Sage Reference Manual: Cryptography

Release 8.3

The Sage Development Team

Aug 04, 2018

CONTENTS

CHAPTER

ONE

CRYPTOSYSTEMS

This module contains base classes for various cryptosystems, including symmetric key and public-key cryptosystems. The classes defined in this module should not be called directly. It is the responsibility of child classes to implement specific cryptosystems. Take for example the Hill or matrix cryptosystem as implemented in *HillCryptosystem*. It is a symmetric key cipher so *HillCryptosystem* is a child class of *SymmetricKeyCryptosystem*, which in turn is a child class of *Cryptosystem*. The following diagram shows the inheritance relationship of particular cryptosystems:

```
Cryptosystem
+ SymmetricKeyCryptosystem
| + HillCryptosystem
| + LFSRCryptosystem
| + ShiftCryptosystem
| + ShrinkingGeneratorCryptosystem
| + SubstitutionCryptosystem
| + TranspositionCryptosystem
| + VigenereCryptosystem
+ PublicKeyCryptosystem
```

class sage.crypto.cryptosystem.**Cryptosystem**(*plaintext_space*, *ciphertext_space*, *key_space*, *block_length=1*, *period=None*)

Bases: sage.structure.parent_old.Parent, sage.structure.parent.Set_generic

A base cryptosystem class. This is meant to be extended by other specialized child classes that implement specific cryptosystems. A cryptosystem is a pair of maps

$$E : \mathcal{K} \to \mathrm{Hom}(\mathcal{M}, \mathcal{C})$$

$$D : \mathcal{K} \to \mathrm{Hom}(\mathcal{C}, \mathcal{M})$$

where $\mathcal{K}$ is the key space, $\mathcal{M}$ is the plaintext or message space, and $\mathcal{C}$ is the ciphertext space. In many instances $\mathcal{M} = \mathcal{C}$ and the images will lie in $\mathrm{Aut}(\mathcal{M})$. An element of the image of E is called a cipher.

We may assume that E and D are injective, hence identify a key K in $\mathcal{K}$ with its image $E_K := E(K)$ in $\mathrm{Hom}(\mathcal{M}, \mathcal{C})$.

The cryptosystem has the property that for every encryption key K_1 there is a decryption key K_2 such that $D_{K_2} \circ E_{K_1}$. A cryptosystem with the property that $K := K_2 = K_1$, is called a symmetric cryptosystem. Otherwise, if the key $K_2 \neq K_1$, nor is K_2 easily derived from K_1, we call the cryptosystem asymmetric or public key. In that case, K_1 is called the public key and K_2 is called the private key.

INPUT:

- `plaintext_space` – the plaintext alphabet.
- `ciphertext_space` – the ciphertext alphabet.

- `key_space` – the key alphabet.
- `block_length` – (default: 1) the block length.
- `period` – (default: `None`) the period.

EXAMPLES:

Various classical cryptosystems:

```
sage: ShiftCryptosystem(AlphabeticStrings())
Shift cryptosystem on Free alphabetic string monoid on A-Z
sage: SubstitutionCryptosystem(HexadecimalStrings())
Substitution cryptosystem on Free hexadecimal string monoid
sage: HillCryptosystem(BinaryStrings(), 3)
Hill cryptosystem on Free binary string monoid of block length 3
sage: TranspositionCryptosystem(OctalStrings(), 5)
Transposition cryptosystem on Free octal string monoid of block length 5
sage: VigenereCryptosystem(Radix64Strings(), 7)
Vigenere cryptosystem on Free radix 64 string monoid of period 7
```

block_length()

Return the block length of this cryptosystem. For some cryptosystems this is not relevant, in which case the block length defaults to 1.

EXAMPLES:

The block lengths of various classical cryptosystems:

```
sage: ShiftCryptosystem(AlphabeticStrings()).block_length()
1
sage: SubstitutionCryptosystem(HexadecimalStrings()).block_length()
1
sage: HillCryptosystem(BinaryStrings(), 3).block_length()
3
sage: TranspositionCryptosystem(OctalStrings(), 5).block_length()
5
sage: VigenereCryptosystem(Radix64Strings(), 7).block_length()
1
```

cipher_codomain()

Return the alphabet used by this cryptosystem for encoding ciphertexts. This is the same as the ciphertext space.

EXAMPLES:

The cipher codomains, or ciphertext spaces, of various classical cryptosystems:

```
sage: ShiftCryptosystem(AlphabeticStrings()).cipher_codomain()
Free alphabetic string monoid on A-Z
sage: SubstitutionCryptosystem(HexadecimalStrings()).cipher_codomain()
Free hexadecimal string monoid
sage: HillCryptosystem(BinaryStrings(), 3).cipher_codomain()
Free binary string monoid
sage: TranspositionCryptosystem(OctalStrings(), 5).cipher_codomain()
Free octal string monoid
sage: VigenereCryptosystem(Radix64Strings(), 7).cipher_codomain()
Free radix 64 string monoid
```

cipher_domain()

Return the alphabet used by this cryptosystem for encoding plaintexts. This is the same as the plaintext

space.

EXAMPLES:

The cipher domains, or plaintext spaces, of various classical cryptosystems:

```
sage: ShiftCryptosystem(AlphabeticStrings()).cipher_domain()
Free alphabetic string monoid on A-Z
sage: SubstitutionCryptosystem(HexadecimalStrings()).cipher_domain()
Free hexadecimal string monoid
sage: HillCryptosystem(BinaryStrings(), 3).cipher_domain()
Free binary string monoid
sage: TranspositionCryptosystem(OctalStrings(), 5).cipher_domain()
Free octal string monoid
sage: VigenereCryptosystem(Radix64Strings(), 7).cipher_domain()
Free radix 64 string monoid
```

ciphertext_space()

Return the ciphertext alphabet of this cryptosystem.

EXAMPLES:

The ciphertext spaces of various classical cryptosystems:

```
sage: ShiftCryptosystem(AlphabeticStrings()).ciphertext_space()
Free alphabetic string monoid on A-Z
sage: SubstitutionCryptosystem(HexadecimalStrings()).ciphertext_space()
Free hexadecimal string monoid
sage: HillCryptosystem(BinaryStrings(), 3).ciphertext_space()
Free binary string monoid
sage: TranspositionCryptosystem(OctalStrings(), 5).ciphertext_space()
Free octal string monoid
sage: VigenereCryptosystem(Radix64Strings(), 7).ciphertext_space()
Free radix 64 string monoid
```

key_space()

Return the alphabet used by this cryptosystem for encoding keys.

EXAMPLES:

The key spaces of various classical cryptosystems:

```
sage: ShiftCryptosystem(AlphabeticStrings()).key_space()
Ring of integers modulo 26
sage: SubstitutionCryptosystem(HexadecimalStrings()).key_space()
Free hexadecimal string monoid
sage: HillCryptosystem(BinaryStrings(), 3).key_space()
Full MatrixSpace of 3 by 3 dense matrices over Ring of integers modulo 2
sage: TranspositionCryptosystem(OctalStrings(), 5).key_space()
Symmetric group of order 5! as a permutation group
sage: VigenereCryptosystem(Radix64Strings(), 7).key_space()
Free radix 64 string monoid
```

period()

plaintext_space()

Return the plaintext alphabet of this cryptosystem.

EXAMPLES:

The plaintext spaces of various classical cryptosystems:

```
sage: ShiftCryptosystem(AlphabeticStrings()).plaintext_space()
Free alphabetic string monoid on A-Z
sage: SubstitutionCryptosystem(HexadecimalStrings()).plaintext_space()
Free hexadecimal string monoid
sage: HillCryptosystem(BinaryStrings(), 3).plaintext_space()
Free binary string monoid
sage: TranspositionCryptosystem(OctalStrings(), 5).plaintext_space()
Free octal string monoid
sage: VigenereCryptosystem(Radix64Strings(), 7).plaintext_space()
Free radix 64 string monoid
```

class sage.crypto.cryptosystem.**PublicKeyCryptosystem**(*plaintext_space*, *ciphertext_space*, *key_space*, *block_length=1*, *period=None*)

Bases: *sage.crypto.cryptosystem.Cryptosystem*

The base class for asymmetric or public-key cryptosystems.

class sage.crypto.cryptosystem.**SymmetricKeyCryptosystem**(*plaintext_space*, *ciphertext_space*, *key_space*, *block_length=1*, *period=None*)

Bases: *sage.crypto.cryptosystem.Cryptosystem*

The base class for symmetric key, or secret key, cryptosystems.

alphabet_size()

Return the number of elements in the alphabet of this cryptosystem. This only applies to any cryptosystem whose plaintext and ciphertext spaces are the same alphabet.

EXAMPLES:

```
sage: ShiftCryptosystem(AlphabeticStrings()).alphabet_size()
26
sage: ShiftCryptosystem(BinaryStrings()).alphabet_size()
2
sage: ShiftCryptosystem(HexadecimalStrings()).alphabet_size()
16
sage: SubstitutionCryptosystem(OctalStrings()).alphabet_size()
8
sage: SubstitutionCryptosystem(Radix64Strings()).alphabet_size()
64
```

CHAPTER

TWO

CIPHERS

class sage.crypto.cipher.**Cipher**(*parent*, *key*)
Bases: sage.structure.element.Element

Cipher class

codomain()

domain()

key()

class sage.crypto.cipher.**PublicKeyCipher**(*parent*, *key*, *public=True*)
Bases: *sage.crypto.cipher.Cipher*

Public key cipher class

class sage.crypto.cipher.**SymmetricKeyCipher**(*parent*, *key*)
Bases: *sage.crypto.cipher.Cipher*

Symmetric key cipher class

CHAPTER

THREE

CLASSICAL CRYPTOSYSTEMS

A convenient user interface to various classical ciphers. These include:

- affine cipher; see *AffineCryptosystem*
- Hill or matrix cipher; see *HillCryptosystem*
- shift cipher; see *ShiftCryptosystem*
- substitution cipher; see *SubstitutionCryptosystem*
- transposition cipher; see *TranspositionCryptosystem*
- Vigenere cipher; see *VigenereCryptosystem*

These classical cryptosystems support alphabets such as:

- the capital letters of the English alphabet; see `AlphabeticStrings()`
- the hexadecimal number system; see `HexadecimalStrings()`
- the binary number system; see `BinaryStrings()`
- the octal number system; see `OctalStrings()`
- the radix-64 number system; see `Radix64Strings()`

AUTHORS:

- David Kohel (2007): initial version with the Hill, substitution, transposition, and Vigenere cryptosystems.
- Minh Van Nguyen (2009-08): shift cipher, affine cipher

class sage.crypto.classical.**AffineCryptosystem**(*A*)

Bases: *sage.crypto.cryptosystem.SymmetricKeyCryptosystem*

Create an affine cryptosystem.

Let $A = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$ be a non-empty alphabet consisting of n unique elements. Define a mapping $f : A \longrightarrow \mathbf{Z}/n\mathbf{Z}$ from the alphabet A to the set $\mathbf{Z}/n\mathbf{Z}$ of integers modulo n, given by $f(a_i) = i$. Thus we can identify each element of the alphabet A with a unique integer $0 \leq i < n$. A key of the affine cipher is an ordered pair of integers $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ such that $\gcd(a, n) = 1$. Therefore the key space is $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$. Since we assume that A does not have repeated elements, the mapping $f : A \longrightarrow \mathbf{Z}/n\mathbf{Z}$ is bijective. Encryption and decryption functions are both affine functions. Let (a, b) be a secret key, i.e. an element of the key space, and let p be a plaintext character and consequently $p \in \mathbf{Z}/n\mathbf{Z}$. Then the ciphertext character c corresponding to p is given by

$$c \equiv ap + b \pmod{n}$$

Similarly, given a ciphertext character $c \in \mathbf{Z}/n\mathbf{Z}$ and a secret key (a, b), we can recover the corresponding plaintext character as follows:

$$p \equiv a^{-1}(c - b) \pmod{n}$$

where a^{-1} is the inverse of a modulo n. Use the bijection $f : A \longrightarrow \mathbf{Z}/n\mathbf{Z}$ to convert c and p back to elements of the alphabet A. Currently, only the following alphabet is supported for the affine cipher:

- capital letters of the English alphabet as implemented in `AlphabeticStrings()`

EXAMPLES:

Encryption and decryption over the capital letters of the English alphabet:

```
sage: A = AffineCryptosystem(AlphabeticStrings()); A
Affine cryptosystem on Free alphabetic string monoid on A-Z
sage: P = A.encoding("The affine cryptosystem generalizes the shift cipher.")
sage: P
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: a, b = (9, 13)
sage: C = A.enciphering(a, b, P); C
CYXNGGHAXFKVSCJTVTCXRPXAXKNIHEXTCYXTYHGCFHSYXK
sage: A.deciphering(a, b, C)
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: A.deciphering(a, b, C) == P
True
```

We can also use functional notation to work through the previous example:

```
sage: A = AffineCryptosystem(AlphabeticStrings()); A
Affine cryptosystem on Free alphabetic string monoid on A-Z
sage: P = A.encoding("The affine cryptosystem generalizes the shift cipher.")
sage: P
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: a, b = (9, 13)
sage: E = A(a, b); E
Affine cipher on Free alphabetic string monoid on A-Z
sage: C = E(P); C
CYXNGGHAXFKVSCJTVTCXRPXAXKNIHEXTCYXTYHGCFHSYXK
sage: aInv, bInv = A.inverse_key(a, b)
sage: D = A(aInv, bInv); D
Affine cipher on Free alphabetic string monoid on A-Z
sage: D(C)
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: D(C) == P
True
sage: D(C) == P == D(E(P))
True
```

Encrypting the ciphertext with the inverse key also produces the plaintext:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("Encrypt with inverse key.")
sage: a, b = (11, 8)
sage: C = A.enciphering(a, b, P)
sage: P; C
ENCRYPTWITHINVERSEKEY
AVENMRJQSJHSVFANYAOAM
sage: aInv, bInv = A.inverse_key(a, b)
```

(continues on next page)

(continued from previous page)

```
sage: A.enciphering(aInv, bInv, C)
ENCRYPTWITHINVERSEKEY
sage: A.enciphering(aInv, bInv, C) == P
True
```

For a secret key $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$, if $a = 1$ then any affine cryptosystem with key $(1, b)$ for any $b \in \mathbf{Z}/n\mathbf{Z}$ is a shift cryptosystem. Here is how we can create a Caesar cipher using an affine cipher:

```
sage: caesar = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (1, 3)
sage: P = caesar.encoding("abcdef"); P
ABCDEF
sage: C = caesar.enciphering(a, b, P); C
DEFGHI
sage: caesar.deciphering(a, b, C) == P
True
```

Any affine cipher with keys of the form $(a, 0) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ is called a decimation cipher on the Roman alphabet, or decimation cipher for short:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("A decimation cipher is a specialized affine cipher.")
sage: a, b = (17, 0)
sage: C = A.enciphering(a, b, P)
sage: P; C
ADECIMATIONCIPHERISASPECIALIZEDAFFINECIPHER
AZQIGWALGENIGVPQDGUAUVQIGAFGJQZAHHGNQIGVPQD
sage: A.deciphering(a, b, C) == P
True
```

