribute), 74
 ENTIRE_REVISION (pyhanko.sign.validation.SignatureCoverageLevel attribute), 74
 enumerate_sig_fields() (in module pyhanko.sign.fields), 62
 ETSI_RFC3161 (pyhanko.sign.fields.SigSeedSubFilter attribute), 60
 evaluate_modifications() (pyhanko.sign.validation.EmbeddedPdfSignature method), 77
 evaluate_signature_coverage() (pyhanko.sign.validation.EmbeddedPdfSignature method), 77
 EXCLUDE (pyhanko.sign.fields.FieldMDPAction attribute), 61
 EXT_G_STATE (pyhanko.pdf_utils.content.ResourceType attribute), 30
 extd_key_usage (pyhanko.sign.general.SignatureStatus attribute),

63
 extd_key_usage (py-
 hanko.sign.timestamps.TimestampSignatureStatus attribute), 72
 external_timestamp_data() (py-
 hanko.sign.validation.EmbeddedPdfSignature property), 76
 extra_commands() (pyhanko.stamp.QRStamp
 method), 86
 extra_commands() (pyhanko.stamp.TextStamp
 method), 85

F

feed_string() (py-
 hanko.pdf_utils.font.GlyphAccumulator method), 35
 field_mdp_spec (pyhanko.sign.fields.SigFieldSpec
 attribute), 57
 field_name (pyhanko.sign.signers.PdfSignatureMetadata
 attribute), 65
 field_name() (pyhanko.sign.validation.EmbeddedPdfSignature
 property), 76
 fieldmdp() (pyhanko.sign.validation.EmbeddedPdfSignature
 property), 77
 FieldMDPAction (class in pyhanko.sign.fields), 61
 FieldMDPSpec (class in pyhanko.sign.fields), 61
 fields (pyhanko.sign.fields.FieldMDPSpec attribute),
 61
 FILL_FORMS (pyhanko.sign.fields.MDPPerm attribute),
 60
 FILTER (pyhanko.sign.fields.SigSeedValFlags attribute),
 57
 find_cms_attribute() (in module py-
 hanko.sign.general), 63
 find_page_container() (py-
 hanko.pdf_utils.rw_common.PdfHandler method), 50
 find_page_for_modification() (py-
 hanko.pdf_utils.rw_common.PdfHandler method), 50
 flags (pyhanko.sign.fields.SigCertConstraints at-
 tribute), 58
 flags (pyhanko.sign.fields.SigSeedValueSpec attribute),
 59
 FlateDecode (class in pyhanko.pdf_utils.filters), 34
 FloatObject (class in pyhanko.pdf_utils.generic), 39
 FONT (pyhanko.pdf_utils.content.ResourceType at-
 tribute), 31
 font (pyhanko.pdf_utils.text.TextStyle attribute), 51
 font_file (pyhanko.pdf_utils.font.GlyphAccumulatorFactory
 attribute), 36
 font_size (pyhanko.pdf_utils.text.TextStyle attribute),
 51
 FontEngine (class in pyhanko.pdf_utils.font), 34

FORM_FILLING (pyhanko.sign.validation.ModificationLevel
 attribute), 75
 format_lock_dictionary() (py-
 hanko.sign.fields.SigFieldSpec method),
 57
 format_revinfo() (pyhanko.sign.signers.Signer
 static method), 67
 from_config() (py-
 hanko.pdf_utils.config_utils.ConfigurableMixin
 class method), 30
 from_pdf_object() (py-
 hanko.sign.fields.FieldMDPSpec class
 method), 61
 from_pdf_object() (py-
 hanko.sign.fields.SigCertConstraints class
 method), 58
 from_pdf_object() (py-
 hanko.sign.fields.SigSeedValueSpec class
 method), 59

