Python

unknown

Jan 20, 2023

CONTENTS

CHAPTER

ONE

HISTORY

class keyrings.alt.file.**Encrypted**

Bases: object

PyCryptodome-backed Encryption support

block_size = 32

scheme = '[PBKDF2] AES256.CFB'

version = '1.0'

class keyrings.alt.file.**EncryptedKeyring**

Bases: *Encrypted*, Keyring

PyCryptodome File Keyring

decrypt(*password_encrypted*, *assoc=None*)

Given a password encrypted by a previous call to *encrypt*, and assoc (byte string, optional), return the original byte string.

assoc provides associated data (typically: service and username)

encrypt(*password*, *assoc=None*)

Given a password (byte string) and assoc (byte string, optional), return an encrypted byte string.

assoc provides associated data (typically: service and username)

filename = 'crypted_pass.cfg'

keyring_key

Much like the property builtin, but only implements __get__, making it a non-data property, and can be subsequently reset.

See http://users.rcn.com/python/download/Descriptor.htm for more information.

```
>>> class X(object):
...   @NonDataProperty
...   def foo(self):
...     return 3
>>> x = X()
>>> x.foo
3
>>> x.foo = 4
>>> x.foo
4
```

'…' below should be 'jaraco.classes' but for pytest-dev/pytest#3396 >>> X.foo <….properties.NonDataProperty object at …>

priority = 0.6

pw_prefix = b'pw:'

class keyrings.alt.file.**PlaintextKeyring**

Bases: Keyring

Simple File Keyring with no encryption

decrypt(*password_encrypted*, *assoc=None*)

Directly return encrypted password, ignore associated data.

encrypt(*password*, *assoc=None*)

Directly return the password itself, ignore associated data.

filename = 'keyring_pass.cfg'

priority = 0.5

Applicable for all platforms, but not recommended

scheme = 'no encyption'

version = '1.0'

class keyrings.alt.Gnome.**Keyring**

Bases: KeyringBackend

Gnome Keyring

KEYRING_NAME = None

Name of the keyring in which to store the passwords. Use None for the default keyring.

delete_password(*service*, *username*)

Delete the password for the username of the service.

get_password(*service*, *username*)

Get password of the username for the service

property keyring_name

priority

Like @property but applies at the class level.

```
>>> class X(metaclass=classproperty.Meta):
...   val = None
...   @classproperty
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
```

(continues on next page)

(continued from previous page)

```
>>> x = X()
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Setting the property on an instance affects the class.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo
5
>>> vars(x)
{}
>>> X().foo
5
```

Attempting to set an attribute where no setter was defined results in an AttributeError:

```
>>> class GetOnly(metaclass=classproperty.Meta):
...   @classproperty
...   def foo(cls):
...     return 'bar'
>>> GetOnly.foo = 3
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

It is also possible to wrap a classmethod or staticmethod in a classproperty.

```
>>> class Static(metaclass=classproperty.Meta):
...   @classproperty
...   @classmethod
...   def foo(cls):
...     return 'foo'
...   @classproperty
...   @staticmethod
...   def bar():
...     return 'bar'
>>> Static.foo
'foo'
>>> Static.bar
'bar'
```

Legacy

For compatibility, if the metaclass isn't specified, the legacy behavior will be invoked.

```
>>> class X:
...   val = None
...   @classproperty
```

(continues on next page)

(continued from previous page)

```
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
>>> x = X()
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Note, because the metaclass was not specified, setting a value on an instance does not have the intended effect.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo  # should be 5
4
>>> vars(x)  # should be empty
{'foo': 5}
>>> X().foo  # should be 5
4
```

set_password(*service*, *username*, *password*)

Set password for the username of the service

class keyrings.alt.Google.**DocsKeyring**(*credential*, *source*, *crypter*, *collection=None*, *client=None*, *can_create=True*, *input_getter=<built-in function input>*)

Bases: KeyringBackend

Backend that stores keyring on Google Docs. Note that login and any other initialisation is deferred until it is actually required to allow this keyring class to be added to the global _all_keyring list.

CONFLICT = -1

FAIL = 0

OK = 1

property client

property collection

delete_password(*service*, *username*)

Delete the password for the username of the service.

If the backend cannot delete passwords, raise PasswordDeleteError.

get_password(*service*, *username*)

Get password of the username for the service

keyring_title = 'GoogleKeyring'

priority

Like @property but applies at the class level.

