python-gnupg Documentation

Release unknown

Isis Agora Lovecruft

04 December 2013

Contents

A Python interface for handling interactions with GnuPG, including keyfile generation, keyring maintainance, import and export, encryption and decryption, sending to and recieving from keyservers, and signing and verification.

Contents:

CHAPTER 1

gnupg package

1.1 gnupg module

This module contains public classes for working with GnuPG. To get started, do:

```
>>> import gnupg
>>> gpg = gnupg.GPG()
```

class gnupg.**GPG**(*binary=None*, *homedir=None*, *verbose=False*, *use_agent=False*, *keyring=None*, *secring=None*, *options=None*)

Bases: gnupg._meta.GPGBase

Python interface for handling interactions with GnuPG, including keyfile generation, keyring maintainance, import and export, encryption and decryption, sending to and recieving from keyservers, and signing and verification.

Initialize a GnuPG process wrapper.

Parameters

- **binary** (*str*) – Name for GnuPG binary executable. If the absolute path is not given, the evironment variable $PATH is searched for the executable and checked that the real uid/gid of the user has sufficient permissions.
- **homedir** (*str*) – Full pathname to directory containing the public and private keyrings. Default is whatever GnuPG defaults to.
- **verbose** (*str,int,bool*) – String or numeric value to pass to gpg's `--debug-level` option. See the gpg man page for the list of valid options. If False, debug output is not generated by the gpg binary. If True, defaults to `--debug-level basic`.
- **keyring** (*str*) – Name of keyring file containing public key data, if unspecified, defaults to 'pubring.gpg' in the `homedir` directory.
- **secring** (*str*) – Name of alternative secret keyring file to use. If left unspecified, this will default to using 'secring.gpg' in the :param:homedir directory, and create that file if it does not exist.
- **options** (*list*) – A list of additional options to pass to the GPG binary.

Raises `RuntimeError` with explanation message if there is a problem invoking gpg.

Example:

```
>>> import gnupg
GnuPG logging disabled...
>>> gpg = gnupg.GPG(homedir='doctests')
>>> gpg.keyring
'./doctests/pubring.gpg'
>>> gpg.secring
'./doctests/secring.gpg'
>>> gpg.use_agent
False
>>> gpg.binary
'/usr/bin/gpg'
```

_batch_limit = 25

_create_trustdb()
Create the trustdb file in our homedir, if it doesn't exist.

_export_ownertrust(*trustdb=None*)
Export ownertrust to a trustdb file.

If there is already a file named 'trustdb.gpg' in the current GnuPG homedir, it will be renamed to 'trustdb.gpg.bak'.

Parameters **trustdb** (*string*) – The path to the trustdb.gpg file. If not given, defaults to 'trustdb.gpg' in the current GnuPG homedir.

_fix_trustdb(*trustdb=None*)
Attempt to repair a broken trustdb.gpg file.

GnuPG>=2.0.x has this magical-seeming flag: '–fix-trustdb'. You'd think it would fix the the trustdb. Hah! It doesn't. Here's what it does instead:

(python-gnupg)!isiswintermute:(testing/digest-algo *$=)~/code/python-gnupg gpg2 –fix-trustdb gpg: You may try to re-create the trustdb using the commands: gpg: cd ~/.gnupg gpg: gpg2 –export-ownertrust > otrust.tmp gpg: rm trustdb.gpg gpg: gpg2 –import-ownertrust < otrust.tmp gpg: If that does not work, please consult the manual

Brilliant piece of software engineering right there.

Parameters **trustdb** (*string*) – The path to the trustdb.gpg file. If not given, defaults to 'trustdb.gpg' in the current GnuPG homedir.

_import_ownertrust(*trustdb=None*)
Import ownertrust from a trustdb file.

Parameters **trustdb** (*string*) – The path to the trustdb.gpg file. If not given, defaults to 'trustdb.gpg' in the current GnuPG homedir.

decrypt(*message*, ***kwargs*)
Decrypt the contents of a string or file-like object `message`.

Parameters

- **message** (file or str or `io.BytesIO`) – A string or file-like object to decrypt.
- **always_trust** (*bool*) – Instruct GnuPG to ignore trust checks.
- **passphrase** (*str*) – The passphrase for the secret key used for decryption.
- **output** (*str*) – A filename to write the decrypted output to.

decrypt_file (*filename*, *always_trust=False*, *passphrase=None*, *output=None*)
Decrypt the contents of a file-like object `filename` .

Parameters

- **filename** (*str*) – A file-like object to decrypt.
- **always_trust** (*bool*) – Instruct GnuPG to ignore trust checks.
- **passphrase** (*str*) – The passphrase for the secret key used for decryption.
- **output** (*str*) – A filename to write the decrypted output to.

delete_keys (*fingerprints*, *secret=False*, *subkeys=False*)
Delete a key, or list of keys, from the current keyring.

The keys must be refered to by their full fingerprint for GnuPG to delete them. If `secret=True`, the corresponding secret keyring will be deleted from `GPG.secring`.

Parameters

- **fingerprints** (*str or list or tuple*) – A string, or a list/tuple of strings, representing the fingerprint(s) for the key(s) to delete.
- **secret** (*bool*) – If True, delete the corresponding secret key(s) also. (default: False)
- **subkeys** (*bool*) – If True, delete the secret subkey first, then the public key. (default: False) Same as:

```
$ gpg --delete-secret-and-public-key 0x12345678
```

encrypt (*data*, **recipients*, ***kwargs*)
Encrypt the message contained in `data` to `recipients`.

Parameters

- **data** (*str*) – The file or bytestream to encrypt.
- **recipients** (*str*) – The recipients to encrypt to. Recipients must be specified keyID/fingerprint. Care should be taken in Python2.x to make sure that the given fingerprint is in fact a string and not a unicode object.
- **default_key** (*str*) – The keyID/fingerprint of the key to use for signing. If given, `data` will be encrypted and signed.
- **passphrase** (*str*) – If given, and `default_key` is also given, use this passphrase to unlock the secret portion of the `default_key` to sign the encrypted `data`. Otherwise, if `default_key` is not given, but `symmetric=True`, then use this passphrase as the passphrase for symmetric encryption. Signing and symmetric encryption should *not* be combined when sending the `data` to other recipients, else the passphrase to the secret key would be shared with them.
- **armor** (*bool*) – If True, ascii armor the output; otherwise, the output will be in binary format. (Default: True)
- **encrypt** (*bool*) – If True, encrypt the `data` using the `recipients` public keys. (Default: True)
- **symmetric** (*bool*) – If True, encrypt the `data` to `recipients` using a symmetric key. See the `passphrase` parameter. Symmetric encryption and public key encryption can be used simultaneously, and will result in a ciphertext which is decryptable with either the symmetric `passphrase` or one of the corresponding private keys.
- **always_trust** (*bool*) – If True, ignore trust warnings on recipient keys. If False, display trust warnings. (default: True)