Generate a random key for encryption and decryption:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("An affine cipher with a random key.")
sage: a, b = A.random_key()
sage: C = A.enciphering(a, b, P)
sage: A.deciphering(a, b, C) == P
True
```

REFERENCES:

- [Sti2006]

brute_force(*C*, *ranking='none'*)

Attempt a brute force cryptanalysis of the ciphertext `C`.

INPUT:

- `C` – A ciphertext over one of the supported alphabets of this affine cryptosystem. See the class *AffineCryptosystem* for documentation on the supported alphabets.
- `ranking` – (default `"none"`) the method to use for ranking all possible keys. If `ranking="none"`, then do not use any ranking function. The following ranking functions are supported:
  - `"chi_square"` – the chi-square ranking function as implemented in the method *rank_by_chi_square()*.

– "squared_differences" – the squared differences ranking function as implemented in the method *rank_by_squared_differences()*.

OUTPUT:

- All the possible plaintext sequences corresponding to the ciphertext C. This method effectively uses all the possible keys in this affine cryptosystem to decrypt C. The method is also referred to as exhaustive key search. The output is a dictionary of key, candidate decipherment pairs.

EXAMPLES:

Cryptanalyze using all possible keys with the option ranking="none":

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Linear"); P
LINEAR
sage: C = A.enciphering(a, b, P)
sage: L = A.brute_force(C)
sage: sorted(L.items())[:26]  # display 26 candidate decipherments

[((1, 0), OFUTHG),
((1, 1), NETSGF),
((1, 2), MDSRFE),
((1, 3), LCRQED),
((1, 4), KBQPDC),
((1, 5), JAPOCB),
((1, 6), IZONBA),
((1, 7), HYNMAZ),
((1, 8), GXMLZY),
((1, 9), FWLKYX),
((1, 10), EVKJXW),
((1, 11), DUJIWV),
((1, 12), CTIHVU),
((1, 13), BSHGUT),
((1, 14), ARGFTS),
((1, 15), ZQFESR),
((1, 16), YPEDRQ),
((1, 17), XODCQP),
((1, 18), WNCBPO),
((1, 19), VMBAON),
((1, 20), ULAZNM),
((1, 21), TKZYML),
((1, 22), SJYXLK),
((1, 23), RIXWKJ),
((1, 24), QHWVJI),
((1, 25), PGVUIH)]
```

Use the chi-square ranking function, i.e. ranking="chisquare":

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Linear functions for encrypting and decrypting."); P
LINEARFUNCTIONSFORENCRYPTINGANDDECRYPTING
sage: C = A.enciphering(a, b, P)
sage: Rank = A.brute_force(C, ranking="chisquare")
sage: Rank[:10]  # display only the top 10 candidate keys

[((3, 7), LINEARFUNCTIONSFORENCRYPTINGANDDECRYPTING),
```

(continues on next page)

(continued from previous page)

```
((23, 25), VYTCGPBMTENYSTOBSPCTEPIRNYTAGTDDCEPIRNYTA),
((1, 12), CTIHVUKDIBATLIXKLUHIBUPOATINVIEEHBUPOATIN),
((11, 15), HSRYELDAROVSWRQDWLYROLUBVSRIERTTYOLUBVSRI),
((25, 1), NWHIUVFMHOPWEHSFEVIHOVABPWHCUHLLIOVABPWHC),
((25, 7), TCNOABLSNUVCKNYLKBONUBGHVCNIANRROUBGHVCNI),
((15, 4), SHIBVOWZILEHDIJWDOBILOFYEHIRVIGGBLOFYEHIR),
((15, 23), PEFYSLTWFIBEAFGTALYFILCVBEFOSFDDYILCVBEFO),
((7, 10), IDUFHSYXUTEDNULYNSFUTSVGEDURHUMMFTSVGEDUR),
((19, 22), QVETRGABEFUVLENALGTEFGDSUVEHREMMTFGDSUVEH)]
```

Use the squared differences ranking function, i.e. `ranking="squared_differences"`:

```
sage: Rank = A.brute_force(C, ranking="squared_differences")
sage: Rank[:10]  # display only the top 10 candidate keys

[((3, 7), LINEARFUNCTIONSFORENCRYPTINGANDDECRYPTING),
((23, 6), GJENRAMXEPYJDEZMDANEPATCYJELREOONPATCYJEL),
((23, 25), VYTCGPBMTENYSTOBSPCTEPIRNYTAGTDDCEPIRNYTA),
((19, 22), QVETRGABEFUVLENALGTEFGDSUVEHREMMTFGDSUVEH),
((19, 9), DIRGETNORSHIYRANYTGRSTQFHIRUERZZGSTQFHIRU),
((23, 18), KNIRVEQBITCNHIDQHERITEXGCNIPVISSRTEXGCNIP),
((17, 16), GHORBEIDOJMHFOVIFEROJETWMHOZBOAARJETWMHOZ),
((21, 14), AHEZRMOFEVQHTEBOTMZEVMNIQHEDREKKZVMNIQHED),
((1, 12), CTIHVUKDIBATLIXKLUHIBUPOATINVIEEHBUPOATIN),
((7, 18), SNEPRCIHEDONXEVIXCPEDCFQONEBREWWPDCFQONEB)]
```

deciphering(*a*, *b*, *C*)

Decrypt the ciphertext `C` with the key `(a, b)` using affine cipher decryption.

INPUT:

- `a, b` – a secret key belonging to the key space of this affine cipher. This key must be an element of $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ such that $\gcd(a, n) = 1$ with n being the size of the ciphertext and plaintext spaces.
- `C` – a string of ciphertext; possibly an empty string. Characters in this string must be encoded using one of the supported alphabets. See the method *`encoding()`* for more information.

OUTPUT:

- The plaintext corresponding to the ciphertext `C`.

EXAMPLES:

Decryption over the capital letters of the English alphabet:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (5, 2)
sage: P = A.encoding("Affine functions are linear functions.")
sage: C = A.enciphering(a, b, P); C
CBBQPWBYPMTQUPOCJWFQPWCJBYPMTQUPO
sage: P == A.deciphering(a, b, C)
True
```

The previous example can also be worked through using functional notation:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (5, 2)
sage: P = A.encoding("Affine functions are linear functions.")
sage: E = A(a, b); E
```

(continues on next page)

(continued from previous page)

```
Affine cipher on Free alphabetic string monoid on A-Z
sage: C = E(P); C
CBBQPWBYPMTQUPOCJWFQPWCJBYPMTQUPO
sage: aInv, bInv = A.inverse_key(a, b)
sage: D = A(aInv, bInv); D
Affine cipher on Free alphabetic string monoid on A-Z
sage: D(C) == P
True
```

If the ciphertext is an empty string, then the plaintext is also an empty string regardless of the value of the secret key:

```
sage: a, b = A.random_key()
sage: A.deciphering(a, b, A.encoding(""))

sage: A.deciphering(a, b, A.encoding(" "))
```

enciphering(*a*, *b*, *P*)

Encrypt the plaintext `P` with the key `(a, b)` using affine cipher encryption.

INPUT:

- `a, b` – a secret key belonging to the key space of this affine cipher. This key must be an element of $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ such that $\gcd(a, n) = 1$ with n being the size of the ciphertext and plaintext spaces.
- `P` – a string of plaintext; possibly an empty string. Characters in this string must be encoded using one of the supported alphabets. See the method *`encoding()`* for more information.

OUTPUT:

- The ciphertext corresponding to the plaintext `P`.

EXAMPLES:

Encryption over the capital letters of the English alphabet:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 6)
sage: P = A.encoding("Affine ciphers work with linear functions.")
sage: A.enciphering(a, b, P)
GVVETSMEZBSFIUWFKUELBNETSGFVOTMLEWTI
```

Now work through the previous example using functional notation:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 6)
sage: P = A.encoding("Affine ciphers work with linear functions.")
sage: E = A(a, b); E
Affine cipher on Free alphabetic string monoid on A-Z
sage: E(P)
GVVETSMEZBSFIUWFKUELBNETSGFVOTMLEWTI
```

If the plaintext is an empty string, then the ciphertext is also an empty string regardless of the value of the secret key:

```
sage: a, b = A.random_key()
sage: A.enciphering(a, b, A.encoding(""))

sage: A.enciphering(a, b, A.encoding(" "))
```

encoding(S)

The encoding of the string S over the string monoid of this affine cipher. For example, if the string monoid of this cryptosystem is AlphabeticStringMonoid, then the encoding of S would be its upper-case equivalent stripped of all non-alphabetic characters. Only the following alphabet is supported for the affine cipher:

- capital letters of the English alphabet as implemented in AlphabeticStrings()

INPUT:

- S – a string, possibly empty.

OUTPUT:

- The encoding of S over the string monoid of this cryptosystem. If S is an empty string, return an empty string.

EXAMPLES:

Encoding over the upper-case letters of the English alphabet:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: A.encoding("Affine cipher over capital letters of the English alphabet.
↪")
AFFINECIPHEROVERCAPITALLETTERSOFTHEENGLISHALPHABET
```

The argument S can be an empty string, in which case an empty string is returned:

```
sage: AffineCryptosystem(AlphabeticStrings()).encoding("")
```

inverse_key(a, b)

The inverse key corresponding to the secret key (a, b). If p is a plaintext character so that $p \in \mathbf{Z}/n\mathbf{Z}$ and n is the alphabet size, then the ciphertext c corresponding to p is

$$c \equiv ap + b \pmod{n}$$

As (a, b) is a key, then the multiplicative inverse a^{-1} exists and the original plaintext can be recovered as follows

$$p \equiv a^{-1}(c - b) \pmod{n} \equiv a^{-1}c + a^{-1}(-b) \pmod{n}$$

Therefore the ordered pair $(a^{-1}, -ba^{-1})$ is the inverse key corresponding to (a, b).

INPUT:

- a, b – a secret key for this affine cipher. The ordered pair (a, b) must be an element of $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ such that $\gcd(a, n) = 1$.

OUTPUT:

- The inverse key $(a^{-1}, -ba^{-1})$ corresponding to (a, b).

EXAMPLES:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (1, 2)
sage: A.inverse_key(a, b)
(1, 24)
sage: A.inverse_key(3, 2)
(9, 8)
```

Suppose that the plaintext and ciphertext spaces are the capital letters of the English alphabet so that $n = 26$. If $\varphi(n)$ is the Euler phi function of n, then there are $\varphi(n)$ integers $0 \leq a < n$ that are relatively prime to n. For the capital letters of the English alphabet, there are 12 such integers relatively prime to n:

```
sage: euler_phi(A.alphabet_size())
12
```

And here is a list of those integers:

```
sage: n = A.alphabet_size()
sage: L = [i for i in range(n) if gcd(i, n) == 1]; L
[1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25]
```

Then a secret key (a, b) of this shift cryptosystem is such that a is an element of the list `L` in the last example. Any inverse key (A, B) corresponding to (a, b) is such that A is also in the list `L` above:

```
sage: a, b = (3, 9)
sage: a in L
True
sage: aInv, bInv = A.inverse_key(a, b)
sage: aInv, bInv
(9, 23)
sage: aInv in L
True
```

random_key()

Generate a random key within the key space of this affine cipher. The generated secret key is an ordered pair $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ with n being the size of the cipher domain and $\gcd(a, n) = 1$. Let $\varphi(n)$ denote the Euler phi function of n. Then the affine cipher has $n \cdot \varphi(n)$ possible keys (see page 10 of [Sti2006]).

OUTPUT:

- A random key within the key space of this affine cryptosystem. The output key is an ordered pair (a, b).

EXAMPLES:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: A.random_key()  # random
(17, 25)
```

If (a, b) is a secret key and n is the size of the plaintext and ciphertext alphabets, then $\gcd(a, n) = 1$:

```
sage: a, b = A.random_key()
sage: n = A.alphabet_size()
sage: gcd(a, n)
1
```

rank_by_chi_square(*C*, *pdict*)

Use the chi-square statistic to rank all possible keys. Currently, this method only applies to the capital letters of the English alphabet.

ALGORITHM:

Consider a non-empty alphabet A consisting of n elements, and let C be a ciphertext encoded using elements of A. The plaintext P corresponding to C is also encoded using elements of A. Let M be a candidate decipherment of C, i.e. M is the result of attempting to decrypt C using a key (a, b) which is not necessarily the same key used to encrypt P. Suppose $F_A(e)$ is the characteristic frequency probability of

$e \in A$ and let $F_M(e)$ be the message frequency probability with respect to M. The characteristic frequency probability distribution of an alphabet is the expected frequency probability distribution for that alphabet. The message frequency probability distribution of M provides a distribution of the ratio of character occurrences over message length. One can interpret the characteristic frequency probability $F_A(e)$ as the expected probability, while the message frequency probability $F_M(e)$ is the observed probability. If M is of length L, then the observed frequency of $e \in A$ is

$$O_M(e) = F_M(e) \cdot L$$

and the expected frequency of $e \in A$ is

$$E_A(e) = F_A(e) \cdot L$$

The chi-square rank $R_{\chi^2}(M)$ of M corresponding to a key $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{\chi^2}(M) = \sum_{e \in A} \frac{\big(O_M(e) - E_A(e)\big)^2}{E_A(e)}$$

Cryptanalysis by exhaustive key search produces a candidate decipherment $M_{a,b}$ for each possible key (a, b). For a set $D = \{M_{a_1,b_1}, M_{a_2,b_2}, \ldots, M_{a_k,b_k}\}$ of all candidate decipherments corresponding to a ciphertext C, the smaller is the rank $R_{\chi^2}(M_{a_i,b_i})$ the more likely that (a_i, b_i) is the secret key. This key ranking method is based on the Pearson chi-square test [PearsonTest].

INPUT:

- `C` – The ciphertext, a non-empty string. The ciphertext must be encoded using the upper-case letters of the English alphabet.
- `pdict` – A dictionary of key, possible plaintext pairs. This should be the output of *`brute_force()`* with `ranking="none"`.

OUTPUT:

- A list ranking the most likely keys first. Each element of the list is a tuple of key, possible plaintext pairs.

EXAMPLES:

Use the chi-square statistic to rank all possible keys and their corresponding decipherment:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Line.")
sage: C = A.enciphering(a, b, P)
sage: Plist = A.brute_force(C)
sage: Rank = A.rank_by_chi_square(C, Plist)
sage: Rank[:10]  # display only the top 10 candidate keys

[((1, 1), NETS),
((3, 7), LINE),
((17, 20), STAD),
((5, 2), SLOT),
((5, 5), HADI),
((9, 25), TSLI),
((17, 15), DELO),
((15, 6), ETUN),
((21, 8), ELID),
((7, 17), HCTE)]
```

As more ciphertext is available, the reliability of the chi-square ranking function increases:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (11, 24)
sage: P = A.encoding("Longer message is more information for cryptanalysis.")
sage: C = A.enciphering(a, b, P)
sage: Plist = A.brute_force(C)
sage: Rank = A.rank_by_chi_square(C, Plist)
sage: Rank[:10]  # display only the top 10 candidate keys

[((11, 24), LONGERMESSAGEISMOREINFORMATIONFORCRYPTANALYSIS),
((17, 9), INURFSBFLLHRFDLBNSFDUYNSBHEDNUYNSTSVGEHUHIVLDL),
((9, 18), RMFIUHYUOOSIUWOYMHUWFBMHYSVWMFBMHGHETVSFSREOWO),
((15, 12), VSTACPUCOOGACYOUSPCYTBSPUGNYSTBSPEPIRNGTGVIOYO),
((3, 22), PAFOYLKYGGSOYEGKALYEFTALKSBEAFTALILCVBSFSPCGEG),
((25, 3), OHSRNADNPPFRNVPDHANVSCHADFEVHSCHAJABWEFSFOBPVP),
((7, 25), GHYNVIPVRRLNVFRPHIVFYEHIPLAFHYEHIDITQALYLGTRFR),
((5, 2), NEHCIVKISSUCIWSKEVIWHFEVKUPWEHFEVOVABPUHUNASWS),
((15, 25), IFGNPCHPBBTNPLBHFCPLGOFCHTALFGOFCRCVEATGTIVBLB),
((9, 6), BWPSERIEYYCSEGYIWREGPLWRICFGWPLWRQRODFCPCBOYGY)]
```

rank_by_squared_differences(*C*, *pdict*)

Use the squared-differences measure to rank all possible keys. Currently, this method only applies to the capital letters of the English alphabet.