```
>>> class X(metaclass=classproperty.Meta):
...   val = None
...   @classproperty
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
>>> x = X()
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Setting the property on an instance affects the class.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo
5
>>> vars(x)
{}
>>> X().foo
5
```

Attempting to set an attribute where no setter was defined results in an AttributeError:

```
>>> class GetOnly(metaclass=classproperty.Meta):
...   @classproperty
...   def foo(cls):
...     return 'bar'
>>> GetOnly.foo = 3
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

It is also possible to wrap a classmethod or staticmethod in a classproperty.

```
>>> class Static(metaclass=classproperty.Meta):
...   @classproperty
```

(continues on next page)

(continued from previous page)

```
...   @classmethod
...   def foo(cls):
...     return 'foo'
...   @classproperty
...   @staticmethod
...   def bar():
...     return 'bar'
>>> Static.foo
'foo'
>>> Static.bar
'bar'
```

Legacy

For compatibility, if the metaclass isn't specified, the legacy behavior will be invoked.

```
>>> class X:
...   val = None
...   @classproperty
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
>>> x = X()
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Note, because the metaclass was not specified, setting a value on an instance does not have the intended effect.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo  # should be 5
4
>>> vars(x)  # should be empty
{'foo': 5}
>>> X().foo  # should be 5
4
```

set_password(*service*, *username*, *password*)

Set password for the username of the service

class keyrings.alt.Google.**EnvironCredential**

Bases: EnvironCredential

Retrieve credentials from specifically named environment variables

class keyrings.alt.Google.**KeyczarDocsKeyring**

Bases: *DocsKeyring*

Google Docs keyring using keyczar initialized from environment variables

supported()

Return if this keyring supports current environment: -1: not applicable

0: suitable 1: recommended

class keyrings.alt.keyczar.**BaseCrypter**

Bases: Crypter

Base Keyczar keyset encryption and decryption. The keyset initialisation is deferred until required.

property crypter

The actual keyczar crypter

decrypt(*value*)

Decrypt the value.

encrypt(*value*)

Encrypt the value.

abstract property encrypting_keyset_location

Location for the encrypting keyset. Use None to indicate that the main keyset is not encrypted

abstract property keyset_location

Location for the main keyset that may be encrypted or not

class keyrings.alt.keyczar.**Crypter**(*keyset_location*, *encrypting_keyset_location=None*)

Bases: *BaseCrypter*

A Keyczar crypter using locations specified in the constructor

property encrypting_keyset_location

Location for the encrypting keyset. Use None to indicate that the main keyset is not encrypted

property keyset_location

Location for the main keyset that may be encrypted or not

class keyrings.alt.keyczar.**EnvironCrypter**

Bases: *BaseCrypter*

A Keyczar crypter using locations specified by environment vars

ENC_KEYSET_ENV_VAR = 'KEYRING_KEYCZAR_ENCRYPTING_LOCATION'

KEYSET_ENV_VAR = 'KEYRING_KEYCZAR_ENCRYPTED_LOCATION'

property encrypting_keyset_location

Location for the encrypting keyset. Use None to indicate that the main keyset is not encrypted

property keyset_location

Location for the main keyset that may be encrypted or not

keyrings.alt.keyczar.**has_keyczar**()

class keyrings.alt.multi.**MultipartKeyringWrapper**(*keyring*, *max_password_size=512*)

Bases: KeyringBackend

A wrapper around an existing keyring that breaks the password into smaller parts to handle implementations that have limits on the maximum length of passwords i.e. Windows Vault

delete_password(*service*, *username*)

Delete the password for the username of the service.