- **output** (*str*) – The output file to write to. If not specified, the encrypted output is returned, and thus should be stored as an object in Python. For example:

```
>>> import shutil
>>> import gnupg
>>> if os.path.exists("doctests"):
...     shutil.rmtree("doctests")
>>> gpg = gnupg.GPG(homedir="doctests")
>>> key_settings = gpg.gen_key_input(key_type='RSA',
...     key_length=1024,
...     key_usage='ESCA',
...     passphrase='foo')
>>> key = gpg.gen_key(key_settings)
>>> message = "The crow flies at midnight."
>>> encrypted = str(gpg.encrypt(message, key.printprint))
>>> assert encrypted != message
>>> assert not encrypted.isspace()
>>> decrypted = str(gpg.decrypt(encrypted))
>>> assert not decrypted.isspace()
>>> decrypted
'The crow flies at midnight.'
```

Parameters

- **cipher_algo** (*str*) – The cipher algorithm to use. To see available algorithms with your version of GnuPG, do:

  ```
  $ gpg --with-colons --list-config ciphername.
  ```

  The default `cipher_algo`, if unspecified, is `'AES256'`.

- **digest_algo** (*str*) – The hash digest to use. Again, to see which hashes your GnuPG is capable of using, do:

  ```
  $ gpg --with-colons --list-config digestname.
  ```

  The default, if unspecified, is `'SHA512'`.

- **compress_algo** (*str*) – The compression algorithm to use. Can be one of `'ZLIB'`, `'BZIP2'`, `'ZIP'`, or `'Uncompressed'`.

See also: `GPGBase._encrypt()`

export_keys(*keyids*, *secret=False*, *subkeys=False*)

Export the indicated `keyids`.

Parameters

- **keyids** (*str*) – A keyid or fingerprint in any format that GnuPG will accept.
- **secret** (*bool*) – If True, export only the secret key.
- **subkeys** (*bool*) – If True, export the secret subkeys.

gen_key(*input*)

Generate a GnuPG key through batch file key generation. See `GPG.gen_key_input()` for creating the control input.

```
>>> import gnupg
>>> gpg = gnupg.GPG(homedir="doctests")
>>> key_input = gpg.gen_key_input()
>>> key = gpg.gen_key(key_input)
>>> assert key.fingerprint
```

Parameters **input** (*dict*) – A dictionary of parameters and values for the new key.

Returns The result mapping with details of the new key, which is a `parsers.GenKey` object.

gen_key_input(*separate_keyring=False*, *save_batchfile=False*, *testing=False*, ***kwargs*)

Generate a batch file for input to `GPG.gen_key()`.

The GnuPG batch file key generation feature allows unattended key generation by creating a file with special syntax and then providing it to: `gpg --gen-key --batch`. Batch files look like this:

Name-Real: Alice
Name-Email: alice@inter.net
Expire-Date: 2014-04-01
Key-Type: RSA
Key-Length: 4096
Key-Usage: cert
Subkey-Type: RSA
Subkey-Length: 4096
Subkey-Usage: encrypt,sign,auth
Passphrase: sekrit
%pubring foo.gpg
%secring sec.gpg
%commit

which is what this function creates for you. All of the available, non-control parameters are detailed below (control parameters are the ones which begin with a '%'). For example, to generate the batch file example above, use like this:

```
>>> import gnupg
GnuPG logging disabled...
>>> from __future__ import print_function
>>> gpg = gnupg.GPG(homedir='doctests')
>>> alice = { 'name_real': 'Alice',
...     'name_email': 'alice@inter.net',
...     'expire_date': '2014-04-01',
...     'key_type': 'RSA',
...     'key_length': 4096,
...     'key_usage': '',
...     'subkey_type': 'RSA',
...     'subkey_length': 4096,
...     'subkey_usage': 'encrypt,sign,auth',
...     'passphrase': 'sekrit' }
>>> alice_input = gpg.gen_key_input(**alice)
>>> print(alice_input)
Key-Type: RSA
Subkey-Type: RSA
Subkey-Usage: encrypt,sign,auth
Expire-Date: 2014-04-01
Passphrase: sekrit
Name-Real: Alice
Name-Email: alice@inter.net
```

```
Key-Length: 4096
Subkey-Length: 4096
%pubring ./doctests/alice.pubring.gpg
%secring ./doctests/alice.secring.gpg
%commit

>>> alice_key = gpg.gen_key(alice_input)
>>> assert alice_key is not None
>>> assert alice_key.fingerprint is not None
>>> message = "no one else can read my sekrit message"
>>> encrypted = gpg.encrypt(message, alice_key.fingerprint)
>>> assert isinstance(encrypted.data, str)
```

Parameters

- **separate_keyring** (*bool*) – Specify for the new key to be written to a separate pubring.gpg and secring.gpg. If True, `GPG.gen_key()` will automatically rename the separate keyring and secring to whatever the fingerprint of the generated key ends up being, suffixed with '.pubring' and '.secring' respectively.
- **save_batchfile** (*bool*) – Save a copy of the generated batch file to disk in a file named <name_real>.batch, where <name_real> is the `name_real` parameter stripped of punctuation, spaces, and non-ascii characters.
- **testing** (*bool*) – Uses a faster, albeit insecure random number generator to create keys. This should only be used for testing purposes, for keys which are going to be created and then soon after destroyed, and never for the generation of actual use keys.
- **name_real** (*str*) – The name field of the UID in the generated key.
- **name_comment** (*str*) – The comment in the UID of the generated key.
- **name_email** (*str*) – The email in the UID of the generated key. (default: $USER@$(hostname)) Remember to use UTF-8 encoding for the entirety of the UID. At least one of `name_real`, `name_comment`, or `name_email` must be provided, or else no user ID is created.
- **key_type** (*str*) – One of 'RSA', 'DSA', 'ELG-E', or 'default'. (default: 'RSA', if using GnuPG v1.x, otherwise 'default') Starts a new parameter block by giving the type of the primary key. The algorithm must be capable of signing. This is a required parameter. The algorithm may either be an OpenPGP algorithm number or a string with the algorithm name. The special value 'default' may be used for algo to create the default key type; in this case a `key_usage` should not be given and 'default' must also be used for `subkey_type`.
- **key_length** (*int*) – The requested length of the generated key in bits. (Default: 4096)
- **key_grip** (*str*) – hexstring This is an optional hexidecimal string which is used to generate a CSR or certificate for an already existing key. `key_length` will be ignored if this parameter is given.
- **key_usage** (*str*) – Space or comma delimited string of key usages. Allowed values are 'encrypt', 'sign', and 'auth'. This is used to generate the key flags. Please make sure that the algorithm is capable of this usage. Note that OpenPGP requires that all primary keys are capable of certification, so no matter what usage is given here, the 'cert' flag will be on. If no 'Key-Usage' is specified and the 'Key-Type' is not 'default', all allowed usages for that particular algorithm are used; if it is not given but 'default' is used the usage will be 'sign'.