ALGORITHM:

Consider a non-empty alphabet A consisting of n elements, and let C be a ciphertext encoded using elements of A. The plaintext P corresponding to C is also encoded using elements of A. Let M be a candidate decipherment of C, i.e. M is the result of attempting to decrypt C using a key (a, b) which is not necessarily the same key used to encrypt P. Suppose $F_A(e)$ is the characteristic frequency probability of $e \in A$ and let $F_M(e)$ be the message frequency probability with respect to M. The characteristic frequency probability distribution of an alphabet is the expected frequency probability distribution for that alphabet. The message frequency probability distribution of M provides a distribution of the ratio of character occurrences over message length. One can interpret the characteristic frequency probability $F_A(e)$ as the expected probability, while the message frequency probability $F_M(e)$ is the observed probability. If M is of length L, then the observed frequency of $e \in A$ is

$$O_M(e) = F_M(e) \cdot L$$

and the expected frequency of $e \in A$ is

$$E_A(e) = F_A(e) \cdot L$$

The squared-differences, or residual sum of squares, rank $R_{RSS}(M)$ of M corresponding to a key $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{RSS}(M) = \sum_{e \in A} \big(O_M(e) - E_A(e)\big)^2$$

Cryptanalysis by exhaustive key search produces a candidate decipherment $M_{a,b}$ for each possible key (a, b). For a set $D = \left\{M_{a_1,b_1}, M_{a_2,b_2}, \ldots, M_{a_k,b_k}\right\}$ of all candidate decipherments corresponding to a ciphertext C, the smaller is the rank $R_{RSS}(M_{a_i,b_i})$ the more likely that (a_i, b_i) is the secret key. This key ranking method is based on the residual sum of squares measure [RSS].

INPUT:

- `C` – The ciphertext, a non-empty string. The ciphertext must be encoded using the upper-case letters of the English alphabet.

- pdict – A dictionary of key, possible plaintext pairs. This should be the output of *brute_force()* with ranking="none".

OUTPUT:

- A list ranking the most likely keys first. Each element of the list is a tuple of key, possible plaintext pairs.

EXAMPLES:

Use the method of squared differences to rank all possible keys and their corresponding decipherment:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Line.")
sage: C = A.enciphering(a, b, P)
sage: Plist = A.brute_force(C)
sage: Rank = A.rank_by_squared_differences(C, Plist)
sage: Rank[:10]  # display only the top 10 candidate keys

[((1, 1), NETS),
((15, 6), ETUN),
((7, 17), HCTE),
((3, 7), LINE),
((17, 15), DELO),
((9, 4), EDWT),
((9, 9), POHE),
((21, 8), ELID),
((17, 20), STAD),
((7, 18), SNEP)]
```

As more ciphertext is available, the reliability of the squared-differences ranking function increases:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (11, 24)
sage: P = A.encoding("Longer message is more information for cryptanalysis.")
sage: C = A.enciphering(a, b, P)
sage: Plist = A.brute_force(C)
sage: Rank = A.rank_by_squared_differences(C, Plist)
sage: Rank[:10]  # display only the top 10 candidate keys

[((11, 24), LONGERMESSAGEISMOREINFORMATIONFORCRYPTANALYSIS),
((9, 14), DYRUGTKGAAEUGIAKYTGIRNYTKEHIYRNYTSTQFHEREDQAIA),
((23, 24), DSNEUHIUMMAEUOMISHUONZSHIAROSNZSHKHQXRANADQMOM),
((23, 1), ETOFVIJVNNBFVPNJTIVPOATIJBSPTOATILIRYSBOBERNPN),
((21, 16), VEBGANYAQQOGAMQYENAMBDENYOTMEBDENUNIHTOBOVIQMQ),
((7, 12), TULAIVCIEEYAISECUVISLRUVCYNSULRUVQVGDNYLYTGESE),
((5, 20), ZQTOUHWUEEGOUIEWQHUITRQHWGBIQTRQHAHMNBGTGZMEIE),
((21, 8), JSPUOBMOEECUOAEMSBOAPRSBMCHASPRSBIBWVHCPCJWEAE),
((25, 7), SLWVREHRTTJVRZTHLERZWGLEHJIZLWGLENEFAIJWJSFTZT),
((25, 15), ATEDZMPZBBRDZHBPTMZHEOTMPRQHTEOTMVMNIQRERANBHB)]
```

class sage.crypto.classical.**HillCryptosystem**(S, m)

Bases: *sage.crypto.cryptosystem.SymmetricKeyCryptosystem*

Create a Hill cryptosystem defined by the m x m matrix space over $\mathbf{Z}/N\mathbf{Z}$, where N is the alphabet size of the string monoid S.

INPUT:

- S - a string monoid over some alphabet

- m - integer > 0; the block length of matrices that specify block permutations

OUTPUT:

- A Hill cryptosystem of block length m over the alphabet S.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = HillCryptosystem(S,3)
sage: E
Hill cryptosystem on Free alphabetic string monoid on A-Z of block length 3
sage: R = IntegerModRing(26)
sage: M = MatrixSpace(R,3,3)
sage: A = M([[1,0,1],[0,1,1],[2,2,3]])
sage: A
[1 0 1]
[0 1 1]
[2 2 3]
sage: e = E(A)
sage: e
Hill cipher on Free alphabetic string monoid on A-Z of block length 3
sage: e(S("LAMAISONBLANCHE"))
JYVKSKQPELAYKPV
```

block_length()

The row or column dimension of a matrix specifying a block permutation. Encryption and decryption keys of a Hill cipher are square matrices, i.e. the row and column dimensions of an encryption or decryption key are the same. This row/column dimension is referred to as the *block length*.

OUTPUT:

- The block length of an encryption/decryption key.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: n = randint(1, A.ngens() - 1)
sage: H = HillCryptosystem(A, n)
sage: H.block_length() == n
True
```

deciphering(*A*, *C*)

Decrypt the ciphertext C using the key A.

INPUT:

- A - a key within the key space of this Hill cipher
- C - a string (possibly empty) over the string monoid of this Hill cipher

OUTPUT:

- The plaintext corresponding to the ciphertext C.

EXAMPLES:

```
sage: H = HillCryptosystem(AlphabeticStrings(), 3)
sage: K = H.random_key()
sage: M = H.encoding("Good day, mate! How ya going?")
sage: H.deciphering(K, H.enciphering(K, M)) == M
True
```

enciphering(*A*, *M*)

Encrypt the plaintext `M` using the key `A`.

INPUT:

- `A` - a key within the key space of this Hill cipher
- `M` - a string (possibly empty) over the string monoid of this Hill cipher.

OUTPUT:

- The ciphertext corresponding to the plaintext `M`.

EXAMPLES:

```
sage: H = HillCryptosystem(AlphabeticStrings(), 3)
sage: K = H.random_key()
sage: M = H.encoding("Good day, mate! How ya going?")
sage: H.deciphering(K, H.enciphering(K, M)) == M
True
```

encoding(*M*)

The encoding of the string `M` over the string monoid of this Hill cipher. For example, if the string monoid of this Hill cipher is `AlphabeticStringMonoid`, then the encoding of `M` would be its upper-case equivalent stripped of all non-alphabetic characters.

INPUT:

- `M` - a string, possibly empty

OUTPUT:

- The encoding of `M` over the string monoid of this Hill cipher.

EXAMPLES:

```
sage: M = "The matrix cipher by Lester S. Hill."
sage: A = AlphabeticStrings()
sage: H = HillCryptosystem(A, 7)
sage: H.encoding(M) == A.encoding(M)
True
```

inverse_key(*A*)

The inverse key corresponding to the key `A`.

INPUT:

- `A` - an invertible matrix of the key space of this Hill cipher

OUTPUT:

- The inverse matrix of `A`.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = HillCryptosystem(S,3)
sage: A = E.random_key()
sage: B = E.inverse_key(A)
sage: M = S("LAMAISONBLANCHE")
sage: e = E(A)
sage: c = E(B)
sage: c(e(M))
LAMAISONBLANCHE
```

random_key()

A random key within the key space of this Hill cipher. That is, generate a random m x m matrix to be used as a block permutation, where m is the block length of this Hill cipher. If n is the size of the cryptosystem alphabet, then there are n^{m^2} possible keys. However the number of valid keys, i.e. invertible m x m square matrices, is smaller than n^{m^2}.

OUTPUT:

- A random key within the key space of this Hill cipher.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: n = 3
sage: H = HillCryptosystem(A, n)
sage: K = H.random_key()
sage: Ki = H.inverse_key(K)
sage: M = "LAMAISONBLANCHE"
sage: e = H(K)
sage: d = H(Ki)
sage: d(e(A(M))) == A(M)
True
```

class sage.crypto.classical.**ShiftCryptosystem**(A)

Bases: *sage.crypto.cryptosystem.SymmetricKeyCryptosystem*

Create a shift cryptosystem.

Let $A = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$ be a non-empty alphabet consisting of n unique elements. Define a mapping $f : A \longrightarrow \mathbf{Z}/n\mathbf{Z}$ from the alphabet A to the set $\mathbf{Z}/n\mathbf{Z}$ of integers modulo n, given by $f(a_i) = i$. Thus we can identify each element of the alphabet A with a unique integer $0 \leq i < n$. A key of the shift cipher is an integer $0 \leq k < n$. Therefore the key space is $\mathbf{Z}/n\mathbf{Z}$. Since we assume that A does not have repeated elements, the mapping $f : A \longrightarrow \mathbf{Z}/n\mathbf{Z}$ is bijective. Encryption works by moving along the alphabet by k positions, with wrap around. Decryption reverses the process by moving backwards by k positions, with wrap around. More generally, let k be a secret key, i.e. an element of the key space, and let p be a plaintext character and consequently $p \in \mathbf{Z}/n\mathbf{Z}$. Then the ciphertext character c corresponding to p is given by

$$c \equiv p + k \pmod{n}$$

Similarly, given a ciphertext character $c \in \mathbf{Z}/n\mathbf{Z}$ and a secret key k, we can recover the corresponding plaintext character as follows:

$$p \equiv c - k \pmod{n}$$

Use the bijection $f : A \longrightarrow \mathbf{Z}/n\mathbf{Z}$ to convert c and p back to elements of the alphabet A. Currently, the following alphabets are supported for the shift cipher:

- capital letters of the English alphabet as implemented in `AlphabeticStrings()`
- the alphabet consisting of the hexadecimal number system as implemented in `HexadecimalStrings()`
- the alphabet consisting of the binary number system as implemented in `BinaryStrings()`

EXAMPLES:

Some examples illustrating encryption and decryption over various alphabets. Here is an example over the upper-case letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings()); S
Shift cryptosystem on Free alphabetic string monoid on A-Z
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: P
THESHIFTCRYPTOSYSTEMGENERALIZESTHECAESARCIPHER
sage: K = 7
sage: C = S.enciphering(K, P); C
AOLZOPMAJYFWAVZFZALTNLULYHSPGLZAOLJHLZHYJPWOLY
sage: S.deciphering(K, C)
THESHIFTCRYPTOSYSTEMGENERALIZESTHECAESARCIPHER
sage: S.deciphering(K, C) == P
True
```

The previous example can also be done as follows:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: K = 7
sage: E = S(K); E
Shift cipher on Free alphabetic string monoid on A-Z
sage: C = E(P); C
AOLZOPMAJYFWAVZFZALTNLULYHSPGLZAOLJHLZHYJPWOLY
sage: D = S(S.inverse_key(K)); D
Shift cipher on Free alphabetic string monoid on A-Z
sage: D(C) == P
True
sage: D(C) == P == D(E(P))
True
```

Over the hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings()); S
Shift cryptosystem on Free hexadecimal string monoid
sage: P = S.encoding("Encryption & decryption shifts along the alphabet."); P
456e6372797074696f6e2026206465637279707469 6f6e207368696674732061 6c6f6e672074686520616c706861626
sage: K = 5
sage: C = S.enciphering(K, P); C
9ab3b8c7cec5c9beb4b3757b75b9bab8c7cec5c9beb4b375c8bdbebbc9c875b6b1b4b3bc75c9bdba75b6b1c5bdb6b7ba
sage: S.deciphering(K, C)
456e6372797074696f6e2026206465637279707469 6f6e207368696674732061 6c6f6e672074686520616c706861626
sage: S.deciphering(K, C) == P
True
```

And over the binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings()); S
Shift cryptosystem on Free binary string monoid
sage: P = S.encoding("The binary alphabet is very insecure."); P
010101000110100001100101001000000110001001101001011011100110000101110010011110010010000001100001
sage: K = 1
sage: C = S.enciphering(K, P); C
101010111001011110011010110111111001110110010110100100011001111010001101100001101101111110011110
sage: S.deciphering(K, C)
010101000110100001100101001000000110001001101001011011100110000101110010011110010010000001100001
sage: S.deciphering(K, C) == P
True
```

A shift cryptosystem with key $k = 3$ is commonly referred to as the Caesar cipher. Create a Caesar cipher over

the upper-case letters of the English alphabet:

```
sage: caesar = ShiftCryptosystem(AlphabeticStrings())
sage: K = 3
sage: P = caesar.encoding("abcdef"); P
ABCDEF
sage: C = caesar.enciphering(K, P); C
DEFGHI
sage: caesar.deciphering(K, C) == P
True
```

Generate a random key for encryption and decryption:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shift cipher with a random key.")
sage: K = S.random_key()
sage: C = S.enciphering(K, P)
sage: S.deciphering(K, C) == P
True
```

Decrypting with the key `K` is equivalent to encrypting with its corresponding inverse key:

```
sage: S.enciphering(S.inverse_key(K), C) == P
True
```

brute_force(*C*, *ranking='none'*)

Attempt a brute force cryptanalysis of the ciphertext `C`.

INPUT:

- `C` – A ciphertext over one of the supported alphabets of this shift cryptosystem. See the class *ShiftCryptosystem* for documentation on the supported alphabets.
- `ranking` – (default `"none"`) the method to use for ranking all possible keys. If `ranking="none"`, then do not use any ranking function. The following ranking functions are supported:
  - `"chisquare"` – the chi-square ranking function as implemented in the method *rank_by_chi_square()*.
  - `"squared_differences"` – the squared differences ranking function as implemented in the method *rank_by_squared_differences()*.

OUTPUT:

- All the possible plaintext sequences corresponding to the ciphertext `C`. This method effectively uses all the possible keys in this shift cryptosystem to decrypt `C`. The method is also referred to as exhaustive key search. The output is a dictionary of key, plaintext pairs.

EXAMPLES:

Cryptanalyze using all possible keys for various alphabets. Over the upper-case letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: K = 7
sage: C = S.enciphering(K, P)
sage: Dict = S.brute_force(C)
sage: for k in range(len(Dict)):
```

(continues on next page)

(continued from previous page)

```
....:     if Dict[k] == P:
....:         print("key = " + str(k))
key = 7
```