G

generate_timestamp_field_name() (py-
 hanko.sign.signers.PdfSigner method), 70
 generate_timestamp_field_name() (py-
 hanko.sign.signers.PdfTimeStamper method),
 69
 generation (pyhanko.pdf_utils.generic.Reference at-
 tribute), 37
 generation() (pyhanko.pdf_utils.generic.IndirectObject
 property), 39
 get_container_ref() (py-
 hanko.pdf_utils.generic.PdfObject method),
 38
 get_default_text_params() (py-
 hanko.stamp.QRStamp method), 86
 get_default_text_params() (py-
 hanko.stamp.TextStamp method), 85
 get_historical_resolver() (py-
 hanko.pdf_utils.reader.PdfFileReader method),
 49
 get_historical_root() (py-
 hanko.pdf_utils.reader.PdfFileReader method),
 48
 get_object() (pyhanko.pdf_utils.generic.Dereferenceable
 method), 37
 get_object() (pyhanko.pdf_utils.generic.IndirectObject
 method), 38
 get_object() (pyhanko.pdf_utils.generic.PdfObject
 method), 38
 get_object() (pyhanko.pdf_utils.generic.Reference
 method), 37
 get_object() (pyhanko.pdf_utils.generic.TrailerReference
 method), 37

`get_object()` (`pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter` method), 45
`get_object()` (`pyhanko.pdf_utils.reader.HistoricalResolver` method), 49
`get_object()` (`pyhanko.pdf_utils.reader.PdfFileReader` method), 48
`get_object()` (`pyhanko.pdf_utils.rw_common.PdfHandler` method), 50
`get_object()` (`pyhanko.pdf_utils.writer.BasePdfFileWriter` method), 53
`get_pdf_handler()` (`pyhanko.pdf_utils.generic.Dereferenceable` method), 37
`get_pdf_handler()` (`pyhanko.pdf_utils.generic.IndirectObject` method), 39
`get_pdf_handler()` (`pyhanko.pdf_utils.generic.Reference` method), 37
`get_pdf_handler()` (`pyhanko.pdf_utils.generic.TrailerReference` method), 38
`get_stamp_height()` (`pyhanko.stamp.QRStamp` method), 86
`get_stamp_height()` (`pyhanko.stamp.TextStamp` method), 85
`get_stamp_style()` (`pyhanko.config.CLIFConfig` method), 82
`get_stamp_width()` (`pyhanko.stamp.TextStamp` method), 85
`get_text_height()` (`pyhanko.pdf_utils.text.TextBox` method), 52
`get_validation_context()` (`pyhanko.config.CLIFConfig` method), 82
`GlyphAccumulator` (class in `pyhanko.pdf_utils.font`), 35
`GlyphAccumulatorFactory` (class in `pyhanko.pdf_utils.font`), 36

H

`has_xref_stream` (`pyhanko.pdf_utils.reader.PdfFileReader` attribute), 48
`height()` (`pyhanko.pdf_utils.layout.BoxConstraints` property), 47
`height_defined()` (`pyhanko.pdf_utils.layout.BoxConstraints` property), 47
`HistoricalResolver` (class in `pyhanko.pdf_utils.reader`), 49
`HTTPTimeStamper` (class in `pyhanko.sign.timestamps`), 73

K

`KEY_USAGE` (`pyhanko.sign.fields.SigCertConstraintFlags` attribute), 60
`key_usage` (`pyhanko.sign.general.SignatureStatus` attribute), 63
`key_usage` (`pyhanko.sign.timestamps.TimestampSignatureStatus` attribute), 72

L

`last_startxref` (`pyhanko.pdf_utils.reader.PdfFileReader` attribute), 48
`leading` (`pyhanko.pdf_utils.text.TextStyle` attribute), 51

leading() (*pyhanko.pdf_utils.text.TextBox* property), 52

LEGAL_ATTESTATION (*pyhanko.sign.fields.SigSeedValFlags* attribute), 57

level (*pyhanko.config.LogConfig* attribute), 82

load() (*pyhanko.sign.signers.SimpleSigner* class method), 69

load_certs_from_pemder() (in module *pyhanko.sign.signers*), 71

load_pkcs12() (*pyhanko.sign.signers.SimpleSigner* class method), 68

location (*pyhanko.sign.signers.PdfSignatureMetadata* attribute), 65

LOCK_DOCUMENT (*pyhanko.sign.fields.SigSeedValFlags* attribute), 57

log_config (*pyhanko.config.CLIConfig* attribute), 82

LogConfig (class in *pyhanko.config*), 82

LTA_UPDATES (*pyhanko.sign.validation.ModificationLevel* attribute), 75

M

mark_update() (*pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter* method), 45