- **subkey_type** (*str*) – This generates a secondary key (subkey). Currently only one subkey can be handled. See also `key_type` above.
- **subkey_length** (*int*) – The length of the secondary subkey in bits.
- **subkey_usage** (*str*) – Key usage for a subkey; similar to `key_usage`.
- **expire_date** (*int or str*) – Can be specified as an iso-date or as <int>[d|w|m|y] Set the expiration date for the key (and the subkey). It may either be entered in ISO date format (2000-08-15) or as number of days, weeks, month or years. The special notation "seconds=N" is also allowed to directly give an Epoch value. Without a letter days are assumed. Note that there is no check done on the overflow of the type used by OpenPGP for timestamps. Thus you better make sure that the given value make sense. Although OpenPGP works with time intervals, GnuPG uses an absolute value internally and thus the last year we can represent is 2105.
- **creation_date** (*str*) – Set the creation date of the key as stored in the key information and which is also part of the fingerprint calculation. Either a date like "1986-04-26" or a full timestamp like "19860426T042640" may be used. The time is considered to be UTC. If it is not given the current time is used.
- **passphrase** (*str*) – The passphrase for the new key. The default is to not use any passphrase. Note that GnuPG>=2.1.x will not allow you to specify a passphrase for batch key generation – GnuPG will ignore the `passphrase` parameter, stop, and ask the user for the new passphrase. However, we can put the command '%no-protection' into the batch key generation file to allow a passwordless key to be created, which can then have its passphrase set later with '–edit-key'.
- **preferences** (*str*) – Set the cipher, hash, and compression preference values for this key. This expects the same type of string as the sub-command 'setpref' in the –edit-key menu.
- **revoker** (*str*) – Should be given as 'algo:fpr' [case sensitive]. Add a designated revoker to the generated key. Algo is the public key algorithm of the designated revoker (i.e. RSA=1, DSA=17, etc.) fpr is the fingerprint of the designated revoker. The optional 'sensitive' flag marks the designated revoker as sensitive information. Only v4 keys may be designated revokers.
- **keyserver** (*str*) – This is an optional parameter that specifies the preferred keyserver URL for the key.
- **handle** (*str*) – This is an optional parameter only used with the status lines KEY_CREATED and KEY_NOT_CREATED. string may be up to 100 characters and should not contain spaces. It is useful for batch key generation to associate a key parameter block with a status line.

Return type str

Returns A suitable input string for the `GPG.gen_key()` method, the latter of which will create the new keypair.

see http://www.gnupg.org/documentation/manuals/gnupg-devel/Unattended-GPG-key-generation.html for more details.

import_keys(*key_data*)

Import the key_data into our keyring.

```
>>> import shutil
>>> shutil.rmtree("doctests")
>>> gpg = gnupg.GPG(homedir="doctests")
>>> inpt = gpg.gen_key_input()
```

```
>>> key1 = gpg.gen_key(inpt)
>>> print1 = str(key1.fingerprint)
>>> pubkey1 = gpg.export_keys(print1)
>>> seckey1 = gpg.export_keys(print1,secret=True)
>>> key2 = gpg.gen_key(inpt)
>>> print2 = key2.fingerprint
>>> seckeys = gpg.list_keys(secret=True)
>>> pubkeys = gpg.list_keys()
>>> assert print1 in seckeys.fingerprints
>>> assert print1 in pubkeys.fingerprints
>>> str(gpg.delete_keys(print1))
'Must delete secret key first'
>>> str(gpg.delete_keys(print1,secret=True))
'ok'
>>> str(gpg.delete_keys(print1))
'ok'
>>> pubkeys = gpg.list_keys()
>>> assert not print1 in pubkeys.fingerprints
>>> result = gpg.import_keys(pubkey1)
>>> pubkeys = gpg.list_keys()
>>> seckeys = gpg.list_keys(secret=True)
>>> assert not print1 in seckeys.fingerprints
>>> assert print1 in pubkeys.fingerprints
>>> result = gpg.import_keys(seckey1)
>>> assert result
>>> seckeys = gpg.list_keys(secret=True)
>>> assert print1 in seckeys.fingerprints
```

is_gpg1()
Returns true if using GnuPG <= 1.x.

is_gpg2()
Returns true if using GnuPG >= 2.x.

list_keys(*secret=False*)
List the keys currently in the keyring.

The GnuPG option '–show-photos', according to the GnuPG manual, "does not work with –with-colons", but since we can't rely on all versions of GnuPG to explicitly handle this correctly, we should probably include it in the args.

```
>>> import shutil
>>> shutil.rmtree("doctests")
>>> gpg = GPG(homedir="doctests")
>>> input = gpg.gen_key_input()
>>> result = gpg.gen_key(input)
>>> print1 = result.fingerprint
>>> result = gpg.gen_key(input)
>>> print2 = result.fingerprint
>>> pubkeys = gpg.list_keys()
>>> assert print1 in pubkeys.fingerprints
>>> assert print2 in pubkeys.fingerprints
```

list_packets(*raw_data*)
List the packet contents of a file.

list_sigs(**keyids*)
Get the signatures for each of the `keyids`.

```
>>> import gnupg
>>> gpg = gnupg.GPG(homedir="doctests")
>>> key_input = gpg.gen_key_input()
>>> key = gpg.gen_key(key_input)
>>> assert key.fingerprint
```

Return type dict

Returns A dictionary whose keys are the original keyid parameters, and whose values are lists of signatures.

recv_keys (**keyids*, ***kwargs*)

Import keys from a keyserver.

```
>>> gpg = gnupg.GPG(homedir="doctests")
>>> key = gpg.recv_keys('hkp://pgp.mit.edu', '3FF0DB166A7476EA')
>>> assert key
```

Parameters

- **keyids** (*str*) – Each `keyids` argument should be a string containing a keyid to request.
- **keyserver** (*str*) – The keyserver to request the `keyids` from; defaults to **:property:'gnupg.GPG.keyserver'**.

sign (*data*, ***kwargs*)

Create a signature for a message string or file.

Note that this method is not for signing other keys. (In GnuPG's terms, what we all usually call 'keysigning' is actually termed 'certification'...) Even though they are cryptographically the same operation, GnuPG differentiates between them, presumedly because these operations are also the same as the decryption operation. If the `key_usage``s ``C (certification), S (sign)`, and `E (encrypt)`, were all the same key, the key would "wear down" through frequent signing usage – since signing data is usually done often – meaning that the secret portion of the keypair, also used for decryption in this scenario, would have a statistically higher probability of an adversary obtaining an oracle for it (or for a portion of the rounds in the cipher algorithm, depending on the family of cryptanalytic attack used).

In simpler terms: this function isn't for signing your friends' keys, it's for something like signing an email.