Over the hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: P = S.encoding("Encryption & decryption shifts along the alphabet.")
sage: K = 5
sage: C = S.enciphering(K, P)
sage: Dict = S.brute_force(C)
sage: for k in range(len(Dict)):
....:     if Dict[k] == P:
....:         print("key = " + str(k))
key = 5
```

And over the binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: P = S.encoding("The binary alphabet is very insecure.")
sage: K = 1
sage: C = S.enciphering(K, P)
sage: Dict = S.brute_force(C)
sage: for k in range(len(Dict)):
....:     if Dict[k] == P:
....:         print("key = " + str(k))
key = 1
```

Don't use any ranking functions, i.e. `ranking="none"`:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shifting using modular arithmetic.")
sage: K = 8
sage: C = S.enciphering(K, P)
sage: pdict = S.brute_force(C)
sage: sorted(pdict.items())

[(0, APQNBQVOCAQVOUWLCTIZIZQBPUMBQK),
(1, ZOPMAPUNBZPUNTVKBSHYHYPAOTLAPJ),
(2, YNOLZOTMAYOTMSUJARGXGXOZNSKZOI),
(3, XMNKYNSLZXNSLRTIZQFWFWNYMRJYNH),
(4, WLMJXMRKYWMRKQSHYPEVEVMXLQIXMG),
(5, VKLIWLQJXVLQJPRGXODUDULWKPHWLF),
(6, UJKHVKPIWUKPIOQFWNCTCTKVJOGVKE),
(7, TIJGUJOHVTJOHNPEVMBSBSJUINFUJD),
(8, SHIFTINGUSINGMODULARARITHMETIC),
(9, RGHESHMFTRHMFLNCTKZQZQHSGLDSHB),
(10, QFGDRGLESQGLEKMBSJYPYPGRFKCRGA),
(11, PEFCQFKDRPFKDJLARIXOXOFQEJBQFZ),
(12, ODEBPEJCQOEJCIKZQHWNWNEPDIAPEY),
(13, NCDAODIBPNDIBHJYPGVMVMDOCHZODX),
(14, MBCZNCHAOMCHAGIXOFULULCNBGYNCW),
(15, LABYMBGZNLBGZFHWNETKTKBMAFXMBV),
(16, KZAXLAFYMKAFYEGVMDSJSJALZEWLAU),
(17, JYZWKZEXLJZEXDFULCRIRIZKYDVKZT),
(18, IXYVJYDWKIYDWCETKBQHQHYJXCUJYS),
(19, HWXUIXCVJHXCVBDSJAPGPGXIWBTIXR),
```

(continues on next page)

(continued from previous page)

```
(20, GVWTHWBUIGWBUACRIZOFOFWHVASHWQ),
(21, FUVSGVATHFVATZBQHYNENEVGUZRGVP),
(22, ETURFUZSGEUZSYAPGXMDMDUFTYQFUO),
(23, DSTQETYRFDTYRXZOFWLCLCTESXPETN),
(24, CRSPDSXQECSXQWYNEVKBKBSDRWODSM),
(25, BQROCRWPDBRWPVXMDUJAJARCQVNCRL)]
```

Use the chi-square ranking function, i.e. `ranking="chisquare"`:

```
sage: S.brute_force(C, ranking="chisquare")

[(8, SHIFTINGUSINGMODULARARITHMETIC),
(14, MBCZNCHAOMCHAGIXOFULULCNBGYNCW),
(20, GVWTHWBUIGWBUACRIZOFOFWHVASHWQ),
(13, NCDAODIBPNDIBHJYPGVMVMDOCHZODX),
(1, ZOPMAPUNBZPUNTVKBSHYHYPAOTLAPJ),
(23, DSTQETYRFDTYRXZOFWLCLCTESXPETN),
(10, QFGDRGLESQGLEKMBSJYPYPGRFKCRGA),
(6, UJKHVKPIWUKPIOQFWNCTCTKVJOGVKE),
(22, ETURFUZSGEUZSYAPGXMDMDUFTYQFUO),
(15, LABYMBGZNLBGZFHWNETKTKBMAFXMBV),
(12, ODEBPEJCQOEJCIKZQHWNWNEPDIAPEY),
(21, FUVSGVATHFVATZBQHYNENEVGUZRGVP),
(16, KZAXLAFYMKAFYEGVMDSJSJALZEWLAU),
(25, BQROCRWPDBRWPVXMDUJAJARCQVNCRL),
(9, RGHESHMFTRHMFLNCTKZQZQHSGLDSHB),
(24, CRSPDSXQECSXQWYNEVKBKBSDRWODSM),
(3, XMNKYNSLZXNSLRTIZQFWFWNYMRJYNH),
(5, VKLIWLQJXVLQJPRGXODUDULWKPHWLF),
(7, TIJGUJOHVTJOHNPEVMBSBSJUINFUJD),
(2, YNOLZOTMAYOTMSUJARGXGXOZNSKZOI),
(18, IXYVJYDWKIYDWCETKBQHQHYJXCUJYS),
(4, WLMJXMRKYWMRKQSHYPEVEVMXLQIXMG),
(11, PEFCQFKDRPFKDJLARIXOXOFQEJBQFZ),
(19, HWXUIXCVJHXCVBDSJAPGPGXIWBTIXR),
(0, APQNBQVOCAQVOUWLCTIZIZQBPUMBQK),
(17, JYZWKZEXLJZEXDFULCRIRIZKYDVKZT)]
```

Use the squared differences ranking function, i.e. `ranking="squared_differences"`:

```
sage: S.brute_force(C, ranking="squared_differences")

[(8, SHIFTINGUSINGMODULARARITHMETIC),
(23, DSTQETYRFDTYRXZOFWLCLCTESXPETN),
(12, ODEBPEJCQOEJCIKZQHWNWNEPDIAPEY),
(2, YNOLZOTMAYOTMSUJARGXGXOZNSKZOI),
(9, RGHESHMFTRHMFLNCTKZQZQHSGLDSHB),
(7, TIJGUJOHVTJOHNPEVMBSBSJUINFUJD),
(21, FUVSGVATHFVATZBQHYNENEVGUZRGVP),
(22, ETURFUZSGEUZSYAPGXMDMDUFTYQFUO),
(1, ZOPMAPUNBZPUNTVKBSHYHYPAOTLAPJ),
(16, KZAXLAFYMKAFYEGVMDSJSJALZEWLAU),
(20, GVWTHWBUIGWBUACRIZOFOFWHVASHWQ),
(24, CRSPDSXQECSXQWYNEVKBKBSDRWODSM),
(14, MBCZNCHAOMCHAGIXOFULULCNBGYNCW),
(13, NCDAODIBPNDIBHJYPGVMVMDOCHZODX),
(3, XMNKYNSLZXNSLRTIZQFWFWNYMRJYNH),
```

(continues on next page)

(continued from previous page)

```
(10, QFGDRGLESQGLEKMBSJYPYPGRFKCRGA),
(15, LABYMBGZNLBGZFHWNETKTKBMAFXMBV),
(6, UJKHVKPIWUKPIOQFWNCTCTKVJOGVKE),
(11, PEFCQFKDRPFKDJLARIXOXOFQEJBQFZ),
(25, BQROCRWPDBRWPVXMDUJAJARCQVNCRL),
(17, JYZWKZEXLJZEXDFULCRIRIZKYDVKZT),
(19, HWXUIXCVJHXCVBDSJAPGPGXIWBTIXR),
(4, WLMJXMRKYWMRKQSHYPEVEVMXLQIXMG),
(0, APQNBQVOCAQVOUWLCTIZIZQBPUMBQK),
(18, IXYVJYDWKIYDWCETKBQHQHYJXCUJYS),
(5, VKLIWLQJXVLQJPRGXODUDULWKPHWLF)]
```

deciphering(*K*, *C*)

Decrypt the ciphertext `C` with the key `K` using shift cipher decryption.

INPUT:

- `K` – a secret key; a key belonging to the key space of this shift cipher. This key is an integer k satisfying the inequality $0 \leq k < n$, where n is the size of the cipher domain.
- `C` – a string of ciphertext; possibly an empty string. Characters in this string must be encoded using one of the supported alphabets. See the method *encoding()* for more information.

OUTPUT:

- The plaintext corresponding to the ciphertext `C`.

EXAMPLES:

Let's perform decryption over the supported alphabets. Here is decryption over the capital letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Stop shifting me."); P
STOPSHIFTINGME
sage: K = 13
sage: C = S.enciphering(K, P); C
FGBCFUVSGVATZR
sage: S.deciphering(K, C) == P
True
```

Decryption over the hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: P = S.encoding("Shift me now."); P
5368696674206d65206e6f772e
sage: K = 7
sage: C = S.enciphering(K, P); C
cadfd0ddeb97d4dc97d5d6ee95
sage: S.deciphering(K, C) == P
True
```

Decryption over the binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: P = S.encoding("OK, enough shifting."); P
010011110100101100101100001000000110010101101110011011110111010101100111011010000010000001111
sage: K = 1
sage: C = S.enciphering(K, P); C
```

(continues on next page)

(continued from previous page)

```
10110000101101001101001111011111100110101001000110010000100010101001100010010111110111111000
sage: S.deciphering(K, C) == P
True
```

enciphering(*K*, *P*)

Encrypt the plaintext `P` with the key `K` using shift cipher encryption.

INPUT:

- `K` – a key belonging to the key space of this shift cipher. This key is an integer k satisfying the inequality $0 \leq k < n$, where n is the size of the cipher domain.
- `P` – a string of plaintext; possibly an empty string. Characters in this string must be encoded using one of the supported alphabets. See the method *encoding()* for more information.

OUTPUT:

- The ciphertext corresponding to the plaintext `P`.

EXAMPLES:

Let's perform encryption over the supported alphabets. Here is encryption over the capital letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shift your gear."); P
SHIFTYOURGEAR
sage: K = 3
sage: S.enciphering(K, P)
VKLIWBRXUJHDU
```

Encryption over the hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: P = S.encoding("Capitalize with the shift key."); P
4361706974616c697a652077697468207468652073686967420 6b65792e
sage: K = 5
sage: S.enciphering(K, P)
98b6c5bec9b6b1becfba75ccbec9bd75c9bdba75c8bdbebbc975b0bace73
```

Encryption over the binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: P = S.encoding("Don't shift."); P
010001000110111101101110001001110111010000100000011100110110100001101001011001100111010000010
sage: K = 1
sage: S.enciphering(K, P)
101110111001000010010001110110001000101111011111100011001001011110010110100110011000101111010
```

encoding(*S*)

The encoding of the string `S` over the string monoid of this shift cipher. For example, if the string monoid of this cryptosystem is `AlphabeticStringMonoid`, then the encoding of `S` would be its upper-case equivalent stripped of all non-alphabetic characters. The following alphabets are supported for the shift cipher:

- capital letters of the English alphabet as implemented in `AlphabeticStrings()`
- the alphabet consisting of the hexadecimal number system as implemented in `HexadecimalStrings()`

- the alphabet consisting of the binary number system as implemented in `BinaryStrings()`

INPUT:

- `S` – a string, possibly empty.

OUTPUT:

- The encoding of `S` over the string monoid of this cryptosystem. If `S` is an empty string, return an empty string.

EXAMPLES:

Encoding over the upper-case letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: S.encoding("Shift cipher on capital letters of the English alphabet.")
SHIFTCIPHERONCAPITALLETTERSOFTHEENGLISHALPHABET
```

Encoding over the binary system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: S.encoding("Binary")
010000100110100101101110011000010111001001111001
```

Encoding over the hexadecimal system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: S.encoding("Over hexadecimal system.")
4f766572206865786164563696d616c2073797374656d2e
```

The argument `S` can be an empty string, in which case an empty string is returned:

```
sage: ShiftCryptosystem(AlphabeticStrings()).encoding("")

sage: ShiftCryptosystem(HexadecimalStrings()).encoding("")

sage: ShiftCryptosystem(BinaryStrings()).encoding("")
```

inverse_key(*K*)

The inverse key corresponding to the key `K`. For the shift cipher, the inverse key corresponding to `K` is $-K \bmod n$, where $n > 0$ is the size of the cipher domain, i.e. the plaintext/ciphertext space. A key k of the shift cipher is an integer $0 \leq k < n$. The key $k = 0$ has no effect on either the plaintext or the ciphertext.

INPUT:

- `K` – a key for this shift cipher. This must be an integer k such that $0 \leq k < n$, where n is the size of the cipher domain.

OUTPUT:

- The inverse key corresponding to `K`.

EXAMPLES:

Some random keys and their respective inverse keys:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: key = S.random_key(); key  # random
2
sage: S.inverse_key(key)         # random
```

(continues on next page)

(continued from previous page)

```
24
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: key = S.random_key(); key  # random
12
sage: S.inverse_key(key)         # random
4
sage: S = ShiftCryptosystem(BinaryStrings())
sage: key = S.random_key(); key  # random
1
sage: S.inverse_key(key)         # random
1
sage: key = S.random_key(); key  # random
0
sage: S.inverse_key(key)         # random
0
```

Regardless of the value of a key, the addition of the key and its inverse must be equal to the alphabet size. This relationship holds exactly when the value of the key is non-zero:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: K = S.random_key()
sage: while K == 0:
....:     K = S.random_key()
sage: invK = S.inverse_key(K)
sage: K + invK == S.alphabet_size()
True
sage: invK + K == S.alphabet_size()
True
sage: K = S.random_key()
sage: while K != 0:
....:     K = S.random_key()
sage: invK = S.inverse_key(K)
sage: K + invK != S.alphabet_size()
True
sage: K; invK
0
0
```

random_key()

Generate a random key within the key space of this shift cipher. The generated key is an integer $0 \leq k < n$ with n being the size of the cipher domain. Thus there are n possible keys in the key space, which is the set $\mathbf{Z}/n\mathbf{Z}$. The key $k = 0$ has no effect on either the plaintext or the ciphertext.

OUTPUT:

- A random key within the key space of this shift cryptosystem.

EXAMPLES:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: S.random_key()  # random
18
sage: S = ShiftCryptosystem(BinaryStrings())
sage: S.random_key()  # random
0
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: S.random_key()  # random
5
```

Regardless of the value of a key, the addition of the key and its inverse must be equal to the alphabet size. This relationship holds exactly when the value of the key is non-zero:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: K = S.random_key()
sage: while K == 0:
....:     K = S.random_key()
sage: invK = S.inverse_key(K)
sage: K + invK == S.alphabet_size()
True
sage: invK + K == S.alphabet_size()
True
sage: K = S.random_key()
sage: while K != 0:
....:     K = S.random_key()
sage: invK = S.inverse_key(K)
sage: K + invK != S.alphabet_size()
True
sage: K; invK
0
0
```

rank_by_chi_square(*C*, *pdict*)

Use the chi-square statistic to rank all possible keys. Currently, this method only applies to the capital letters of the English alphabet.