mark_update() (*pyhanko.pdf_utils.writer.BasePdfFileWriter* method), 53

md_algorithm (*pyhanko.sign.general.SignatureStatus* attribute), 63

md_algorithm (*pyhanko.sign.signers.PdfSignatureMetadata* attribute), 65

MDPPerm (class in *pyhanko.sign.fields*), 60

measure() (*pyhanko.pdf_utils.font.FontEngine* method), 34

measure() (*pyhanko.pdf_utils.font.GlyphAccumulator* method), 36

measure() (*pyhanko.pdf_utils.font.SimpleFontEngine* method), 35

merge_resources() (*pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter* method), 46

modification_level (*pyhanko.sign.validation.PdfSignatureStatus* attribute), 75

ModificationLevel (class in *pyhanko.sign.validation*), 74

module

- pyhanko*, 87
- pyhanko.cli*, 82
- pyhanko.config*, 82
- pyhanko.pdf_utils*, 56
- pyhanko.pdf_utils.barcodes*, 29
- pyhanko.pdf_utils.config_utils*, 30

- pyhanko.pdf_utils.content*, 30
- pyhanko.pdf_utils.filters*, 32
- pyhanko.pdf_utils.font*, 34
- pyhanko.pdf_utils.generic*, 36
- pyhanko.pdf_utils.images*, 44
- pyhanko.pdf_utils.incremental_writer*, 45
- pyhanko.pdf_utils.layout*, 47
- pyhanko.pdf_utils.misc*, 47
- pyhanko.pdf_utils.reader*, 48
- pyhanko.pdf_utils.rw_common*, 50
- pyhanko.pdf_utils.text*, 51
- pyhanko.pdf_utils.writer*, 52
- pyhanko.sign*, 82
- pyhanko.sign.beid*, 56
- pyhanko.sign.fields*, 56
- pyhanko.sign.general*, 62
- pyhanko.sign.pkcs11*, 64
- pyhanko.sign.signers*, 65
- pyhanko.sign.timestamps*, 72
- pyhanko.sign.validation*, 74
- pyhanko.stamp*, 83