Parameters

- **data** (*str or file*) – A string or file stream to sign.
- **default_key** (*str*) – The key to sign with.
- **passphrase** (*str*) – The passphrase to pipe to stdin.
- **clearsign** (*bool*) – If True, create a cleartext signature.
- **detach** (*bool*) – If True, create a detached signature.
- **binary** (*bool*) – If True, do not ascii armour the output.
- **digest_algo** (*str*) – The hash digest to use. Again, to see which hashes your GnuPG is capable of using, do:

  ```
  $ gpg --with-colons --list-config digestname.
  ```

  The default, if unspecified, is `'SHA512'`.

verify(*data*)
Verify the signature on the contents of the string `data`.

```
>>> gpg = GPG(homedir="doctests")
>>> input = gpg.gen_key_input(Passphrase='foo')
>>> key = gpg.gen_key(input)
>>> assert key
>>> sig = gpg.sign('hello',keyid=key.fingerprint,passphrase='bar')
>>> assert not sig
>>> sig = gpg.sign('hello',keyid=key.fingerprint,passphrase='foo')
>>> assert sig
>>> verify = gpg.verify(sig.data)
>>> assert verify
```

verify_file(*file*, *sig_file=None*)
Verify the signature on the contents of a file or file-like object. Can handle embedded signatures as well as detached signatures. If using detached signatures, the file containing the detached signature should be specified as the `sig_file`.

Parameters

- **file** (*file*) – A file descriptor object. Its type will be checked with `_util._is_file()`.
- **sig_file** (*str*) – A file containing the GPG signature data for `file`. If given, `file` is verified via this detached signature.

1.2 meta module

Contains the meta and base classes which `gnupg.GPG` inherits from. Mostly, you shouldn't ever need to touch anything in here, unless you're doing some serious hacking.

1.2.1 meta.py

Meta and base classes for hiding internal functions, and controlling attribute creation and handling.

class `gnupg._meta.`**GPGMeta**
Bases: `type`

Metaclass for changing the :meth:GPG.__init__ initialiser.

Detects running gpg-agent processes and the presence of a pinentry program, and disables pinentry so that python-gnupg can write the passphrase to the controlled GnuPG process without killing the agent.

classmethod _find_agent()
Discover if a gpg-agent process for the current euid is running.

If there is a matching gpg-agent process, set a `psutil.Process` instance containing the gpg-agent process' information to `GPG._agent_proc`.

Returns True if there exists a gpg-agent process running under the same effective user ID as that of this program. Otherwise, returns None.

class `gnupg._meta.`**GPGBase**(*binary=None*, *home=None*, *keyring=None*, *secring=None*, *use_agent=False*, *default_preference_list=None*, *verbose=False*, *options=None*)
Bases: `object`

Base class for property storage and to control process initialisation.

_decode_errors = 'strict'

_result_map = {'verify': <class 'gnupg._parsers.Verify'>, 'packets': <class 'gnupg._parsers.ListPackets'>, 'import': <c

default_preference_list
Get the default preference list.

keyserver
Get the current keyserver setting.

_homedir_getter()
Get the directory currently being used as GnuPG's homedir.

If unspecified, use `~/.config/python-gnupg/`

Return type str

Returns The absolute path to the current GnuPG homedir.

_homedir_setter(*directory*)
Set the directory to use as GnuPG's homedir.

If unspecified, use $HOME/.config/python-gnupg. If specified, ensure that the `directory` does not contain various shell escape characters. If `directory` is not found, it will be automatically created. Lastly, the `direcory` will be checked that the EUID has read and write permissions for it.

Parameters **directory** (*str*) – A relative or absolute path to the directory to use for storing/accessing GnuPG's files, including keyrings and the trustdb.

Raises `RuntimeError` if unable to find a suitable directory to use.

homedir

_generated_keys_getter()
Get the `homedir` subdirectory for storing generated keys.

Return type str

Returns The absolute path to the current GnuPG homedir.

_generated_keys_setter(*directory*)
Set the directory for storing generated keys.

If unspecified, use $GNUPGHOME/generated-keys. If specified, ensure that the `directory` does not contain various shell escape characters. If `directory` is not found, it will be automatically created. Lastly, the `direcory` will be checked that the EUID has read and write permissions for it.

Parameters **directory** (*str*) – A relative or absolute path to the directory to use for storing/accessing GnuPG's files, including keyrings and the trustdb.

Raises `RuntimeError` if unable to find a suitable directory to use.

_generated_keys

_make_args(*args*, *passphrase=False*)
Make a list of command line elements for GPG. The value of `args` will be appended only if it passes the checks in `parsers._sanitise()`. The `passphrase` argument needs to be True if a passphrase will be sent to GPG, else False.

Parameters

- **args** (*list*) – A list of strings of options and flags to pass to `GPG.binary`. This is input safe, meaning that these values go through strict checks (see `parsers._sanitise_list`) before being passed to to the input file descriptor for the

GnuPG process. Each string should be given exactly as it would be on the commandline interface to GnuPG, e.g. ["–cipher-algo AES256", "–default-key A3ADB67A2CDB8B35"].

- **passphrase** (*bool*) – If True, the passphrase will be sent to the stdin file descriptor for the attached GnuPG process.

_open_subprocess (*args=None*, *passphrase=False*)

Open a pipe to a GPG subprocess and return the file objects for communicating with it.

Parameters

- **args** (*list*) – A list of strings of options and flags to pass to `GPG.binary`. This is input safe, meaning that these values go through strict checks (see `parsers._sanitise_list`) before being passed to to the input file descriptor for the GnuPG process. Each string should be given exactly as it would be on the commandline interface to GnuPG, e.g. ["–cipher-algo AES256", "–default-key A3ADB67A2CDB8B35"].
- **passphrase** (*bool*) – If True, the passphrase will be sent to the stdin file descriptor for the attached GnuPG process.

_read_response (*stream*, *result*)

Reads all the stderr output from GPG, taking notice only of lines that begin with the magic [GNUPG:] prefix.

Calls methods on the response object for each valid token found, with the arg being the remainder of the status line.

Parameters

- **stream** – A byte-stream, file handle, or `subprocess.PIPE` to parse the for status codes from the GnuPG process.
- **result** – The result parser class from `_parsers` with which to call `handle_status` and parse the output of `stream`.

_read_data (*stream*, *result*)

Incrementally read from `stream` and store read data.

All data gathered from calling `stream.read()` will be concatenated and stored as `result.data`.

Parameters

- **stream** – An open file-like object to read() from.
- **result** – An instance of one of the result parsing classes from `GPGBase._result_mapping`.

_collect_output (*process*, *result*, *writer=None*, *stdin=None*)

Drain the subprocesses output streams, writing the collected output to the result. If a writer thread (writing to the subprocess) is given, make sure it's joined before returning. If a stdin stream is given, close it before returning.

_handle_io (*args*, *file*, *result*, *passphrase=False*, *binary=False*)

Handle a call to GPG - pass input data, collect output data.