ALGORITHM:

Consider a non-empty alphabet A consisting of n elements, and let C be a ciphertext encoded using elements of A. The plaintext P corresponding to C is also encoded using elements of A. Let M be a candidate decipherment of C, i.e. M is the result of attempting to decrypt C using a key $k \in \mathbf{Z}/n\mathbf{Z}$ which is not necessarily the same key used to encrypt P. Suppose $F_A(e)$ is the characteristic frequency probability of $e \in A$ and let $F_M(e)$ be the message frequency probability with respect to M. The characteristic frequency probability distribution of an alphabet is the expected frequency probability distribution for that alphabet. The message frequency probability distribution of M provides a distribution of the ratio of character occurrences over message length. One can interpret the characteristic frequency probability $F_A(e)$ as the expected probability, while the message frequency probability $F_M(e)$ is the observed probability. If M is of length L, then the observed frequency of $e \in A$ is

$$O_M(e) = F_M(e) \cdot L$$

and the expected frequency of $e \in A$ is

$$E_A(e) = F_A(e) \cdot L$$

The chi-square rank $R_{\chi^2}(M)$ of M corresponding to a key $k \in \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{\chi^2}(M) = \sum_{e \in A} \frac{\big(O_M(e) - E_A(e)\big)^2}{E_A(e)}$$

Cryptanalysis by exhaustive key search produces a candidate decipherment M_k for each possible key $k \in \mathbf{Z}/n\mathbf{Z}$. For a set $D = \{M_{k_1}, M_{k_2}, \ldots, M_{k_r}\}$ of all candidate decipherments corresponding to a ciphertext C, the smaller is the rank $R_{\chi^2}(M_{k_i})$ the more likely that k_i is the secret key. This key ranking method is based on the Pearson chi-square test [PearsonTest].

INPUT:

- `C` – The ciphertext, a non-empty string. The ciphertext must be encoded using the upper-case letters of the English alphabet.

- pdict – A dictionary of key, possible plaintext pairs. This should be the output of *brute_force()* with ranking="none".

OUTPUT:

- A list ranking the most likely keys first. Each element of the list is a tuple of key, possible plaintext pairs.

EXAMPLES:

Use the chi-square statistic to rank all possible keys and their corresponding decipherment:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shi."); P
SHI
sage: K = 5
sage: C = S.enciphering(K, P)
sage: Pdict = S.brute_force(C)
sage: S.rank_by_chi_square(C, Pdict)

[(9, ODE),
(5, SHI),
(20, DST),
(19, ETU),
(21, CRS),
(10, NCD),
(25, YNO),
(6, RGH),
(12, LAB),
(8, PEF),
(1, WLM),
(11, MBC),
(18, FUV),
(17, GVW),
(2, VKL),
(4, TIJ),
(3, UJK),
(0, XMN),
(16, HWX),
(15, IXY),
(23, APQ),
(24, ZOP),
(22, BQR),
(7, QFG),
(13, KZA),
(14, JYZ)]
```

As more ciphertext is available, the reliability of the chi-square ranking function increases:

```
sage: P = S.encoding("Shift cipher."); P
SHIFTCIPHER
sage: C = S.enciphering(K, P)
sage: Pdict = S.brute_force(C)
sage: S.rank_by_chi_square(C, Pdict)

[(5, SHIFTCIPHER),
(9, ODEBPYELDAN),
(18, FUVSGPVCURE),
(2, VKLIWFLSKHU),
```

(continues on next page)

(continued from previous page)

```
(20, DSTQENTASPC),
(19, ETURFOUBTQD),
(21, CRSPDMSZROB),
(6, RGHESBHOGDQ),
(7, QFGDRAGNFCP),
(12, LABYMVBIAXK),
(17, GVWTHQWDVSF),
(24, ZOPMAJPWOLY),
(1, WLMJXGMTLIV),
(0, XMNKYHNUMJW),
(11, MBCZNWCJBYL),
(8, PEFCQZFMEBO),
(25, YNOLZIOVNKX),
(10, NCDAOXDKCZM),
(3, UJKHVEKRJGT),
(4, TIJGUDJQIFS),
(22, BQROCLRYQNA),
(16, HWXUIRXEWTG),
(15, IXYVJSYFXUH),
(14, JYZWKTZGYVI),
(13, KZAXLUAHZWJ),
(23, APQNBKQXPMZ)]
```

rank_by_squared_differences(*C*, *pdict*)

Use the squared-differences measure to rank all possible keys. Currently, this method only applies to the capital letters of the English alphabet.

ALGORITHM:

Consider a non-empty alphabet A consisting of n elements, and let C be a ciphertext encoded using elements of A. The plaintext P corresponding to C is also encoded using elements of A. Let M be a candidate decipherment of C, i.e. M is the result of attempting to decrypt C using a key $k \in \mathbf{Z}/n\mathbf{Z}$ which is not necessarily the same key used to encrypt P. Suppose $F_A(e)$ is the characteristic frequency probability of $e \in A$ and let $F_M(e)$ be the message frequency probability with respect to M. The characteristic frequency probability distribution of an alphabet is the expected frequency probability distribution for that alphabet. The message frequency probability distribution of M provides a distribution of the ratio of character occurrences over message length. One can interpret the characteristic frequency probability $F_A(e)$ as the expected probability, while the message frequency probability $F_M(e)$ is the observed probability. If M is of length L, then the observed frequency of $e \in A$ is

$$O_M(e) = F_M(e) \cdot L$$

and the expected frequency of $e \in A$ is

$$E_A(e) = F_A(e) \cdot L$$

The squared-differences, or residual sum of squares, rank $R_{RSS}(M)$ of M corresponding to a key $k \in \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{RSS}(M) = \sum_{e \in A} \big(O_M(e) - E_A(e)\big)^2$$

Cryptanalysis by exhaustive key search produces a candidate decipherment M_k for each possible key $k \in \mathbf{Z}/n\mathbf{Z}$. For a set $D = \{M_{k_1}, M_{k_2}, \ldots, M_{k_r}\}$ of all candidate decipherments corresponding to a ciphertext C, the smaller is the rank $R_{RSS}(M_{k_i})$ the more likely that k_i is the secret key. This key ranking method is based on the residual sum of squares measure [RSS].

INPUT:

- C – The ciphertext, a non-empty string. The ciphertext must be encoded using the upper-case letters of the English alphabet.
- pdict – A dictionary of key, possible plaintext pairs. This should be the output of *brute_force()* with ranking="none".

OUTPUT:

- A list ranking the most likely keys first. Each element of the list is a tuple of key, possible plaintext pairs.

EXAMPLES:

Use the method of squared differences to rank all possible keys and their corresponding decipherment:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shi."); P
SHI
sage: K = 5
sage: C = S.enciphering(K, P)
sage: Pdict = S.brute_force(C)
sage: S.rank_by_squared_differences(C, Pdict)

[(19, ETU),
(9, ODE),
(20, DST),
(5, SHI),
(8, PEF),
(4, TIJ),
(25, YNO),
(21, CRS),
(6, RGH),
(10, NCD),
(12, LAB),
(23, APQ),
(24, ZOP),
(0, XMN),
(13, KZA),
(15, IXY),
(1, WLM),
(16, HWX),
(22, BQR),
(11, MBC),
(18, FUV),
(2, VKL),
(17, GVW),
(7, QFG),
(3, UJK),
(14, JYZ)]
```

As more ciphertext is available, the reliability of the squared differences ranking function increases:

```
sage: P = S.encoding("Shift cipher."); P
SHIFTCIPHER
sage: C = S.enciphering(K, P)
sage: Pdict = S.brute_force(C)
sage: S.rank_by_squared_differences(C, Pdict)

[(20, DSTQENTASPC),
(5, SHIFTCIPHER),
```

(continues on next page)

(continued from previous page)

```
(9, ODEBPYELDAN),
(19, ETURFOUBTQD),
(6, RGHESBHOGDQ),
(16, HWXUIRXEWTG),
(8, PEFCQZFMEBO),
(21, CRSPDMSZROB),
(22, BQROCLRYQNA),
(25, YNOLZIOVNKX),
(3, UJKHVEKRJGT),
(18, FUVSGPVCURE),
(4, TIJGUDJQIFS),
(10, NCDAOXDKCZM),
(7, QFGDRAGNFCP),
(24, ZOPMAJPWOLY),
(2, VKLIWFLSKHU),
(12, LABYMVBIAXK),
(17, GVWTHQWDVSF),
(1, WLMJXGMTLIV),
(13, KZAXLUAHZWJ),
(0, XMNKYHNUMJW),
(15, IXYVJSYFXUH),
(14, JYZWKTZGYVI),
(11, MBCZNWCJBYL),
(23, APQNBKQXPMZ)]
```

class sage.crypto.classical.**SubstitutionCryptosystem**(*S*)

Bases: *sage.crypto.cryptosystem.SymmetricKeyCryptosystem*

Create a substitution cryptosystem.

INPUT:

- `S` - a string monoid over some alphabet

OUTPUT:

- A substitution cryptosystem over the alphabet `S`.

EXAMPLES:

```
sage: M = AlphabeticStrings()
sage: E = SubstitutionCryptosystem(M)
sage: E
Substitution cryptosystem on Free alphabetic string monoid on A-Z
sage: K = M([ 25-i for i in range(26) ])
sage: K
ZYXWVUTSRQPONMLKJIHGFEDCBA
sage: e = E(K)
sage: m = M("THECATINTHEHAT")
sage: e(m)
GSVXZGRMGSVSZG
```

deciphering(*K*, *C*)

Decrypt the ciphertext `C` using the key `K`.

INPUT:

- `K` - a key belonging to the key space of this substitution cipher
- `C` - a string (possibly empty) over the string monoid of this cryptosystem.

OUTPUT:

- The plaintext corresponding to the ciphertext C.

EXAMPLES:

```
sage: S = SubstitutionCryptosystem(AlphabeticStrings())
sage: K = S.random_key()
sage: M = S.encoding("Don't substitute me!")
sage: S.deciphering(K, S.enciphering(K, M)) == M
True
```

enciphering(*K*, *M*)

Encrypt the plaintext M using the key K.

INPUT:

- K - a key belonging to the key space of this substitution cipher
- M - a string (possibly empty) over the string monoid of this cryptosystem.

OUTPUT:

- The ciphertext corresponding to the plaintext M.

EXAMPLES:

```
sage: S = SubstitutionCryptosystem(AlphabeticStrings())
sage: K = S.random_key()
sage: M = S.encoding("Don't substitute me.")
sage: S.deciphering(K, S.enciphering(K, M)) == M
True
```

encoding(*M*)

The encoding of the string M over the string monoid of this substitution cipher. For example, if the string monoid of this cryptosystem is AlphabeticStringMonoid, then the encoding of M would be its upper-case equivalent stripped of all non-alphabetic characters.

INPUT:

- M - a string, possibly empty

OUTPUT:

- The encoding of M over the string monoid of this cryptosystem.

EXAMPLES:

```
sage: M = "Peter Pan(ning) for gold."
sage: A = AlphabeticStrings()
sage: S = SubstitutionCryptosystem(A)
sage: S.encoding(M) == A.encoding(M)
True
```

inverse_key(*K*)

The inverse key corresponding to the key K. The specified key is a permutation of the cryptosystem alphabet.

INPUT:

- K - a key belonging to the key space of this cryptosystem

OUTPUT:

- The inverse key of K.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = SubstitutionCryptosystem(S)
sage: K = E.random_key()
sage: L = E.inverse_key(K)
sage: M = S("THECATINTHEHAT")
sage: e = E(K)
sage: c = E(L)
sage: c(e(M))
THECATINTHEHAT
```

random_key()

Generate a random key within the key space of this substitution cipher. The generated key is a permutation of the cryptosystem alphabet. Let n be the length of the alphabet. Then there are $n!$ possible keys in the key space.

OUTPUT:

- A random key within the key space of this cryptosystem.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: S = SubstitutionCryptosystem(A)
sage: K = S.random_key()
sage: Ki = S.inverse_key(K)
sage: M = "THECATINTHEHAT"
sage: e = S(K)
sage: d = S(Ki)
sage: d(e(A(M))) == A(M)
True
```

class sage.crypto.classical.**TranspositionCryptosystem**(S, n)

Bases: *sage.crypto.cryptosystem.SymmetricKeyCryptosystem*

Create a transposition cryptosystem of block length n.

INPUT:

- S - a string monoid over some alphabet
- n - integer > 0; a block length of a block permutation

OUTPUT:

- A transposition cryptosystem of block length n over the alphabet S.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = TranspositionCryptosystem(S,14)
sage: E
Transposition cryptosystem on Free alphabetic string monoid on A-Z of block␣
↪length 14
sage: K = [ 14-i for i in range(14) ]
sage: K
[14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: e = E(K)
```

(continues on next page)

(continued from previous page)

```
sage: e(S("THECATINTHEHAT"))
TAHEHTNITACEHT
```

deciphering(*K*, *C*)

Decrypt the ciphertext `C` using the key `K`.

INPUT:

- `K` - a key belonging to the key space of this transposition cipher
- `C` - a string (possibly empty) over the string monoid of this cryptosystem.

OUTPUT:

- The plaintext corresponding to the ciphertext `C`.

EXAMPLES:

```
sage: T = TranspositionCryptosystem(AlphabeticStrings(), 14)
sage: K = T.random_key()
sage: M = T.encoding("The cat in the hat.")
sage: T.deciphering(K, T.enciphering(K, M)) == M
True
```

enciphering(*K*, *M*)

Encrypt the plaintext `M` using the key `K`.

INPUT:

- `K` - a key belonging to the key space of this transposition cipher
- `M` - a string (possibly empty) over the string monoid of this cryptosystem

OUTPUT:

- The ciphertext corresponding to the plaintext `M`.

EXAMPLES:

```
sage: T = TranspositionCryptosystem(AlphabeticStrings(), 14)
sage: K = T.random_key()
sage: M = T.encoding("The cat in the hat.")
sage: T.deciphering(K, T.enciphering(K, M)) == M
True
```

encoding(*M*)

The encoding of the string `M` over the string monoid of this transposition cipher. For example, if the string monoid of this cryptosystem is `AlphabeticStringMonoid`, then the encoding of `M` would be its upper-case equivalent stripped of all non-alphabetic characters.

INPUT:

- `M` - a string, possibly empty

OUTPUT:

- The encoding of `M` over the string monoid of this cryptosystem.

EXAMPLES:

```
sage: M = "Transposition cipher is not about matrix transpose."
sage: A = AlphabeticStrings()
sage: T = TranspositionCryptosystem(A, 11)
sage: T.encoding(M) == A.encoding(M)
True
```

inverse_key(*K*, *check=True*)

The inverse key corresponding to the key `K`.

INPUT:

- `K` - a key belonging to the key space of this transposition cipher
- `check` - bool (default: `True`); check that `K` belongs to the key space of this cryptosystem.

OUTPUT:

- The inverse key corresponding to `K`.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = TranspositionCryptosystem(S, 14)
sage: K = E.random_key()
sage: Ki = E.inverse_key(K)
sage: e = E(K)
sage: d = E(Ki)
sage: M = "THECATINTHEHAT"
sage: C = e(S(M))
sage: d(S(C)) == S(M)
True
```

random_key()

Generate a random key within the key space of this transposition cryptosystem. Let $n > 0$ be the block length of this cryptosystem. Then there are $n!$ possible keys.

OUTPUT:

- A random key within the key space of this cryptosystem.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = TranspositionCryptosystem(S, 14)
sage: K = E.random_key()
sage: Ki = E.inverse_key(K)
sage: e = E(K)
sage: d = E(Ki)
sage: M = "THECATINTHEHAT"
sage: C = e(S(M))
sage: d(S(C)) == S(M)
True
```

class sage.crypto.classical.**VigenereCryptosystem**(*S*, *n*)

Bases: *sage.crypto.cryptosystem.SymmetricKeyCryptosystem*

Create a Vigenere cryptosystem of block length `n`.

INPUT:

- `S`– a string monoid over some alphabet

- n - integer > 0; block length of an encryption/decryption key

OUTPUT:

- A Vigenere cryptosystem of block length n over the alphabet S.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = VigenereCryptosystem(S,14)
sage: E
Vigenere cryptosystem on Free alphabetic string monoid on A-Z of period 14
sage: K = S('ABCDEFGHIJKLMN')
sage: K
ABCDEFGHIJKLMN
sage: e = E(K)
sage: e
Cipher on Free alphabetic string monoid on A-Z
sage: e(S("THECATINTHEHAT"))
TIGFEYOUBQOSMG
```

deciphering(*K*, *C*)

Decrypt the ciphertext C using the key K.