N

name (*pyhanko.sign.signers.PdfSignatureMetadata* attribute), 65

NameObject (class in *pyhanko.pdf_utils.generic*), 41

NO_CHANGES (*pyhanko.sign.fields.MDPPerm* attribute), 60

NONE (*pyhanko.sign.validation.ModificationLevel* attribute), 75

NullObject (class in *pyhanko.pdf_utils.generic*), 39

NumberObject (class in *pyhanko.pdf_utils.generic*), 40

NumberPattern (*pyhanko.pdf_utils.generic.NumberObject* attribute), 40

O

object_streams (*pyhanko.pdf_utils.writer.PdfFileWriter* attribute), 55

ObjectStream (class in *pyhanko.pdf_utils.writer*), 52

ocsps (*pyhanko.sign.validation.VRI* attribute), 78

OID (*pyhanko.sign.fields.SigCertConstraintFlags* attribute), 60

on_page (*pyhanko.sign.fields.SigFieldSpec* attribute), 57

open_beid_session() (in module *pyhanko.sign.beid*), 56

original_bytes() (*pyhanko.pdf_utils.generic.ByteStringObject* property), 40

original_bytes() (pyhanko.pdf_utils.generic.TextStringObject property), 40

OTHER (pyhanko.sign.validation.ModificationLevel attribute), 75

output (pyhanko.config.LogConfig attribute), 82

output_version (pyhanko.pdf_utils.writer.BasePdfFileWriter attribute), 53

P

PADES (pyhanko.sign.fields.SigSeedSubFilter attribute), 60

PADES_LT (pyhanko.sign.validation.RevocationInfoValidationType attribute), 78

PADES_LTA (pyhanko.sign.validation.RevocationInfoValidationType attribute), 78

PageObject (class in pyhanko.pdf_utils.writer), 55

parse_cli_config() (in module pyhanko.config), 83

parse_logging_config() (in module pyhanko.config), 82

parse_output_spec() (pyhanko.config.LogConfig static method), 82

parse_trust_config() (in module pyhanko.config), 82

PATTERN (pyhanko.pdf_utils.content.ResourceType attribute), 30

pdf (pyhanko.pdf_utils.generic.Reference attribute), 37

pdf_date() (in module pyhanko.pdf_utils.generic), 44

pdf_name (in module pyhanko.pdf_utils.generic), 43

pdf_string() (in module pyhanko.pdf_utils.generic), 43

PdfContent (class in pyhanko.pdf_utils.content), 31

PdfError, 47

PdfFileReader (class in pyhanko.pdf_utils.reader), 48

PdfFileWriter (class in pyhanko.pdf_utils.writer), 55

PdfHandler (class in pyhanko.pdf_utils.rw_common), 50

PdfImage (class in pyhanko.pdf_utils.images), 44

PdfObject (class in pyhanko.pdf_utils.generic), 38

PdfReadError, 47

PdfResources (class in pyhanko.pdf_utils.content), 31

PdfSignatureMetadata (class in pyhanko.sign.signers), 65

PdfSignatureStatus (class in pyhanko.sign.validation), 75

PdfSigner (class in pyhanko.sign.signers), 70

PdfStreamError, 47

PdfTimeStamper (class in pyhanko.sign.signers), 69

PdfWriteError, 47

permission() (pyhanko.sign.validation.DocMDPInfo property), 78

pil_image() (in module pyhanko.pdf_utils.images), 44

PKCS11Signer (class in pyhanko.sign.pkcs11), 64

pkcs7_signature_mechanism (pyhanko.sign.general.SignatureStatus attribute), 63

prepare_object_stream() (pyhanko.pdf_utils.writer.BasePdfFileWriter method), 54

pretty_print_details() (pyhanko.sign.validation.PdfSignatureStatus method), 76

process_entries() (pyhanko.pdf_utils.config_utils.ConfigurableMixin class method), 30

process_entries() (pyhanko.pdf_utils.text.TextStyle class method), 51

process_entries() (pyhanko.stamp.TextStampStyle class method), 84

PROPERTIES (pyhanko.pdf_utils.content.ResourceType attribute), 31

pyhanko module, 87

pyhanko.cli module, 82

pyhanko.config module, 82

pyhanko.pdf_utils module, 56

pyhanko.pdf_utils.barcodes module, 29

pyhanko.pdf_utils.config_utils module, 30

pyhanko.pdf_utils.content module, 30

pyhanko.pdf_utils.filters module, 32

pyhanko.pdf_utils.font module, 34

pyhanko.pdf_utils.generic module, 36

pyhanko.pdf_utils.images module, 44

pyhanko.pdf_utils.incremental_writer module, 45

pyhanko.pdf_utils.layout module, 47

pyhanko.pdf_utils.misc module, 47

pyhanko.pdf_utils.reader

module, 48
 pyhanko.pdf_utils.rw_common
 module, 50
 pyhanko.pdf_utils.text
 module, 51
 pyhanko.pdf_utils.writer
 module, 52
 pyhanko.sign
 module, 82
 pyhanko.sign.beid
 module, 56
 pyhanko.sign.fields
 module, 56
 pyhanko.sign.general
 module, 62
 pyhanko.sign.pkcs11
 module, 64
 pyhanko.sign.signers
 module, 65
 pyhanko.sign.timestamps
 module, 72
 pyhanko.sign.validation
 module, 74
 pyhanko.stamp
 module, 83

Q

qr_default_width (pyhanko.stamp.QRStamp
 attribute), 86
 qr_size() (pyhanko.stamp.QRStamp property), 86
 qr_stamp_file() (in module pyhanko.stamp), 87
 QRStamp (class in pyhanko.stamp), 86
 QRStampStyle (class in pyhanko.stamp), 84