_recv_keys (*keyids*, *keyserver=None*)

Import keys from a keyserver.

Parameters

- **keyids** (*str*) – A space-delimited string containing the keyids to request.
- **keyserver** (*str*) – The keyserver to request the `keyids` from; defaults to **:property:'gnupg.GPG.keyserver'**.

_sign_file(*file*, *default_key=None*, *passphrase=None*, *clearsign=True*, *detach=False*, *binary=False*, *digest_algo='SHA512'*)

Create a signature for a file.

Parameters

- **file** – The file stream (i.e. it's already been open()'d) to sign.
- **default_key** (*str*) – The key to sign with.
- **passphrase** (*str*) – The passphrase to pipe to stdin.
- **clearsign** (*bool*) – If True, create a cleartext signature.
- **detach** (*bool*) – If True, create a detached signature.
- **binary** (*bool*) – If True, do not ascii armour the output.
- **digest_algo** (*str*) – The hash digest to use. Again, to see which hashes your GnuPG is capable of using, do:

  ```
  $ gpg --with-colons --list-config digestname.
  ```

  The default, if unspecified, is `'SHA512'`.

_encrypt(*data*, *recipients*, *default_key=None*, *passphrase=None*, *armor=True*, *encrypt=True*, *symmetric=False*, *always_trust=True*, *output=None*, *cipher_algo='AES256'*, *digest_algo='SHA512'*, *compress_algo='ZLIB'*)

Encrypt the message read from the file-like object `data`.

Parameters

- **data** (*str*) – The file or bytestream to encrypt.
- **recipients** (*str*) – The recipients to encrypt to. Recipients must be specified keyID/fingerprint. Care should be taken in Python2.x to make sure that the given fingerprint is in fact a string and not a unicode object.
- **default_key** (*str*) – The keyID/fingerprint of the key to use for signing. If given, `data` will be encrypted and signed.
- **passphrase** (*str*) – If given, and `default_key` is also given, use this passphrase to unlock the secret portion of the `default_key` to sign the encrypted `data`. Otherwise, if `default_key` is not given, but `symmetric=True`, then use this passphrase as the passphrase for symmetric encryption. Signing and symmetric encryption should *not* be combined when sending the `data` to other recipients, else the passphrase to the secret key would be shared with them.
- **armor** (*bool*) – If True, ascii armor the output; otherwise, the output will be in binary format. (Default: True)
- **encrypt** (*bool*) – If True, encrypt the `data` using the `recipients` public keys. (Default: True)
- **symmetric** (*bool*) – If True, encrypt the `data` to `recipients` using a symmetric key. See the `passphrase` parameter. Symmetric encryption and public key encryption can be used simultaneously, and will result in a ciphertext which is decryptable with either the symmetric `passphrase` or one of the corresponding private keys.
- **always_trust** (*bool*) – If True, ignore trust warnings on recipient keys. If False, display trust warnings. (default: True)
- **output** (*str*) – The output file to write to. If not specified, the encrypted output is returned, and thus should be stored as an object in Python. For example:

```
>>> import shutil
>>> import gnupg
>>> if os.path.exists("doctests"):
...     shutil.rmtree("doctests")
>>> gpg = gnupg.GPG(homedir="doctests")
>>> key_settings = gpg.gen_key_input(key_type='RSA',
...                                  key_length=1024,
...                                  key_usage='ESCA',
...                                  passphrase='foo')
>>> key = gpg.gen_key(key_settings)
>>> message = "The crow flies at midnight."
>>> encrypted = str(gpg.encrypt(message, key.printprint))
>>> assert encrypted != message
>>> assert not encrypted.isspace()
>>> decrypted = str(gpg.decrypt(encrypted))
>>> assert not decrypted.isspace()
>>> decrypted
'The crow flies at midnight.'
```

Parameters

- **cipher_algo** (*str*) – The cipher algorithm to use. To see available algorithms with your version of GnuPG, do:

  ```
  $ gpg --with-colons --list-config ciphername.
  ```

  The default `cipher_algo`, if unspecified, is `'AES256'`.

- **digest_algo** (*str*) – The hash digest to use. Again, to see which hashes your GnuPG is capable of using, do:

  ```
  $ gpg --with-colons --list-config digestname.
  ```

  The default, if unspecified, is `'SHA512'`.

- **compress_algo** (*str*) – The compression algorithm to use. Can be one of `'ZLIB'`, `'BZIP2'`, `'ZIP'`, or `'Uncompressed'`.

1.3 parsers module

These are classes for parsing both user inputs and status file descriptor flags from GnuPG's output. The latter are used in order to determine what our GnuPG process is doing and retrieve information about its operations, which are stored in corresponding classes in `gnupg.GPG._result_dict`. Some status flags aren't handled yet – infomation on *all* of the flags (well, at least the documented ones. . .) can be found in the docs/DETAILS file in GnuPG's source, which has been included here as well.

1.3.1 parsers.py

Classes for parsing GnuPG status messages and sanitising commandline options.

exception `gnupg._parsers.`**`ProtectedOption`**
Bases: `exceptions.Exception`

Raised when the option passed to GPG is disallowed.

exception `gnupg._parsers.`**`UsageError`**
Bases: `exceptions.Exception`

Raised when incorrect usage of the API occurs..

gnupg._parsers.**_check_keyserver**(*location*)
Check that a given keyserver is a known protocol and does not contain shell escape characters.

Parameters **location** (*str*) – A string containing the default keyserver. This should contain the desired keyserver protocol which is supported by the keyserver, for example, the default is `'hkp://subkeys.pgp.net'`.

Return type str or None

Returns A string specifying the protocol and keyserver hostname, if the checks passed. If not, returns None.

gnupg._parsers.**_check_preferences**(*prefs*, *pref_type=None*)
Check cipher, digest, and compression preference settings.

MD5 is not allowed. This is not 1994.[0] SHA1 is allowed grudgingly.[1]

[0]: http://www.cs.colorado.edu/~jrblack/papers/md5e-full.pdf [1]: http://eprint.iacr.org/2008/469.pdf

gnupg._parsers.**_fix_unsafe**(*shell_input*)
Find characters used to escape from a string into a shell, and wrap them in quotes if they exist. Regex pilfered from python-3.x shlex module.

Parameters **shell_input** (*str*) – The input intended for the GnuPG process.

gnupg._parsers.**_hyphenate**(*input*, *add_prefix=False*)
Change underscores to hyphens so that object attributes can be easily tranlated to GPG option names.

Parameters

- **input** (*str*) – The attribute to hyphenate.
- **add_prefix** (*bool*) – If True, add leading hyphens to the input.

Return type str

Returns The `input` with underscores changed to hyphens.

gnupg._parsers.**_is_allowed**(*input*)
Check that an option or argument given to GPG is in the set of allowed options, the latter being a strict subset of the set of all options known to GPG.

Parameters **input** (*str*) – An input meant to be parsed as an option or flag to the GnuPG process. Should be formatted the same as an option or flag to the commandline gpg, i.e. “–encrypt-files”.