INPUT:

- K - a key belonging to the key space of this Vigenere cipher
- C - a string (possibly empty) over the string monoid of this cryptosystem

OUTPUT:

- The plaintext corresponding to the ciphertext C.

EXAMPLES:

```
sage: V = VigenereCryptosystem(AlphabeticStrings(), 24)
sage: K = V.random_key()
sage: M = V.encoding("Jack and Jill went up the hill.")
sage: V.deciphering(K, V.enciphering(K, M)) == M
True
```

enciphering(*K*, *M*)

Encrypt the plaintext M using the key K.

INPUT:

- K - a key belonging to the key space of this Vigenere cipher
- M - a string (possibly empty) over the string monoid of this cryptosystem

OUTPUT:

- The ciphertext corresponding to the plaintext M.

EXAMPLES:

```
sage: V = VigenereCryptosystem(AlphabeticStrings(), 24)
sage: K = V.random_key()
sage: M = V.encoding("Jack and Jill went up the hill.")
sage: V.deciphering(K, V.enciphering(K, M)) == M
True
```

encoding(*M*)

The encoding of the string `M` over the string monoid of this Vigenere cipher. For example, if the string monoid of this cryptosystem is `AlphabeticStringMonoid`, then the encoding of `M` would be its upper-case equivalent stripped of all non-alphabetic characters.

INPUT:

- `M` - a string, possibly empty

OUTPUT:

- The encoding of `M` over the string monoid of this cryptosystem.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: V = VigenereCryptosystem(A, 24)
sage: M = "Jack and Jill went up the hill."
sage: V.encoding(M) == A.encoding(M)
True
```

inverse_key(*K*)

The inverse key corresponding to the key `K`.

INPUT:

- `K` - a key within the key space of this Vigenere cryptosystem

OUTPUT:

- The inverse key corresponding to `K`.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = VigenereCryptosystem(S,14)
sage: K = E.random_key()
sage: L = E.inverse_key(K)
sage: M = S("THECATINTHEHAT")
sage: e = E(K)
sage: c = E(L)
sage: c(e(M))
THECATINTHEHAT
```

random_key()

Generate a random key within the key space of this Vigenere cryptosystem. Let $n > 0$ be the length of the cryptosystem alphabet and let $m > 0$ be the block length of this cryptosystem. Then there are n^m possible keys.

OUTPUT:

- A random key within the key space of this cryptosystem.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: V = VigenereCryptosystem(A, 14)
sage: M = "THECATINTHEHAT"
sage: K = V.random_key()
sage: Ki = V.inverse_key(K)
sage: e = V(K)
sage: d = V(Ki)
```

(continues on next page)

(continued from previous page)

```
sage: d(e(A(M))) == A(M)
True
```

CHAPTER

FOUR

CLASSICAL CIPHERS

class sage.crypto.classical_cipher.**AffineCipher**(*parent*, *key*)
Bases: *sage.crypto.cipher.SymmetricKeyCipher*

Affine cipher class. This is the class that does the actual work of encryption and decryption. Users should not directly instantiate or create objects of this class. Instead, functionalities of this class should be accessed via *AffineCryptosystem* as the latter provides a convenient user interface.

class sage.crypto.classical_cipher.**HillCipher**(*parent*, *key*)
Bases: *sage.crypto.cipher.SymmetricKeyCipher*

Hill cipher class

inverse()

class sage.crypto.classical_cipher.**ShiftCipher**(*parent*, *key*)
Bases: *sage.crypto.cipher.SymmetricKeyCipher*

Shift cipher class. This is the class that does the actual work of encryption and decryption. Users should not directly instantiate or create objects of this class. Instead, functionalities of this class should be accessed via *ShiftCryptosystem* as the latter provides a convenient user interface.

class sage.crypto.classical_cipher.**SubstitutionCipher**(*parent*, *key*)
Bases: *sage.crypto.cipher.SymmetricKeyCipher*

Substitution cipher class

inverse()

class sage.crypto.classical_cipher.**TranspositionCipher**(*parent*, *key*)
Bases: *sage.crypto.cipher.SymmetricKeyCipher*

Transition cipher class

inverse()

class sage.crypto.classical_cipher.**VigenereCipher**(*parent*, *key*)
Bases: *sage.crypto.cipher.SymmetricKeyCipher*

Vigenere cipher class

inverse()

CHAPTER

FIVE

SIMPLIFIED DES

A simplified variant of the Data Encryption Standard (DES). Note that Simplified DES or S-DES is for educational purposes only. It is a small-scale version of the DES designed to help beginners understand the basic structure of DES.

AUTHORS:

- Minh Van Nguyen (2009-06): initial version

class sage.crypto.block_cipher.sdes.**SimplifiedDES**

Bases: sage.structure.sage_object.SageObject

This class implements the Simplified Data Encryption Standard (S-DES) described in [Sch1996]. Schaefer's S-DES is for educational purposes only and is not secure for practical purposes. S-DES is a version of the DES with all parameters significantly reduced, but at the same time preserving the structure of DES. The goal of S-DES is to allow a beginner to understand the structure of DES, thus laying a foundation for a thorough study of DES. Its goal is as a teaching tool in the same spirit as Phan's *Mini-AES* [Pha2002].

EXAMPLES:

Encrypt a random block of 8-bit plaintext using a random key, decrypt the ciphertext, and compare the result with the original plaintext:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES(); sdes
Simplified DES block cipher with 10-bit keys
sage: bin = BinaryStrings()
sage: P = [bin(str(randint(0, 1))) for i in range(8)]
sage: K = sdes.random_key()
sage: C = sdes.encrypt(P, K)
sage: plaintxt = sdes.decrypt(C, K)
sage: plaintxt == P
True
```

We can also encrypt binary strings that are larger than 8 bits in length. However, the number of bits in that binary string must be positive and a multiple of 8:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: bin = BinaryStrings()
sage: P = bin.encoding("Encrypt this using S-DES!")
sage: Mod(len(P), 8) == 0
True
sage: K = sdes.list_to_string(sdes.random_key())
sage: C = sdes(P, K, algorithm="encrypt")
sage: plaintxt = sdes(C, K, algorithm="decrypt")
sage: plaintxt == P
True
```

block_length()

Return the block length of Schaefer's S-DES block cipher. A key in Schaefer's S-DES is a block of 10 bits.

OUTPUT:

- The block (or key) length in number of bits.

EXAMPLES:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.block_length()
10
```

decrypt(C, K)

Return an 8-bit plaintext corresponding to the ciphertext `C`, using S-DES decryption with key `K`. The decryption process of S-DES is as follows. Let P be the initial permutation function, P^{-1} the corresponding inverse permutation, Π_F the permutation/substitution function, and σ the switch function. The ciphertext block `C` first goes through P, the output of which goes through Π_F using the second subkey. Then we apply the switch function to the output of the last function, and the result is then fed into Π_F using the first subkey. Finally, run the output through P^{-1} to get the plaintext.

INPUT:

- `C` – an 8-bit ciphertext; a block of 8 bits
- `K` – a 10-bit key; a block of 10 bits

OUTPUT:

The 8-bit plaintext corresponding to `C`, obtained using the key `K`.

EXAMPLES:

Decrypt an 8-bit ciphertext block:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: C = [0, 1, 0, 1, 0, 1, 0, 1]
sage: K = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.decrypt(C, K)
[0, 0, 0, 1, 0, 1, 0, 1]
```

We can also work with strings of bits:

```
sage: C = "01010101"
sage: K = "1010000010"
sage: sdes.decrypt(sdes.string_to_list(C), sdes.string_to_list(K))
[0, 0, 0, 1, 0, 1, 0, 1]
```

encrypt(P, K)

Return an 8-bit ciphertext corresponding to the plaintext `P`, using S-DES encryption with key `K`. The encryption process of S-DES is as follows. Let P be the initial permutation function, P^{-1} the corresponding inverse permutation, Π_F the permutation/substitution function, and σ the switch function. The plaintext block `P` first goes through P, the output of which goes through Π_F using the first subkey. Then we apply the switch function to the output of the last function, and the result is then fed into Π_F using the second subkey. Finally, run the output through P^{-1} to get the ciphertext.

INPUT:

- `P` – an 8-bit plaintext; a block of 8 bits

- `K` – a 10-bit key; a block of 10 bits

OUTPUT:

The 8-bit ciphertext corresponding to `P`, obtained using the key `K`.

EXAMPLES:

Encrypt an 8-bit plaintext block:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: P = [0, 1, 0, 1, 0, 1, 0, 1]
sage: K = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.encrypt(P, K)
[1, 1, 0, 0, 0, 0, 0, 1]
```

We can also work with strings of bits:

```
sage: P = "01010101"
sage: K = "1010000010"
sage: sdes.encrypt(sdes.string_to_list(P), sdes.string_to_list(K))
[1, 1, 0, 0, 0, 0, 0, 1]
```

initial_permutation(*B*, *inverse=False*)

Return the initial permutation of `B`. Denote the initial permutation function by P and let $(b_0, b_1, b_2, \ldots, b_7)$ be a vector of 8 bits, where each $b_i \in \{0, 1\}$. Then

$$P(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (b_1, b_5, b_2, b_0, b_3, b_7, b_4, b_6)$$

The inverse permutation is P^{-1}:

$$P^{-1}(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (b_3, b_0, b_2, b_4, b_6, b_1, b_7, b_5)$$

INPUT:

- `B` – list; a block of 8 bits
- `inverse` – (default: `False`) if `True` then use the inverse permutation P^{-1}; if `False` then use the initial permutation P

OUTPUT:

The initial permutation of `B` if `inverse=False`, or the inverse permutation of `B` if `inverse=True`.

EXAMPLES:

The initial permutation of a list of 8 bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 0, 1, 1, 0, 1, 0, 0]
sage: P = sdes.initial_permutation(B); P
[0, 1, 1, 1, 1, 0, 0, 0]
```

Recovering the original list of 8 bits from the permutation:

```
sage: Pinv = sdes.initial_permutation(P, inverse=True)
sage: Pinv; B
[1, 0, 1, 1, 0, 1, 0, 0]
[1, 0, 1, 1, 0, 1, 0, 0]
```

We can also work with a string of bits:

```
sage: S = "10110100"
sage: L = sdes.string_to_list(S)
sage: P = sdes.initial_permutation(L); P
[0, 1, 1, 1, 1, 0, 0, 0]
sage: sdes.initial_permutation(sdes.string_to_list("01111000"), inverse=True)
[1, 0, 1, 1, 0, 1, 0, 0]
```

left_shift(*B*, *n=1*)

Return a circular left shift of `B` by `n` positions. Let $B = (b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9)$ be a vector of 10 bits. Then the left shift operation L_n is performed on the first 5 bits and the last 5 bits of B separately. That is, if the number of shift positions is `n=1`, then L_1 is defined as

$$L_1(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_1, b_2, b_3, b_4, b_0, b_6, b_7, b_8, b_9, b_5)$$

If the number of shift positions is `n=2`, then L_2 is given by

$$L_2(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_2, b_3, b_4, b_0, b_1, b_7, b_8, b_9, b_5, b_6)$$

INPUT:

- `B` – a list of 10 bits
- `n` – (default: 1) if `n=1` then perform left shift by 1 position; if `n=2` then perform left shift by 2 positions. The valid values for `n` are 1 and 2, since only up to 2 positions are defined for this circular left shift operation.

OUTPUT:

The circular left shift of each half of `B`.

EXAMPLES:

Circular left shift by 1 position of a 10-bit string:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 0, 0, 0, 0, 0, 1, 1, 0, 0]
sage: sdes.left_shift(B)
[0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
sage: sdes.left_shift([1, 0, 1, 0, 0, 0, 0, 0, 1, 0])
[0, 1, 0, 0, 1, 0, 0, 1, 0, 0]
```

Circular left shift by 2 positions of a 10-bit string:

```
sage: B = [0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
sage: sdes.left_shift(B, n=2)
[0, 0, 1, 0, 0, 0, 0, 0, 1, 1]
```

Here we work with a string of bits:

```
sage: S = "1000001100"
sage: L = sdes.string_to_list(S)
sage: sdes.left_shift(L)
[0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
sage: sdes.left_shift(sdes.string_to_list("1010000010"), n=2)
[1, 0, 0, 1, 0, 0, 1, 0, 0, 0]
```

list_to_string(*B*)

Return a binary string representation of the list `B`.

INPUT:

- `B` – a non-empty list of bits

OUTPUT:

The binary string representation of `B`.

EXAMPLES:

A binary string representation of a list of bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: L = [0, 0, 0, 0, 1, 1, 0, 1, 0, 0]
sage: sdes.list_to_string(L)
0000110100
```

permutation10(*B*)

Return a permutation of a 10-bit string. This permutation is called P_{10} and is specified as follows. Let $(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9)$ be a vector of 10 bits where each $b_i \in \{0, 1\}$. Then P_{10} is given by

$$P_{10}(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_2, b_4, b_1, b_6, b_3, b_9, b_0, b_8, b_7, b_5)$$

INPUT:

- `B` – a block of 10-bit string

OUTPUT:

A permutation of `B`.

EXAMPLES:

Permute a 10-bit string:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 1, 0, 0, 1, 0, 0, 1, 0, 1]
sage: sdes.permutation10(B)
[0, 1, 1, 0, 0, 1, 1, 0, 1, 0]
sage: sdes.permutation10([0, 1, 1, 0, 1, 0, 0, 1, 0, 1])
[1, 1, 1, 0, 0, 1, 0, 0, 1, 0]
sage: sdes.permutation10([1, 0, 1, 0, 0, 0, 0, 0, 1, 0])
[1, 0, 0, 0, 0, 0, 1, 1, 0, 0]
```

Here we work with a string of bits:

```
sage: S = "1100100101"
sage: L = sdes.string_to_list(S)
sage: sdes.permutation10(L)
[0, 1, 1, 0, 0, 1, 1, 0, 1, 0]
sage: sdes.permutation10(sdes.string_to_list("0110100101"))
[1, 1, 1, 0, 0, 1, 0, 0, 1, 0]
```

permutation4(*B*)

Return a permutation of a 4-bit string. This permutation is called P_4 and is specified as follows. Let (b_0, b_1, b_2, b_3) be a vector of 4 bits where each $b_i \in \{0, 1\}$. Then P_4 is defined by

$$P_4(b_0, b_1, b_2, b_3) = (b_1, b_3, b_2, b_0)$$

INPUT:

- B – a block of 4-bit string

OUTPUT:

A permutation of B.