If the backend cannot delete passwords, raise PasswordDeleteError.

get_password(*service*, *username*)

Get password of the username for the service

priority = 0

set_password(*service*, *username*, *password*)

Set password for the username of the service

class keyrings.alt.pyfs.**BasicKeyring**(*crypter*, *filename=None*, *can_create=True*, *cache_timeout=None*)

Bases: KeyringBackend

BasicKeyring is a Pyfilesystem-based implementation of keyring.

It stores the password directly in the file, and supports encryption and decryption. The encrypted password is stored in base64 format. Being based on Pyfilesystem the file can be local or network-based and served by any of the filesystems supported by Pyfilesystem including Amazon S3, FTP, WebDAV, memory and more.

property config

load the passwords from the config file

decrypt(*password_encrypted*)

Decrypt the password.

delete_password(*service*, *username*)

Delete the password for the username of the service.

If the backend cannot delete passwords, raise PasswordDeleteError.

encrypt(*password*)

Encrypt the password.

file_path

Much like the property builtin, but only implements __get__, making it a non-data property, and can be subsequently reset.

See http://users.rcn.com/python/download/Descriptor.htm for more information.

```
>>> class X(object):
...   @NonDataProperty
...   def foo(self):
...     return 3
>>> x = X()
>>> x.foo
3
>>> x.foo = 4
>>> x.foo
4
```

'...' below should be 'jaraco.classes' but for pytest-dev/pytest#3396 >>> X.foo <....properties.NonDataProperty object at ...>

property filename

The filename used to store the passwords.

get_password(*service*, *username*)

Read the password from the file.

priority

Like @property but applies at the class level.

```
>>> class X(metaclass=classproperty.Meta):
...   val = None
...   @classproperty
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
>>> x = X()
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Setting the property on an instance affects the class.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo
5
>>> vars(x)
{}
>>> X().foo
5
```

Attempting to set an attribute where no setter was defined results in an AttributeError:

```
>>> class GetOnly(metaclass=classproperty.Meta):
...   @classproperty
...   def foo(cls):
...     return 'bar'
>>> GetOnly.foo = 3
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

It is also possible to wrap a classmethod or staticmethod in a classproperty.

```
>>> class Static(metaclass=classproperty.Meta):
...   @classproperty
...   @classmethod
...   def foo(cls):
...     return 'foo'
...   @classproperty
...   @staticmethod
...   def bar():
...     return 'bar'
>>> Static.foo
'foo'
>>> Static.bar
'bar'
```

Legacy

For compatibility, if the metaclass isn't specified, the legacy behavior will be invoked.

```
>>> class X:
...   val = None
...   @classproperty
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
>>> x = X()
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Note, because the metaclass was not specified, setting a value on an instance does not have the intended effect.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo  # should be 5
4
>>> vars(x)  # should be empty
{'foo': 5}
>>> X().foo  # should be 5
4
```

set_password(*service*, *username*, *password*)

Write the password in the file.

class keyrings.alt.pyfs.**EncryptedKeyring**(*crypter*, *filename=None*, *can_create=True*, *cache_timeout=None*)

Bases: *BasicKeyring*

Encrypted Pyfilesystem Keyring

class keyrings.alt.pyfs.**KeyczarKeyring**

Bases: *EncryptedKeyring*

Encrypted Pyfilesystem Keyring using Keyczar keysets specified in environment vars

class keyrings.alt.pyfs.**PlaintextKeyring**(*filename=None*, *can_create=True*, *cache_timeout=None*)

Bases: *BasicKeyring*

Unencrypted Pyfilesystem Keyring

keyrings.alt.pyfs.**has_pyfs**()

Does this environment have pyfs 1.x installed? Should return False even when Mercurial's Demand Import allowed import of fs.*.

class keyrings.alt.Windows.**EncryptedKeyring**

Bases: Keyring

A File-based keyring secured by Windows Crypto API.

decrypt(*password_encrypted*, *assoc=None*)

Decrypt the password using the CryptAPI.

encrypt(*password*, *assoc=None*)

Encrypt the password using the CryptAPI.

filename = 'wincrypto_pass.cfg'

priority

Like @property but applies at the class level.