Variables

- **gnupg_options** (*frozenset*) – All known GPG options and flags.
- **allowed** (*frozenset*) – All allowed GPG options and flags, e.g. all GPG options and flags which we are willing to acknowledge and parse. If we want to support a new option, it will need to have its own parsing class and its name will need to be added to this set.

Return type Exception or str

Raise :exc:UsageError if `_allowed` is not a subset of `_possible`. ProtectedOption if `input` is not in the set `_allowed`.

Returns The original parameter `input`, unmodified and unsanitized, if no errors occur.

gnupg._parsers.**_is_hex**(*string*)
Check that a string is hexidecimal, with alphabetic characters capitalized and without whitespace.

Parameters **string** (*str*) – The string to check.

gnupg._parsers.**_is_string**(*thing*)
Python character arrays are a mess.

If Python2, check if `thing` is a `unicode()` or `str()`. If Python3, check if `thing` is a `str()`.

Parameters **thing** – The thing to check.

Returns `True` if `thing` is a "string" according to whichever version of Python we're running in.

gnupg._parsers.**_sanitise**(**args*)
Take an arg or the key portion of a kwarg and check that it is in the set of allowed GPG options and flags, and that it has the correct type. Then, attempt to escape any unsafe characters. If an option is not allowed, drop it with a logged warning. Returns a dictionary of all sanitised, allowed options.

Each new option that we support that is not a boolean, but instead has some extra inputs, i.e. "–encrypt-file foo.txt", will need some basic safety checks added here.

GnuPG has three-hundred and eighteen commandline flags. Also, not all implementations of OpenPGP parse PGP packets and headers in the same way, so there is added potential there for messing with calls to GPG.

For information on the PGP message format specification, see: https://www.ietf.org/rfc/rfc1991.txt

If you're asking, "Is this *really* necessary?": No, not really – we could just do a check as described here: https://xkcd.com/1181/

Parameters **args** (*str*) – (optional) The boolean arguments which will be passed to the GnuPG process.

Return type str

Returns `sanitised`

gnupg._parsers.**_sanitise_list**(*arg_list*)
A generator for iterating through a list of gpg options and sanitising them.

Parameters **arg_list** (*list*) – A list of options and flags for GnuPG.

Return type generator

Returns A generator whose next() method returns each of the items in `arg_list` after calling `_sanitise()` with that item as a parameter.

gnupg._parsers.**_get_options_group**(*group=None*)
Get a specific group of options which are allowed.

gnupg._parsers.**_get_all_gnupg_options**()
Get all GnuPG options and flags.

This is hardcoded within a local scope to reduce the chance of a tampered GnuPG binary reporting falsified option sets, i.e. because certain options (namedly the '–no-options' option, which prevents the usage of gpg.conf files) are necessary and statically specified in `gnupg.GPG._makeargs()`, if the inputs into Python are already controlled, and we were to summon the GnuPG binary to ask it for its options, it would be possible to receive a falsified options set missing the '–no-options' option in response. This seems unlikely, and the method is stupid and ugly, but at least we'll never have to debug whether or not an option *actually* disappeared in a different GnuPG version, or some funny business is happening.

These are the options as of GnuPG 1.4.12; the current stable branch of the 2.1.x tree contains a few more – if you need them you'll have to add them in here.

Variables **gnupg_options** (*frozenset*) – All known GPG options and flags.

Return type frozenset

Returns `gnupg_options`

gnupg._parsers.**nodata**(*status_code*)
Translate NODATA status codes from GnuPG to messages.

gnupg._parsers.**progress**(*status_code*)
Translate PROGRESS status codes from GnuPG to messages.

class gnupg._parsers.**GenKey**(*gpg*)
Bases: object

Handle status messages for key generation.

Calling the GenKey.__str__() method of this class will return the generated key's fingerprint, or a status string explaining the results.

type = None
'P':= primary, 'S':= subkey, 'B':= both

keyring = None
This will store the filename of the key's public keyring if GPG.gen_key_input() was called with separate_keyring=True

secring = None
This will store the filename of the key's secret keyring if GPG.gen_key_input() was called with separate_keyring=True

_handle_status(*key*, *value*)
Parse a status code from the attached GnuPG process.

Raises ValueError if the status message is unknown.

class gnupg._parsers.**DeleteResult**(*gpg*)
Bases: object

Handle status messages for –delete-keys and –delete-secret-keys

problem_reason = {'1': 'No such key', '3': 'Ambigious specification', '2': 'Must delete secret key first'}

_handle_status(*key*, *value*)
Parse a status code from the attached GnuPG process.

Raises ValueError if the status message is unknown.

class gnupg._parsers.**Sign**(*gpg*)
Bases: object

Parse GnuPG status messages for signing operations.

Parameters **gpg** – An instance of gnupg.GPG.

sig_type = None
The type of signature created.

sig_algo = None
The algorithm used to create the signature.

sig_hash_also = None
The hash algorithm used to create the signature.

fingerprint = None
The fingerprint of the signing keyid.

timestamp = None
The timestamp on the signature.

what = None
xxx fill me in

_handle_status(*key*, *value*)
Parse a status code from the attached GnuPG process.

Raises ValueError if the status message is unknown.

class gnupg._parsers.**ListKeys**(*gpg*)
Bases: list

Handle status messages for –list-keys.

Handles pub and uid (relating the latter to the former). Don't care about the following attributes/status messages (from doc/DETAILS):

crt = X.509 certificate
crs = X.509 certificate and private key available
ssb = secret subkey (secondary key)
uat = user attribute (same as user id except for field 10).
sig = signature
rev = revocation signature
pkd = public key data (special field format, see below)
grp = reserved for gpgsm
rvk = revocation key

key(*args*)

pub(*args*)

sec(*args*)

fpr(*args*)

uid(*args*)

sub(*args*)

_handle_status(*key*, *value*)

class gnupg._parsers.**ImportResult**(*gpg*)
Bases: object

Parse GnuPG status messages for key import operations.

Parameters **gpg** (gnupg.GPG) – An instance of gnupg.GPG.

_ok_reason = {'17': 'Contains private key', '16': 'Contains private key', '1': 'Entirely new key', '0': 'Not actually char

_problem_reason = {'1': 'Invalid Certificate', '0': 'No specific reason given', '3': 'Certificate Chain too long', '2': 'Iss

_fields = ['count', 'no_user_id', 'imported', 'imported_rsa', 'unchanged', 'n_uids', 'n_subk', 'n_sigs', 'n_revoc', 'sec_

_counts = OrderedDict([('count', 0), ('no_user_id', 0), ('imported', 0), ('imported_rsa', 0), ('unchanged', 0), ('n_uids', 0

fingerprints = []
A list of strings containing the fingerprints of the GnuPG keyIDs imported.

results = []
A list containing dictionaries with information gathered on keys imported.