EXAMPLES:

Permute a 4-bit string:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 1, 0, 0]
sage: sdes.permutation4(B)
[1, 0, 0, 1]
sage: sdes.permutation4([0, 1, 0, 1])
[1, 1, 0, 0]
```

We can also work with a string of bits:

```
sage: S = "1100"
sage: L = sdes.string_to_list(S)
sage: sdes.permutation4(L)
[1, 0, 0, 1]
sage: sdes.permutation4(sdes.string_to_list("0101"))
[1, 1, 0, 0]
```

permutation8(*B*)

Return a permutation of an 8-bit string. This permutation is called P_8 and is specified as follows. Let $(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9)$ be a vector of 10 bits where each $b_i \in \{0, 1\}$. Then P_8 picks out 8 of those 10 bits and permutes those 8 bits:

$$P_8(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_5, b_2, b_6, b_3, b_7, b_4, b_9, b_8)$$

INPUT:

- B – a block of 10-bit string

OUTPUT:

Pick out 8 of the 10 bits of B and permute those 8 bits.

EXAMPLES:

Permute a 10-bit string:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 1, 0, 0, 1, 0, 0, 1, 0, 1]
sage: sdes.permutation8(B)
[0, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.permutation8([0, 1, 1, 0, 1, 0, 0, 1, 0, 1])
[0, 1, 0, 0, 1, 1, 1, 0]
sage: sdes.permutation8([0, 0, 0, 0, 1, 1, 1, 0, 0, 0])
[1, 0, 1, 0, 0, 1, 0, 0]
```

We can also work with a string of bits:

```
sage: S = "1100100101"
sage: L = sdes.string_to_list(S)
sage: sdes.permutation8(L)
[0, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.permutation8(sdes.string_to_list("0110100101"))
[0, 1, 0, 0, 1, 1, 1, 0]
```

permute_substitute(*B*, *key*)

Apply the function Π_F on the block `B` using subkey `key`. Let $(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7)$ be a vector of 8 bits where each $b_i \in \{0, 1\}$, let L and R be the leftmost 4 bits and rightmost 4 bits of `B` respectively, and let F be a function mapping 4-bit strings to 4-bit strings. Then

$$\Pi_F(L, R) = (L \oplus F(R, S), R)$$

where S is a subkey and $\oplus$ denotes the bit-wise exclusive-OR function.

The function F can be described as follows. Its 4-bit input block (n_0, n_1, n_2, n_3) is first expanded into an 8-bit block to become $(n_3, n_0, n_1, n_2, n_1, n_2, n_3, n_0)$. This is usually represented as follows

$$\begin{array}{c|cc|c} n_3 & n_0 & n_1 & n_2 \\ n_1 & n_2 & n_3 & n_0 \end{array}$$

Let $K = (k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7)$ be an 8-bit subkey. Then K is added to the above expanded input block using exclusive-OR to produce

$$\begin{array}{c|cc|c} n_3 + k_0 & n_0 + k_1 & n_1 + k_2 & n_2 + k_3 \\ n_1 + k_4 & n_2 + k_5 & n_3 + k_6 & n_0 + k_7 \end{array} = \begin{array}{c|cc|c} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \end{array}$$

Now read the first row as the 4-bit string $p_{0,0}p_{0,3}p_{0,1}p_{0,2}$ and input this 4-bit string through S-box S_0 to get a 2-bit output.

$S_0 =$

Input	Output	Input	Output
0000	01	1000	00
0001	00	1001	10
0010	11	1010	01
0011	10	1011	11
0100	11	1100	11
0101	10	1101	01
0110	01	1110	11
0111	00	1111	10

Next read the second row as the 4-bit string $p_{1,0}p_{1,3}p_{1,1}p_{1,2}$ and input this 4-bit string through S-box S_1 to get another 2-bit output.

$S_1 =$

Input	Output	Input	Output
0000	00	1000	11
0001	01	1001	00
0010	10	1010	01
0011	11	1011	00
0100	10	1100	10
0101	00	1101	01
0110	01	1110	00
0111	11	1111	11

Denote the 4 bits produced by S_0 and S_1 as $b_0b_1b_2b_3$. This 4-bit string undergoes another permutation called P_4 as follows:

$$P_4(b_0, b_1, b_2, b_3) = (b_1, b_3, b_2, b_0)$$

The output of P_4 is the output of the function F.

INPUT:

- `B` – a list of 8 bits
- `key` – an 8-bit subkey

OUTPUT:

The result of applying the function Π_F to `B`.

EXAMPLES:

Applying the function Π_F to an 8-bit block and an 8-bit subkey:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 0, 1, 1, 1, 1, 0, 1]
sage: K = [1, 1, 0, 1, 0, 1, 0, 1]
sage: sdes.permute_substitute(B, K)
[1, 0, 1, 0, 1, 1, 0, 1]
```

We can also work with strings of bits:

```
sage: B = "10111101"
sage: K = "11010101"
sage: B = sdes.string_to_list(B); K = sdes.string_to_list(K)
sage: sdes.permute_substitute(B, K)
[1, 0, 1, 0, 1, 1, 0, 1]
```

random_key()

Return a random 10-bit key.

EXAMPLES:

The size of each key is the same as the block size:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: key = sdes.random_key()
sage: len(key) == sdes.block_length()
True
```

sbox()

Return the S-boxes of simplified DES.

EXAMPLES:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sbox = sdes.sbox()
sage: sbox[0]; sbox[1]
(1, 0, 3, 2, 3, 2, 1, 0, 0, 2, 1, 3, 3, 1, 3, 2)
(0, 1, 2, 3, 2, 0, 1, 3, 3, 0, 1, 0, 2, 1, 0, 3)
```

string_to_list(S)

Return a list representation of the binary string `S`.

INPUT:

- `S` – a string of bits

OUTPUT:

A list representation of the string S.

EXAMPLES:

A list representation of a string of bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: S = "0101010110"
sage: sdes.string_to_list(S)
[0, 1, 0, 1, 0, 1, 0, 1, 1, 0]
```

subkey(*K*, *n=1*)

Return the n-th subkey based on the key K.

INPUT:

- K – a 10-bit secret key of this Simplified DES
- n – (default: 1) if n=1 then return the first subkey based on K; if n=2 then return the second subkey. The valid values for n are 1 and 2, since only two subkeys are defined for each secret key in Schaefer's S-DES.

OUTPUT:

The n-th subkey based on the secret key K.

EXAMPLES:

Obtain the first subkey from a secret key:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.subkey(key, n=1)
[1, 0, 1, 0, 0, 1, 0, 0]
```

Obtain the second subkey from a secret key:

```
sage: key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.subkey(key, n=2)
[0, 1, 0, 0, 0, 0, 1, 1]
```

We can also work with strings of bits:

```
sage: K = "1010010010"
sage: L = sdes.string_to_list(K)
sage: sdes.subkey(L, n=1)
[1, 0, 1, 0, 0, 1, 0, 1]
sage: sdes.subkey(sdes.string_to_list("0010010011"), n=2)
[0, 1, 1, 0, 1, 0, 1, 0]
```

switch(*B*)

Interchange the first 4 bits with the last 4 bits in the list B of 8 bits. Let $(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7)$ be a vector of 8 bits, where each $b_i \in \{0, 1\}$. Then the switch function σ is given by

$$\sigma(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (b_4, b_5, b_6, b_7, b_0, b_1, b_2, b_3)$$

INPUT:

- B – list; a block of 8 bits

OUTPUT:

A block of the same dimension, but in which the first 4 bits from B has been switched for the last 4 bits in B.

EXAMPLES:

Interchange the first 4 bits with the last 4 bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 1, 1, 0, 1, 0, 0, 0]
sage: sdes.switch(B)
[1, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.switch([1, 1, 1, 1, 0, 0, 0, 0])
[0, 0, 0, 0, 1, 1, 1, 1]
```

We can also work with a string of bits:

```
sage: S = "11101000"
sage: L = sdes.string_to_list(S)
sage: sdes.switch(L)
[1, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.switch(sdes.string_to_list("11110000"))
[0, 0, 0, 0, 1, 1, 1, 1]
```

CHAPTER

SIX

MINI-AES

A simplified variant of the Advanced Encryption Standard (AES). Note that Mini-AES is for educational purposes only. It is a small-scale version of the AES designed to help beginners understand the basic structure of AES.

AUTHORS:

- Minh Van Nguyen (2009-05): initial version

class sage.crypto.block_cipher.miniaes.**MiniAES**

Bases: sage.structure.sage_object.SageObject

This class implements the Mini Advanced Encryption Standard (Mini-AES) described in [Pha2002]. Note that Phan's Mini-AES is for educational purposes only and is not secure for practical purposes. Mini-AES is a version of the AES with all parameters significantly reduced, but at the same time preserving the structure of AES. The goal of Mini-AES is to allow a beginner to understand the structure of AES, thus laying a foundation for a thorough study of AES. Its goal is as a teaching tool and is different from the *SR* small scale variants of the AES. SR defines a family of parameterizable variants of the AES suitable as a framework for comparing different cryptanalytic techniques that can be brought to bear on the AES.

EXAMPLES:

Encrypt a plaintext:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: P = MS([K("x^3 + x"), K("x^2 + 1"), K("x^2 + x"), K("x^3 + x^2")]); P

[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]
sage: key = MS([K("x^3 + x^2"), K("x^3 + x"), K("x^3 + x^2 + x"), K("x^2 + x + 1
↪")]); key

[    x^3 + x^2       x^3 + x]
[x^3 + x^2 + x   x^2 + x + 1]
sage: C = maes.encrypt(P, key); C

[            x       x^2 + x]
[x^3 + x^2 + x       x^3 + x]
```

Decrypt the result:

```
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt; P
```

(continues on next page)

(continued from previous page)

```
[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]

[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]
sage: plaintxt == P
True
```

We can also work directly with binary strings:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: bin = BinaryStrings()
sage: key = bin.encoding("KE"); key
0100101101000101
sage: P = bin.encoding("Encrypt this secret message!"); P
01000101011011100110001101110010011110010111000001110100001000000111010001101000011010010111001101
sage: C = maes(P, key, algorithm="encrypt"); C
10001000101001101111000001111000010011001110110101000111011011010101001011101111110101100111001111
sage: plaintxt = maes(C, key, algorithm="decrypt")
sage: plaintxt == P
True
```

Now we work with integers n such that $0 \leq n \leq 15$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: P = [n for n in range(16)]; P
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: key = [2, 3, 11, 0]; key
[2, 3, 11, 0]
sage: P = maes.integer_to_binary(P); P
00000001001000110100010101100111100010011010101111001101111011111
sage: key = maes.integer_to_binary(key); key
0010001110110000
sage: C = maes(P, key, algorithm="encrypt"); C
11001000001000111110010101010101010110111001111100010000111000001
sage: plaintxt = maes(C, key, algorithm="decrypt")
sage: plaintxt == P
True
```

Generate some random plaintext and a random secret key. Encrypt the plaintext using that secret key and decrypt the result. Then compare the decrypted plaintext with the original plaintext:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: MS = MatrixSpace(FiniteField(16, "x"), 2, 2)
sage: P = MS.random_element()
sage: key = maes.random_key()
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

GF_to_binary(G)

Return the binary representation of `G`. If `G` is an element of the finite field $\mathbf{F}_{2^4}$, then obtain the binary representation of `G`. If `G` is a list of elements belonging to $\mathbf{F}_{2^4}$, obtain the 4-bit representation of each

element of the list, then concatenate the resulting 4-bit strings into a binary string. If G is a matrix with entries over $\mathbf{F}_{2^4}$, convert each matrix entry to its 4-bit representation, then concatenate the 4-bit strings. The concatenation is performed starting from the top-left corner of the matrix, working across left to right, top to bottom. Each element of $\mathbf{F}_{2^4}$ can be associated with a unique 4-bit string according to the following table:

4-bit string	$\mathbf{F}_{2^4}$	4-bit string	$\mathbf{F}_{2^4}$
0000	0	1000	x^3
0001	1	1001	x^3+1
0010	x	1010	x^3+x
0011	$x+1$	1011	x^3+x+1
0100	x^2	1100	x^3+x^2
0101	x^2+1	1101	x^3+x^2+1
0110	x^2+x	1110	x^3+x^2+x
0111	x^2+x+1	1111	x^3+x^2+x+1

INPUT:

- G – an element of $\mathbf{F}_{2^4}$, a list of elements of $\mathbf{F}_{2^4}$, or a matrix over $\mathbf{F}_{2^4}$

OUTPUT:

- A binary string representation of G.

EXAMPLES:

Obtain the binary representation of all elements of $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: S = Set(K); len(S)  # GF(2^4) has this many elements
16
sage: [maes.GF_to_binary(S[i]) for i in range(len(S))]

[0000,
0001,
0010,
0011,
0100,
0101,
0110,
0111,
1000,
1001,
1010,
1011,
1100,
1101,
1110,
1111]
```

The binary representation of a list of elements belonging to $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: G = [K("x^2 + x + 1"), K("x^3 + x^2"), K("x"), K("x^3 + x + 1"), K("x^3␣
↪+ x^2 + x + 1"), K("x^2 + x"), K("1"), K("x^2 + x + 1")]
```

(continues on next page)

(continued from previous page)

```
sage: maes.GF_to_binary(G)
01111100001010111111011000010111
```

The binary representation of a matrix over $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: G = MS([K("x^3 + x^2"), K("x + 1"), K("x^2 + x + 1"), K("x^3 + x^2 + x
↪")]); G

[    x^3 + x^2         x + 1]
[  x^2 + x + 1 x^3 + x^2 + x]
sage: maes.GF_to_binary(G)
1100001101111110
sage: MS = MatrixSpace(K, 2, 4)
sage: G = MS([K("x^2 + x + 1"), K("x^3 + x^2"), K("x"), K("x^3 + x + 1"), K(
↪"x^3 + x^2 + x + 1"), K("x^2 + x"), K("1"), K("x^2 + x + 1")]); G

[      x^2 + x + 1         x^3 + x^2                 x       x^3 + x + 1]
[x^3 + x^2 + x + 1           x^2 + x                 1       x^2 + x + 1]
sage: maes.GF_to_binary(G)
01111100001010111111011000010111
```

GF_to_integer(G)

Return the integer representation of the finite field element `G`. If `G` is an element of the finite field $\mathbf{F}_{2^4}$, then obtain the integer representation of `G`. If `G` is a list of elements belonging to $\mathbf{F}_{2^4}$, obtain the integer representation of each element of the list, and return the result as a list of integers. If `G` is a matrix with entries over $\mathbf{F}_{2^4}$, convert each matrix entry to its integer representation, and return the result as a list of integers. The resulting list is obtained by starting from the top-left corner of the matrix, working across left to right, top to bottom. Each element of $\mathbf{F}_{2^4}$ can be associated with a unique integer according to the following table:

integer	$\mathbf{F}_{2^4}$	integer	$\mathbf{F}_{2^4}$
0	0	8	x^3
1	1	9	x^3+1
2	x	10	x^3+x
3	$x+1$	11	x^3+x+1
4	x^2	12	x^3+x^2
5	x^2+1	13	x^3+x^2+1
6	x^2+x	14	x^3+x^2+x
7	x^2+x+1	15	x^3+x^2+x+1

INPUT:

- `G` – an element of $\mathbf{F}_{2^4}$, a list of elements belonging to $\mathbf{F}_{2^4}$, or a matrix over $\mathbf{F}_{2^4}$

OUTPUT:

- The integer representation of `G`.