R

raw_get() (pyhanko.pdf_utils.generic.ArrayObject
 method), 41
 raw_get() (pyhanko.pdf_utils.generic.DictionaryObject
 method), 42
 RawContent (class in pyhanko.pdf_utils.content), 32
 read() (pyhanko.pdf_utils.reader.PdfFileReader
 method), 48
 read_certification_data() (in module py-
 hanko.sign.validation), 79
 read_dss() (pyhanko.sign.validation.DocumentSecurityStore
 class method), 79
 read_from_stream() (py-
 hanko.pdf_utils.generic.ArrayObject static
 method), 41
 read_from_stream() (py-
 hanko.pdf_utils.generic.BooleanObject static
 method), 39

read_from_stream() (py-
 hanko.pdf_utils.generic.DictionaryObject
 static method), 42
 read_from_stream() (py-
 hanko.pdf_utils.generic.IndirectObject static
 method), 39
 read_from_stream() (py-
 hanko.pdf_utils.generic.NameObject static
 method), 41
 read_from_stream() (py-
 hanko.pdf_utils.generic.NullObject static
 method), 39
 read_from_stream() (py-
 hanko.pdf_utils.generic.NumberObject static
 method), 40
 read_object() (in module py-
 hanko.pdf_utils.generic), 43
 reason (pyhanko.sign.signers.PdfSignatureMetadata
 attribute), 65
 REASONS (pyhanko.sign.fields.SigSeedValFlags at-
 tribute), 57
 reasons (pyhanko.sign.fields.SigSeedValueSpec at-
 tribute), 59
 Reference (class in pyhanko.pdf_utils.generic), 37
 register() (pyhanko.sign.general.CertificateStore
 method), 63
 register() (pyhanko.sign.general.SimpleCertificateStore
 method), 64
 register() (pyhanko.stamp.TextStamp method), 85
 register_annotation() (py-
 hanko.pdf_utils.writer.BasePdfFileWriter
 method), 54
 register_multiple() (py-
 hanko.sign.general.CertificateStore method),
 63
 register_vri() (py-
 hanko.sign.validation.DocumentSecurityStore
 method), 78
 render() (pyhanko.pdf_utils.barcodes.BarcodeBox
 method), 29
 render() (pyhanko.pdf_utils.content.PdfContent
 method), 32
 render() (pyhanko.pdf_utils.content.RawContent
 method), 32
 render() (pyhanko.pdf_utils.font.FontEngine method),
 34
 render() (pyhanko.pdf_utils.font.GlyphAccumulator
 method), 36
 render() (pyhanko.pdf_utils.font.SimpleFontEngine
 method), 35
 render() (pyhanko.pdf_utils.images.PdfImage
 method), 44
 render() (pyhanko.pdf_utils.text.TextBox method), 52
 render() (pyhanko.stamp.TextStamp method), 85