```
>>> class X(metaclass=classproperty.Meta):
...   val = None
...   @classproperty
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
>>> x = X()
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Setting the property on an instance affects the class.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo
5
>>> vars(x)
{}
>>> X().foo
5
```

Attempting to set an attribute where no setter was defined results in an AttributeError:

```
>>> class GetOnly(metaclass=classproperty.Meta):
...   @classproperty
...   def foo(cls):
...     return 'bar'
>>> GetOnly.foo = 3
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

It is also possible to wrap a classmethod or staticmethod in a classproperty.

```
>>> class Static(metaclass=classproperty.Meta):
...   @classproperty
...   @classmethod
...   def foo(cls):
...     return 'foo'
...   @classproperty
...   @staticmethod
...   def bar():
...     return 'bar'
>>> Static.foo
'foo'
>>> Static.bar
'bar'
```

Legacy

For compatibility, if the metaclass isn't specified, the legacy behavior will be invoked.

```
>>> class X:
...   val = None
...   @classproperty
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
>>> x = X()
```

(continues on next page)

(continued from previous page)

```
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Note, because the metaclass was not specified, setting a value on an instance does not have the intended effect.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo  # should be 5
4
>>> vars(x)  # should be empty
{'foo': 5}
>>> X().foo  # should be 5
4
```

version = '1.0'

class keyrings.alt.Windows.**RegistryKeyring**

Bases: KeyringBackend

RegistryKeyring is a keyring which use Windows CryptAPI to encrypt the user's passwords and store them under registry keys

delete_password(*service*, *username*)

Delete the password for the username of the service.

get_password(*service*, *username*)

Get password of the username for the service

priority

Like @property but applies at the class level.

```
>>> class X(metaclass=classproperty.Meta):
...   val = None
...   @classproperty
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
>>> x = X()
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Setting the property on an instance affects the class.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo
5
>>> vars(x)
{}
>>> X().foo
5
```

Attempting to set an attribute where no setter was defined results in an AttributeError:

```
>>> class GetOnly(metaclass=classproperty.Meta):
...   @classproperty
...   def foo(cls):
...     return 'bar'
>>> GetOnly.foo = 3
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

It is also possible to wrap a classmethod or staticmethod in a classproperty.

```
>>> class Static(metaclass=classproperty.Meta):
...   @classproperty
...   @classmethod
...   def foo(cls):
...     return 'foo'
...   @classproperty
...   @staticmethod
...   def bar():
...     return 'bar'
>>> Static.foo
'foo'
>>> Static.bar
'bar'
```

Legacy

For compatibility, if the metaclass isn't specified, the legacy behavior will be invoked.

```
>>> class X:
...   val = None
...   @classproperty
...   def foo(cls):
...     return cls.val
...   @foo.setter
...   def foo(cls, val):
...     cls.val = val
>>> X.foo
>>> X.foo = 3
>>> X.foo
3
```

(continues on next page)

(continued from previous page)

```
>>> x = X()
>>> x.foo
3
>>> X.foo = 4
>>> x.foo
4
```

Note, because the metaclass was not specified, setting a value on an instance does not have the intended effect.

```
>>> x.foo = 5
>>> x.foo
5
>>> X.foo  # should be 5
4
>>> vars(x)  # should be empty
{'foo': 5}
>>> X().foo  # should be 5
4
```

set_password(*service*, *username*, *password*)

Write the password to the registry

keyrings.alt.Windows.**has_wincrypto**()

Does this environment have wincrypto? Should return False even when Mercurial's Demand Import allowed import of _win_crypto, so accesses an attribute of the module.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

K

M

O

P

R

S

V

Some radii options for box corners used; they were ignored as pict2e was not found