_handle_status(*key*, *value*)
Parse a status code from the attached GnuPG process.

Raises ValueError if the status message is unknown.

summary()

x = 12

class gnupg._parsers.**Verify**(*gpg*)
Bases: object

Parser for status messages from GnuPG for certifications and signature verifications.

People often mix these up, or think that they are the same thing. While it is true that certifications and signatures *are* the same cryptographic operation – and also true that both are the same as the decryption operation – a distinction is made for important reasons.

A certification:

- is made on a key,
- can help to validate or invalidate the key owner's identity,
- can assign trust levels to the key (or to uids and/or subkeys that the key contains),
- and can be used in absense of in-person fingerprint checking to try to build a path (through keys whose fingerprints have been checked) to the key, so that the identity of the key's owner can be more reliable without having to actually physically meet in person.

A signature:

- is created for a file or other piece of data,
- can help to prove that the data hasn't been altered,
- and can help to prove that the data was sent by the person(s) in possession of the private key that created the signature, and for parsing portions of status messages from decryption operations.

There are probably other things unique to each that have been scatterbrainedly omitted due to the programmer sitting still and staring at GnuPG debugging logs for too long without snacks, but that is the gist of it.

Create a parser for verification and certification commands.

Parameters **gpg** – An instance of gnupg.GPG.

TRUST_UNDEFINED = 0

TRUST_NEVER = 1

TRUST_MARGINAL = 2

TRUST_FULLY = 3

TRUST_ULTIMATE = 4

TRUST_LEVELS = {'TRUST_UNDEFINED': 0, 'TRUST_FULLY': 3, 'TRUST_NEVER': 1, 'TRUST_MARGINAL': 2, '

valid = None
True if the signature is valid, False otherwise.

status = None
A string describing the status of the signature verification. Can be one of signature bad, signature good, signature valid, signature error, decryption failed, no public key, key exp, or key rev.

fingerprint = None
The fingerprint of the signing keyid.

pubkey_fingerprint = None
The fingerprint of the corresponding public key, which may be different if the signature was created with a subkey.

key_id = None
The keyid of the signing key.

signature_id = None
The id of the signature itself.

creation_date = None
The creation date of the signing key.

timestamp = None
The timestamp of the purported signature, if we are unable to parse and/or validate it.

sig_timestamp = None
The timestamp for when the valid signature was created.

username = None
The userid of the signing key which was used to create the signature.

expire_timestamp = None
When the signing key is due to expire.

trust_level = None
An integer 0-4 describing the trust level of the signature.

trust_text = None
The string corresponding to the `trust_level` number.

_handle_status(*key*, *value*)
Parse a status code from the attached GnuPG process.

Raises `ValueError` if the status message is unknown.

class gnupg._parsers.**Crypt**(*gpg*)
Bases: gnupg._parsers.Verify

Parser for internal status messages from GnuPG for `--encrypt`, `--decrypt`, and `--decrypt-files`.

data = None
A string containing the encrypted or decrypted data.

ok = None
True if the decryption/encryption process turned out okay.

status = None
A string describing the current processing status, or error, if one has occurred.

_handle_status(*key*, *value*)
Parse a status code from the attached GnuPG process.

Raises `ValueError` if the status message is unknown.

class gnupg._parsers.**ListPackets**(*gpg*)
Bases: object

Handle status messages for –list-packets.

status = None
A string describing the current processing status, or error, if one has occurred.

need_passphrase = None
True if the passphrase to a public/private keypair is required.

need_passphrase_sym = None
True if a passphrase for a symmetric key is required.

userid_hint = None
The keyid and uid which this data is encrypted to.

_handle_status(*key*, *value*)
Parse a status code from the attached GnuPG process.

Raises `ValueError` if the status message is unknown.

1.4 util module

You shouldn't really need to mess with this module either, it mostly deals with low-level IO and file handling operations, de-/en- coding issues, and setting up basic package facilities such as logging.

1.4.1 util.py

Extra utilities for python-gnupg.

gnupg._util.**find_encodings**(*enc=None*, *system=False*)
Find functions for encoding translations for a specific codec.

Parameters

- **enc** (*str*) – The codec to find translation functions for. It will be normalized by converting to lowercase, excluding everything which is not ascii, and hyphens will be converted to underscores.
- **system** (*bool*) – If True, find encodings based on the system's stdin encoding, otherwise assume utf-8.

Raises :exc:LookupError if the normalized codec, `enc`, cannot be found in Python's encoding translation map.

gnupg._util.**author_info**(*name*, *contact=None*, *public_key=None*)
Easy object-oriented representation of contributor info.

Parameters

- **name** (*str*) – The contributor´s name.
- **contact** (*str*) – The contributor´s email address or contact information, if given.
- **public_key** (*str*) – The contributor´s public keyid, if given.

gnupg._util.**_copy_data**(*instream*, *outstream*)
Copy data from one stream to another.

Parameters

- **instream** (`io.BytesIO` or `io.StringIO` or file) – A byte stream or open file to read from.
- **outstream** (*file*) – The file descriptor of a tmpfile to write to.

gnupg._util.**_create_if_necessary**(*directory*)
Create the specified directory, if necessary.

Parameters **directory** (*str*) – The directory to use.

Return type bool

Returns True if no errors occurred and the directory was created or existed beforehand, False otherwise.

gnupg._util.**create_uid_email**(*username=None*, *hostname=None*)
Create an email address suitable for a UID on a GnuPG key.

Parameters

- **username** (*str*) – The username portion of an email address. If None, defaults to the username of the running Python process.
- **hostname** (*str*) – The FQDN portion of an email address. If None, the hostname is obtained from gethostname(2).

Return type str

Returns A string formatted as <username>@<hostname>.

gnupg._util.**_deprefix**(*line*, *prefix*, *callback=None*)
Remove the prefix string from the beginning of line, if it exists.

Parameters

- **line** (*string*) – A line, such as one output by GnuPG's status-fd.
- **prefix** (*string*) – A substring to remove from the beginning of `line`. Case insensitive.
- **callback** (*callable*) – Function to call if the prefix is found. The signature to callback will be only one argument, the `line` without the `prefix`, i.e. `callback(line)`.

Return type string

Returns If the prefix was found, the `line` without the prefix is returned. Otherwise, the original `line` is returned.

gnupg._util.**_find_binary**(*binary=None*)
Find the absolute path to the GnuPG binary.

Also run checks that the binary is not a symlink, and check that our process real uid has exec permissions.

Parameters **binary** (*str*) – The path to the GnuPG binary.

Raises :exc:RuntimeError if it appears that GnuPG is not installed.

Return type str

Returns The absolute path to the GnuPG binary to use, if no exceptions occur.

gnupg._util.**_has_readwrite**(*path*)
Determine if the real uid/gid of the executing user has read and write permissions for a directory or a file.

Parameters **path** (*str*) – The path to the directory or file to check permissions for.