`request_cms()` (`pyhanko.sign.timestamps.TimeStamper` method), 72
`request_headers()` (`pyhanko.sign.timestamps.HTTPTimeStamper` method), 73
`request_tsa_response()` (`pyhanko.sign.timestamps.HTTPTimeStamper` method), 74
`request_tsa_response()` (`pyhanko.sign.timestamps.TimeStamper` method), 73
`RESERVED` (`pyhanko.sign.fields.SigCertConstraintFlags` attribute), 60
`ResourceManagementError`, 31
`resources()` (`pyhanko.pdf_utils.content.PdfContent` property), 32
`ResourceType` (class in `pyhanko.pdf_utils.content`), 30
`RevocationInfoValidationType` (class in `pyhanko.sign.validation`), 78
`revoked` (`pyhanko.sign.general.SignatureStatus` attribute), 62
`RFC`
 `RFC 3161`, 8, 23, 72
 `RFC 5652`, 4, 21, 62
 `RFC 8933`, 73, 74
`root()` (`pyhanko.pdf_utils.rw_common.PdfHandler` property), 50
`root_ref()` (`pyhanko.pdf_utils.reader.HistoricalResolvers` property), 49
`root_ref()` (`pyhanko.pdf_utils.reader.PdfFileReader` property), 48
`root_ref()` (`pyhanko.pdf_utils.rw_common.PdfHandler` property), 50
`root_ref()` (`pyhanko.pdf_utils.writer.BasePdfFileWriter` property), 53
S
`satisfied_by()` (`pyhanko.sign.fields.SigCertConstraints` method), 58
`seed_value_dict` (`pyhanko.sign.fields.SigFieldSpec` attribute), 57
`seed_value_ok` (`pyhanko.sign.validation.PdfSignatureStatus` attribute), 75
`self_reported_timestamp()` (`pyhanko.sign.validation.EmbeddedPdfSignature` property), 76
`set_resource()` (`pyhanko.pdf_utils.content.PdfContent` method), 31
`set_writer()` (`pyhanko.pdf_utils.content.PdfContent` method), 32
`setdefault()` (`pyhanko.pdf_utils.generic.DictionaryObject` method), 42
`SHADING` (`pyhanko.pdf_utils.content.ResourceType` attribute), 30
`sig_content_identifier()` (`pyhanko.sign.validation.DocumentSecurityStore` static method), 78
`sig_field` (`pyhanko.sign.validation.EmbeddedPdfSignature` attribute), 76
`sig_field_name` (`pyhanko.sign.fields.SigFieldSpec` attribute), 56
`sig_object` (`pyhanko.sign.validation.EmbeddedPdfSignature` attribute), 76
`SigCertConstraintFlags` (class in `pyhanko.sign.fields`), 60
`SigCertConstraints` (class in `pyhanko.sign.fields`), 57
`SigFieldSpec` (class in `pyhanko.sign.fields`), 56
`sign()` (`pyhanko.sign.signers.Signer` method), 67
`sign_pdf()` (in module `pyhanko.sign.signers`), 71
`sign_pdf()` (`pyhanko.sign.signers.PdfSigner` method), 70
`sign_raw()` (`pyhanko.sign.pkcs11.PKCS11Signer` method), 64
`sign_raw()` (`pyhanko.sign.signers.Signer` method), 66
`sign_raw()` (`pyhanko.sign.signers.SimpleSigner` method), 68
`signature_mechanism` (`pyhanko.sign.beid.BEIDSigner` attribute), 56
`signature_mechanism` (`pyhanko.sign.pkcs11.PKCS11Signer` attribute), 64
`signature_mechanism` (`pyhanko.sign.signers.Signer` attribute), 66
`SignatureCoverageLevel` (class in `pyhanko.sign.validation`), 74
`SignatureFormField` (class in `pyhanko.sign.fields`), 61
`SignatureStatus` (class in `pyhanko.sign.general`), 62
`SignatureValidationError`, 81
`signed_attrs()` (`pyhanko.sign.signers.Signer` method), 67
`signed_data` (`pyhanko.sign.validation.EmbeddedPdfSignature` attribute), 76
`Signer` (class in `pyhanko.sign.signers`), 66
`signer_cert` (`pyhanko.sign.validation.EmbeddedPdfSignature` attribute), 76
`signer_info()` (`pyhanko.sign.signers.Signer` method), 67
`signer_reported_dt` (`pyhanko.sign.validation.PdfSignatureStatus`

attribute), 76
 signing_cert (pyhanko.sign.general.SignatureStatus attribute), 63
 signing_cert (pyhanko.sign.signers.Signer attribute), 66
 signing_cert() (pyhanko.sign.pkcs11.PKCS11Signer property), 64
 signing_key (pyhanko.sign.signers.SimpleSigner attribute), 68
 SigningError, 64
 SigSeedSubFilter (class in pyhanko.sign.fields), 60
 SigSeedValFlags (class in pyhanko.sign.fields), 57
 SigSeedValueSpec (class in pyhanko.sign.fields), 59
 SigSeedValueValidationError, 81
 simple_cms_attribute() (in module pyhanko.sign.general), 63
 SimpleCertificateStore (class in pyhanko.sign.general), 64
 SimpleFontEngine (class in pyhanko.pdf_utils.font), 35
 SimpleSigner (class in pyhanko.sign.signers), 68
 STAMP_ART_CONTENT (in module pyhanko.stamp), 84
 stamp_qrsize (pyhanko.stamp.QRStampStyle attribute), 84
 stamp_styles (pyhanko.config.CLIConfig attribute), 82
 stamp_text (pyhanko.stamp.QRStampStyle attribute), 84
 stamp_text (pyhanko.stamp.TextStampStyle attribute), 84
 STDERR (pyhanko.config.StdLogOutput attribute), 82
 StdLogOutput (class in pyhanko.config), 82
 STDOUT (pyhanko.config.StdLogOutput attribute), 82
 stream_xrefs (pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter attribute), 46
 stream_xrefs (pyhanko.pdf_utils.writer.BasePdfFileWriter attribute), 53
 stream_xrefs (pyhanko.pdf_utils.writer.PdfFileWriter attribute), 55
 StreamObject (class in pyhanko.pdf_utils.generic), 42
 strip_filters() (pyhanko.pdf_utils.generic.StreamObject method), 42
 SUBFILTER (pyhanko.sign.fields.SigSeedValFlags attribute), 57
 subfilter (pyhanko.sign.signers.PdfSignatureMetadata attribute), 65
 subfilters (pyhanko.sign.fields.SigSeedValueSpec attribute), 59
 SUBJECT (pyhanko.sign.fields.SigCertConstraintFlags attribute), 60
 SUBJECT_DN (pyhanko.sign.fields.SigCertConstraintFlags attribute), 60
 subject_dn (pyhanko.sign.fields.SigCertConstraints attribute), 58
 subject_name() (pyhanko.sign.signers.Signer property), 67
 subjects (pyhanko.sign.fields.SigCertConstraints attribute), 58
 summarise_integrity_info() (pyhanko.sign.validation.EmbeddedPdfSignature method), 77
 summary() (pyhanko.sign.general.SignatureStatus method), 63
 summary_fields() (pyhanko.sign.general.SignatureStatus method), 63
 summary_fields() (pyhanko.sign.validation.PdfSignatureStatus method), 76
 SuspiciousModification, 81

T

text_box_style (pyhanko.stamp.TextStampStyle attribute), 83
 text_box_x() (pyhanko.stamp.QRStamp method), 86
 text_box_x() (pyhanko.stamp.TextStamp method), 85
 text_box_y() (pyhanko.stamp.TextStamp method), 85
 text_sep (pyhanko.pdf_utils.text.TextBoxStyle attribute), 51
 text_stamp_file() (in module pyhanko.stamp), 86
 text_x() (pyhanko.pdf_utils.text.TextBox method), 52
 text_y() (pyhanko.pdf_utils.text.TextBox method), 52
 TextBox (class in pyhanko.pdf_utils.text), 51
 TextBoxStyle (class in pyhanko.pdf_utils.text), 51
 TextStamp (class in pyhanko.stamp), 84
 TextStampStyle (class in pyhanko.stamp), 83
 TextStringObject (class in pyhanko.pdf_utils.generic), 40
 TextStyle (class in pyhanko.pdf_utils.text), 51
 time_tolerance (pyhanko.config.CLIConfig attribute), 82
 timestamp (pyhanko.sign.timestamps.TimestampSignatureStatus attribute), 72
 timestamp() (pyhanko.sign.timestamps.HTTPTimeStamper method), 73
 timestamp() (pyhanko.sign.timestamps.TimeStamper method), 73
 timestamp_field_name (pyhanko.sign.signers.PdfSignatureMetadata attribute), 66
 timestamp_format (pyhanko.stamp.TextStampStyle attribute), 84

`timestamp_pdf()` (py-
hanko.sign.signers.PdfTimeStamper method),
 69
`timestamp_required` (py-
hanko.sign.fields.SigSeedValueSpec attribute),
 59
`timestamp_server_url` (py-
hanko.sign.fields.SigSeedValueSpec attribute),
 59
`timestamp_validity` (py-
hanko.sign.validation.PdfSignatureStatus
 attribute), 76
`TimeStamper` (class in *pyhanko.sign.timestamps*), 72
`TimestampRequestError`, 74
`TimestampSignatureStatus` (class in py-
hanko.sign.timestamps), 72
`total_revisions()` (py-
hanko.pdf_utils.reader.PdfFileReader prop-
 erty), 48
`TrailerReference` (class in py-
hanko.pdf_utils.generic), 37
`trusted` (*pyhanko.sign.general.SignatureStatus* at-
 tribute), 62