Return type bool

Returns True if real uid/gid has read+write permissions, False otherwise.

gnupg._util.**_is_file**(*input*)
Check that the size of the thing which is supposed to be a filename has size greater than zero, without following symbolic links or using :func:os.path.isfile.

Parameters **input** – An object to check.

Return type bool

Returns True if :param:input is file-like, False otherwise.

gnupg._util.**_is_stream**(*input*)
Check that the input is a byte stream.

Parameters **input** – An object provided for reading from or writing to.

Return type bool

Returns True if :param:input is a stream, False if otherwise.

gnupg._util.**_is_list_or_tuple**(*instance*)
Check that `instance` is a list or tuple.

Parameters **instance** – The object to type check.

Return type bool

Returns True if `instance` is a list or tuple, False otherwise.

gnupg._util.**_is_gpg1**(*version*)
Returns True if using GnuPG version 1.x.

Parameters **version** (*tuple*) – A tuple of three integers indication major, minor, and micro version numbers.

gnupg._util.**_is_gpg2**(*version*)
Returns True if using GnuPG version 2.x.

Parameters **version** (*tuple*) – A tuple of three integers indication major, minor, and micro version numbers.

gnupg._util.**_make_binary_stream**(*s*, *encoding*)
xxx fill me in

gnupg._util.**_make_passphrase**(*length=None*, *save=False*, *file=None*)
Create a passphrase and write it to a file that only the user can read.

This is not very secure, and should not be relied upon for actual key passphrases.

Parameters

- **length** (*int*) – The length in bytes of the string to generate.
- **file** (*file*) – The file to save the generated passphrase in. If not given, defaults to 'passphrase-<the real user id>-<seconds since epoch>' in the top-level directory.

gnupg._util.**_make_random_string**(*length*)
Returns a random lowercase, uppercase, alphanumerical string.

Parameters **length** (*int*) – The length in bytes of the string to generate.

gnupg._util.**_match_version_string**(*version*)
Sort a binary version string into major, minor, and micro integers.

Parameters **version** (*str*) – A version string in the form x.x.x

gnupg._util.**_next_year**()
Get the date of today plus one year.

Return type str

Returns The date of this day next year, in the format '%Y-%m-%d'.

gnupg._util.**_now**()
Get a timestamp for right now, formatted according to ISO 8601.

gnupg._util.**_separate_keyword**(*line*)
Split the line, and return (first_word, the_rest).

gnupg._util.**_threaded_copy_data**(*instream*, *outstream*)
Copy data from one stream to another in a separate thread.

Wraps `_copy_data()` in a `threading.Thread`.

Parameters

- **instream** (`io.BytesIO` or `io.StringIO`) – A byte stream to read from.
- **outstream** (*file*) – The file descriptor of a tmpfile to write to.

gnupg._util.**_utc_epoch**()
Get the seconds since epoch.

gnupg._util.**_which**(*executable*, *flags=1*)
Borrowed from Twisted's :mod:twisted.python.proutils .

Search PATH for executable files with the given name.

On newer versions of MS-Windows, the PATHEXT environment variable will be set to the list of file extensions for files considered executable. This will normally include things like ".EXE". This fuction will also find files with the given name ending with any of these extensions.

On MS-Windows the only flag that has any meaning is os.F_OK. Any other flags will be ignored.

Note: This function does not help us prevent an attacker who can already manipulate the environment's PATH settings from placing malicious code higher in the PATH. It also does happily follows links.

Parameters

- **name** (*str*) – The name for which to search.
- **flags** (*int*) – Arguments to L{os.access}.

Return type list

Returns A list of the full paths to files found, in the order in which they were found.

gnupg._util.**_write_passphrase**(*stream*, *passphrase*, *encoding*)
Write the passphrase from memory to the GnuPG process' stdin.

Parameters

- **stream** (*file, :class:BytesIO, or :class:StringIO*) – The input file descriptor to write the password to.
- **passphrase** (*str*) – The passphrase for the secret key material.
- **encoding** (*str*) – The data encoding expected by GnuPG. Usually, this is `sys.getfilesystemencoding()`.

gnupg._util.**InheritableProperty**
Based on the emulation of PyProperty_Type() in Objects/descrobject.c

class gnupg._util.**Storage**
Bases: `dict`

A dictionary where keys are stored as class attributes.

For example, `obj.foo` can be used in addition to `obj['foo']`:

```
>>> o = Storage(a=1)
>>> o.a
1
>>> o['a']
1
>>> o.a = 2
>>> o['a']
2
>>> del o.a
>>> o.a
None
```

1.5 About this fork

This is a modified version of python-gnupg, (forked from version 0.3.2) which was created by Vinay Sajip, which itself is a modification of GPG.py written by Steve Traugott, which in turn is a modification of the pycrypto GnuPG interface written by A.M. Kuchling.

This version is patched to sanitize untrusted inputs, due to the necessity of executing `subprocess.Popen([...], shell=True)` in order to communicate with GnuPG. Several speed improvements were also made based on code profiling, and the API has been cleaned up to support an easier, more Pythonic, interaction.

1.6 Previous Authors' Documentation

Steve Traugott's documentation:

> Portions of this module are derived from A.M. Kuchling's well-designed GPG.py, using Richard Jones' updated version 1.3, which can be found in the pycrypto CVS repository on Sourceforge:
>
> http://pycrypto.cvs.sourceforge.net/viewvc/pycrypto/gpg/GPG.py
>
> This module is *not* forward-compatible with amk's; some of the old interface has changed. For instance, since I've added decrypt functionality, I elected to initialize with a 'gpghome' argument instead of 'keyring', so that gpg can find both the public and secret keyrings. I've also altered some of the returned objects in order for the caller to not have to know as much about the internals of the result classes.
>
> While the rest of ISconf is released under the GPL, I am releasing this single file under the same terms that A.M. Kuchling used for pycrypto.
>
> Steve Traugott, stevegt@terraluna.org
> Thu Jun 23 21:27:20 PDT 2005

Vinay Sajip's documentation:

> This version of the module has been modified from Steve Traugott's version (see http://trac.t7a.org/isconf/browser/trunk/lib/python/isconf/GPG.py) by

Vinay Sajip to make use of the subprocess module (Steve's version uses os.fork() and so does not work on Windows). Renamed to gnupg.py to avoid confusion with the previous versions.

A unittest harness (test_gnupg.py) has also been added.

CHAPTER 2

Source, license, & bug reports

The source code which was used to generate this documentation is accessible by clicking the little [source] links next to the docs. Current source code can be found in this github repository. The **master** branch always reflects the latest release, all releases are tagged with signed, annotated git tags, and the **develop** branch represents the state of the next release.

This package is released under GPLv3 or greater.

If you find a bug, or would like to request a feature, please use our public bugtracker on github. Patches warmly welcome.

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

g

gnupg, **??**
gnupg._meta, **??**
gnupg._parsers, **??**
gnupg._util, **??**