U

`UnacceptableSignerError`, 64
`UNCLEAR` (*pyhanko.sign.validation.SignatureCoverageLevel*
 attribute), 74
`UNSUPPORTED` (*pyhanko.sign.fields.SigSeedValFlags*
 attribute), 57
`update_archival_timestamp_chain()` (py-
hanko.sign.signers.PdfTimeStamper method),
 70
`update_container()` (py-
hanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter
 method), 45
`update_container()` (py-
hanko.pdf_utils.writer.BasePdfFileWriter
 method), 53
`update_root()` (py-
hanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter
 method), 45
`URL` (*pyhanko.sign.fields.SigCertConstraintFlags* at-
 tribute), 60
`url_type` (*pyhanko.sign.fields.SigCertConstraints* at-
 tribute), 58
`use_pades_lta` (py-
hanko.sign.signers.PdfSignatureMetadata
 attribute), 66

V

`V` (*pyhanko.sign.fields.SigSeedValFlags* attribute), 57
`valid` (*pyhanko.sign.general.SignatureStatus* attribute),
 62

`validate_cert_usage()` (py-
hanko.sign.general.SignatureStatus class
 method), 63
`validate_cms_signature()` (in module py-
hanko.sign.validation), 80
`validate_pdf_ltv_signature()` (in module py-
hanko.sign.validation), 80
`validate_pdf_signature()` (in module py-
hanko.sign.validation), 80
`validation_context` (py-
hanko.sign.signers.PdfSignatureMetadata
 attribute), 66
`validation_contexts` (*pyhanko.config.CLIconfig*
 attribute), 82
`validation_path` (py-
hanko.sign.general.SignatureStatus attribute),
 63
`validation_paths()` (py-
hanko.sign.timestamps.TimeStamper method),
 72
`ValidationInfoReadingError`, 81
`vertical_center` (py-
hanko.pdf_utils.text.TextBoxStyle attribute),
 51
`VRI` (class in *pyhanko.sign.validation*), 78

W

`width()` (*pyhanko.pdf_utils.layout.BoxConstraints*
 property), 47
`width_defined()` (py-
hanko.pdf_utils.layout.BoxConstraints prop-
 erty), 47
`wrap_string()` (*pyhanko.pdf_utils.text.TextBox*
 method), 52
`write()` (*pyhanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter*
 method), 45
`write()` (*pyhanko.pdf_utils.writer.BasePdfFileWriter*
 method), 54
`write_in_place()` (py-
hanko.pdf_utils.incremental_writer.IncrementalPdfFileWriter
 method), 45
`write_to_stream()` (py-
hanko.pdf_utils.generic.ArrayObject method),
 41
`write_to_stream()` (py-
hanko.pdf_utils.generic.BooleanObject
 method), 39
`write_to_stream()` (py-
hanko.pdf_utils.generic.ByteStringObject
 method), 40
`write_to_stream()` (py-
hanko.pdf_utils.generic.DictionaryObject
 method), 42

```
write_to_stream() (py-  
    hanko.pdf_utils.generic.FloatObject method),  
40  
write_to_stream() (py-  
    hanko.pdf_utils.generic.IndirectObject  
    method), 39  
write_to_stream() (py-  
    hanko.pdf_utils.generic.NameObject method),  
41  
write_to_stream() (py-  
    hanko.pdf_utils.generic.NullObject method),  
39  
write_to_stream() (py-  
    hanko.pdf_utils.generic.NumberObject  
    method), 40  
write_to_stream() (py-  
    hanko.pdf_utils.generic.PdfObject method),  
38  
write_to_stream() (py-  
    hanko.pdf_utils.generic.StreamObject method),  
43  
write_to_stream() (py-  
    hanko.pdf_utils.generic.TextStringObject  
    method), 41  
writer (pyhanko.pdf_utils.content.PdfContent at-  
    tribute), 31
```

X

```
XOBJECT (pyhanko.pdf_utils.content.ResourceType at-  
    tribute), 31
```