EXAMPLES:

Obtain the integer representation of all elements of $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: S = Set(K); len(S)  # GF(2^4) has this many elements
16
sage: [maes.GF_to_integer(S[i]) for i in range(len(S))]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

The integer representation of a list of elements belonging to $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: G = [K("x^2 + x + 1"), K("x^3 + x^2"), K("x"), K("x^3 + x + 1"), K("x^3␣
↪+ x^2 + x + 1"), K("x^2 + x"), K("1"), K("x^2 + x + 1")]
sage: maes.GF_to_integer(G)
[7, 12, 2, 11, 15, 6, 1, 7]
```

The integer representation of a matrix over $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: G = MS([K("x^3 + x^2"), K("x + 1"), K("x^2 + x + 1"), K("x^3 + x^2 + x
↪")]); G

[    x^3 + x^2         x + 1]
[  x^2 + x + 1 x^3 + x^2 + x]
sage: maes.GF_to_integer(G)
[12, 3, 7, 14]
sage: MS = MatrixSpace(K, 2, 4)
sage: G = MS([K("x^2 + x + 1"), K("x^3 + x^2"), K("x"), K("x^3 + x + 1"), K(
↪"x^3 + x^2 + x + 1"), K("x^2 + x"), K("1"), K("x^2 + x + 1")]); G

[      x^2 + x + 1         x^3 + x^2                 x       x^3 + x + 1]
[x^3 + x^2 + x + 1           x^2 + x                 1       x^2 + x + 1]
sage: maes.GF_to_integer(G)
[7, 12, 2, 11, 15, 6, 1, 7]
```

add_key (*block*, *rkey*)

Return the matrix addition of `block` and `rkey`. Both `block` and `rkey` are 2×2 matrices over the finite field $\mathbf{F}_{2^4}$. This method just return the matrix addition of these two matrices.

INPUT:

- `block` – a 2×2 matrix with entries over $\mathbf{F}_{2^4}$
- `rkey` – a round key; a 2×2 matrix with entries over $\mathbf{F}_{2^4}$

OUTPUT:

- The matrix addition of `block` and `rkey`.

EXAMPLES:

We can work with elements of $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
```

(continues on next page)

(continued from previous page)

```
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: D = MS([ [K("x^3 + x^2 + x + 1"), K("x^3 + x")], [K("0"), K("x^3 + x^2
↪")] ]); D

[x^3 + x^2 + x + 1           x^3 + x]
[                0         x^3 + x^2]
sage: k = MS([ [K("x^2 + 1"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")]␣
↪]); k

[          x^2 + 1 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: maes.add_key(D, k)

[  x^3 + x   x^2 + 1]
[    x + 1 x^3 + x^2]
```

Or work with binary strings:

```
sage: bin = BinaryStrings()
sage: B = bin.encoding("We"); B
0101011101100101
sage: B = MS(maes.binary_to_GF(B)); B

[    x^2 + 1 x^2 + x + 1]
[    x^2 + x     x^2 + 1]
sage: key = bin.encoding("KY"); key
0100101101011001
sage: key = MS(maes.binary_to_GF(key)); key

[        x^2 x^3 + x + 1]
[    x^2 + 1     x^3 + 1]
sage: maes.add_key(B, key)

[        1 x^3 + x^2]
[    x + 1 x^3 + x^2]
```

We can also work with integers n such that $0 \leq n \leq 15$:

```
sage: N = [2, 3, 5, 7]; N
[2, 3, 5, 7]
sage: key = [9, 11, 13, 15]; key
[9, 11, 13, 15]
sage: N = MS(maes.integer_to_GF(N)); N

[          x       x + 1]
[    x^2 + 1 x^2 + x + 1]
sage: key = MS(maes.integer_to_GF(key)); key

[          x^3 + 1       x^3 + x + 1]
[    x^3 + x^2 + 1 x^3 + x^2 + x + 1]
sage: maes.add_key(N, key)

[x^3 + x + 1         x^3]
[        x^3         x^3]
```

binary_to_GF(B)

Return a list of elements of $\mathbf{F}_{2^4}$ that represents the binary string `B`. The number of bits in `B` must be greater than zero and a multiple of 4. Each nibble (or 4-bit string) is uniquely associated with an element of $\mathbf{F}_{2^4}$ as specified by the following table:

4-bit string	$\mathbf{F}_{2^4}$	4-bit string	$\mathbf{F}_{2^4}$
0000	0	1000	x^3
0001	1	1001	x^3+1
0010	x	1010	x^3+x
0011	$x+1$	1011	x^3+x+1
0100	x^2	1100	x^3+x^2
0101	x^2+1	1101	x^3+x^2+1
0110	x^2+x	1110	x^3+x^2+x
0111	x^2+x+1	1111	x^3+x^2+x+1

INPUT:

- `B` – a binary string, where the number of bits is positive and a multiple of 4

OUTPUT:

- A list of elements of the finite field $\mathbf{F}_{2^4}$ that represent the binary string `B`.

EXAMPLES:

Obtain all the elements of the finite field $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: bin = BinaryStrings()
sage: B = bin(
↪"0000000100100011010001010110011110001001101010111100110111101111")
sage: maes.binary_to_GF(B)

[0,
1,
x,
x + 1,
x^2,
x^2 + 1,
x^2 + x,
x^2 + x + 1,
x^3,
x^3 + 1,
x^3 + x,
x^3 + x + 1,
x^3 + x^2,
x^3 + x^2 + 1,
x^3 + x^2 + x,
x^3 + x^2 + x + 1]
```

binary_to_integer(*B*)

Return a list of integers representing the binary string `B`. The number of bits in `B` must be greater than zero and a multiple of 4. Each nibble (or 4-bit string) is uniquely associated with an integer as specified by the

following table:

4-bit string	integer	4-bit string	integer
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

INPUT:

- `B` – a binary string, where the number of bits is positive and a multiple of 4

OUTPUT:

- A list of integers that represent the binary string `B`.

EXAMPLES:

Obtain the integer representation of every 4-bit string:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: bin = BinaryStrings()
sage: B = bin(
↪"0000000100100011010001010110011110001001101010111100110111101111")
sage: maes.binary_to_integer(B)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

block_length()

Return the block length of Phan's Mini-AES block cipher. A key in Phan's Mini-AES is a block of 16 bits. Each nibble of a key can be considered as an element of the finite field $\mathbf{F}_{2^4}$. Therefore the key consists of four elements from $\mathbf{F}_{2^4}$.

OUTPUT:

- The block (or key) length in number of bits.

EXAMPLES:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.block_length()
16
```

decrypt(*C*, *key*)

Use Phan's Mini-AES to decrypt the ciphertext `C` with the secret key `key`. Both `C` and `key` must be 2×2 matrices over the finite field $\mathbf{F}_{2^4}$. Let γ denote the operation of nibble-sub, π denote shift-row, θ denote mix-column, and σ_{K_i} denote add-key with the round key K_i. Then decryption D using Phan's Mini-AES is the function composition

$$D = \sigma_{K_0} \circ \gamma^{-1} \circ \pi \circ \theta \circ \sigma_{K_1} \circ \gamma^{-1} \circ \pi \circ \sigma_{K_2}$$

where γ^{-1} is the nibble-sub operation that uses the S-box for decryption, and the order of execution is from right to left.

INPUT:

- `C` – a ciphertext block; must be a 2×2 matrix over the finite field $\mathbf{F}_{2^4}$
- `key` – a secret key for this Mini-AES block cipher; must be a 2×2 matrix over the finite field $\mathbf{F}_{2^4}$

OUTPUT:

- The plaintext corresponding to `C`.

EXAMPLES:

We encrypt a plaintext, decrypt the ciphertext, then compare the decrypted plaintext with the original plaintext. Here we work with elements of $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: P = MS([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ]);␣
↪P

[  x^3 + 1   x^2 + x]
[x^3 + x^2     x + 1]
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0
↪")] ]); key

[        x^3 + x^2 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: C = maes.encrypt(P, key); C

[x^2 + x + 1   x^3 + x^2]
[          x     x^2 + x]
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt; P

[  x^3 + 1   x^2 + x]
[x^3 + x^2     x + 1]

[  x^3 + 1   x^2 + x]
[x^3 + x^2     x + 1]
sage: plaintxt == P
True
```

But we can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: P = bin.encoding("de"); P
0110010001100101
sage: P = MS(maes.binary_to_GF(P)); P

[x^2 + x     x^2]
[x^2 + x x^2 + 1]
sage: key = bin.encoding("ke"); key
0110101101100101
sage: key = MS(maes.binary_to_GF(key)); key

[    x^2 + x x^3 + x + 1]
[    x^2 + x     x^2 + 1]
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
```

(continues on next page)

(continued from previous page)

```
sage: plaintxt == P
True
```

Here we work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [3, 5, 7, 14]; P
[3, 5, 7, 14]
sage: key = [2, 6, 7, 8]; key
[2, 6, 7, 8]
sage: P = MS(maes.integer_to_GF(P)); P

[        x + 1        x^2 + 1]
[  x^2 + x + 1 x^3 + x^2 + x]
sage: key = MS(maes.integer_to_GF(key)); key

[          x      x^2 + x]
[x^2 + x + 1          x^3]
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

encrypt(*P*, *key*)

Use Phan's Mini-AES to encrypt the plaintext `P` with the secret key `key`. Both `P` and `key` must be 2×2 matrices over the finite field $\mathbf{F}_{2^4}$. Let γ denote the operation of nibble-sub, π denote shift-row, θ denote mix-column, and σ_{K_i} denote add-key with the round key K_i. Then encryption E using Phan's Mini-AES is the function composition

$$E = \sigma_{K_2} \circ \pi \circ \gamma \circ \sigma_{K_1} \circ \theta \circ \pi \circ \gamma \circ \sigma_{K_0}$$

where the order of execution is from right to left. Note that γ is the nibble-sub operation that uses the S-box for encryption.

INPUT:

- `P` – a plaintext block; must be a 2×2 matrix over the finite field $\mathbf{F}_{2^4}$
- `key` – a secret key for this Mini-AES block cipher; must be a 2×2 matrix over the finite field $\mathbf{F}_{2^4}$

OUTPUT:

- The ciphertext corresponding to `P`.

EXAMPLES:

Here we work with elements of $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: P = MS([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ]);␣
↪P

[  x^3 + 1   x^2 + x]
[x^3 + x^2     x + 1]
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0
↪")] ]); key
```

(continues on next page)

(continued from previous page)

```
[         x^3 + x^2 x^3 + x^2 + x + 1]
[             x + 1                 0]
sage: maes.encrypt(P, key)

[x^2 + x + 1   x^3 + x^2]
[           x     x^2 + x]
```

But we can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: P = bin.encoding("de"); P
0110010001100101
sage: P = MS(maes.binary_to_GF(P)); P

[x^2 + x     x^2]
[x^2 + x x^2 + 1]
sage: key = bin.encoding("ke"); key
0110101101100101
sage: key = MS(maes.binary_to_GF(key)); key

[    x^2 + x x^3 + x + 1]
[    x^2 + x     x^2 + 1]
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

Now we work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [1, 5, 8, 12]; P
[1, 5, 8, 12]
sage: key = [5, 9, 15, 0]; key
[5, 9, 15, 0]
sage: P = MS(maes.integer_to_GF(P)); P

[        1   x^2 + 1]
[      x^3 x^3 + x^2]
sage: key = MS(maes.integer_to_GF(key)); key

[          x^2 + 1           x^3 + 1]
[x^3 + x^2 + x + 1                 0]
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

integer_to_GF(*N*)

Return the finite field representation of `N`. If N is an integer such that $0 \leq N \leq 15$, return the element of $\mathbf{F}_{2^4}$ that represents `N`. If `N` is a list of integers each of which is ≥ 0 and ≤ 15, then obtain the element of $\mathbf{F}_{2^4}$ that represents each such integer, and return a list of such finite field representations. Each integer between 0 and 15, inclusive, can be associated with a unique element of $\mathbf{F}_{2^4}$ according to the following

table:

integer	$\mathbf{F}_{2^4}$	integer	$\mathbf{F}_{2^4}$
0	0	8	x^3
1	1	9	x^3+1
2	x	10	x^3+x
3	$x+1$	11	x^3+x+1
4	x^2	12	x^3+x^2
5	x^2+1	13	x^3+x^2+1
6	x^2+x	14	x^3+x^2+x
7	x^2+x+1	15	x^3+x^2+x+1

INPUT:

- N – a non-negative integer less than or equal to 15, or a list of such integers

OUTPUT:

- Elements of the finite field $\mathbf{F}_{2^4}$.

EXAMPLES:

Obtain the element of $\mathbf{F}_{2^4}$ representing an integer n, where $0 \leq n \leq 15$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.integer_to_GF(0)
0
sage: maes.integer_to_GF(2)
x
sage: maes.integer_to_GF(7)
x^2 + x + 1
```

Obtain the finite field elements corresponding to all non-negative integers less than or equal to 15:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: lst = [n for n in range(16)]; lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: maes.integer_to_GF(lst)

[0,
1,
x,
x + 1,
x^2,
x^2 + 1,
x^2 + x,
x^2 + x + 1,
x^3,
x^3 + 1,
x^3 + x,
x^3 + x + 1,
x^3 + x^2,
x^3 + x^2 + 1,
x^3 + x^2 + x,
x^3 + x^2 + x + 1]
```

integer_to_binary(*N*)

Return the binary representation of N. If N is an integer such that $0 \leq N \leq 15$, return the binary representation of N. If N is a list of integers each of which is ≥ 0 and ≤ 15, then obtain the binary representation of

each integer, and concatenate the individual binary representations into a single binary string. Each integer between 0 and 15, inclusive, can be associated with a unique 4-bit string according to the following table:

4-bit string	integer	4-bit string	integer
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

INPUT:

- `N` – a non-negative integer less than or equal to 15, or a list of such integers

OUTPUT:

- A binary string representing `N`.

EXAMPLES:

The binary representations of all integers between 0 and 15, inclusive:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: lst = [n for n in range(16)]; lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: maes.integer_to_binary(lst)
0000000100100011010001010110011110001001101010111100110111101111
```

The binary representation of an integer between 0 and 15, inclusive:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.integer_to_binary(3)
0011
sage: maes.integer_to_binary(5)
0101
sage: maes.integer_to_binary(7)
0111
```

mix_column(*block*)

Return the matrix multiplication of `block` with a constant matrix. The constant matrix is

$$\begin{bmatrix} x+1 & x \\ x & x+1 \end{bmatrix}$$

If the input block is

$$\begin{bmatrix} c_0 & c_2 \\ c_1 & c_3 \end{bmatrix}$$

then the output block is

$$\begin{bmatrix} d_0 & d_2 \\ d_1 & d_3 \end{bmatrix} = \begin{bmatrix} x+1 & x \\ x & x+1 \end{bmatrix} \begin{bmatrix} c_0 & c_2 \\ c_1 & c_3 \end{bmatrix}$$

INPUT:

- `block` – a 2×2 matrix with entries over $\mathbf{F}_{2^4}$

OUTPUT:

- A 2×2 matrix resulting from multiplying the above constant matrix with the input matrix `block`.

EXAMPLES:

Here we work with elements of $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([ [K("x^2 + x + 1"), K("x^3 + x^2 + 1")], [K("x^3"), K("x")] ])
sage: maes.mix_column(mat)

[          x^3 + x                   0]
[          x^2 + 1 x^3 + x^2 + x + 1]
```

Multiplying by the identity matrix should leave the constant matrix unchanged:

```
sage: eye = MS([ [K("1"), K("0")], [K("0"), K("1")] ])
sage: maes.mix_column(eye)

[x + 1     x]
[    x x + 1]
```

We can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: B = bin.encoding("rT"); B
0111001001010100
sage: B = MS(maes.binary_to_GF(B)); B

[x^2 + x + 1           x]
[    x^2 + 1         x^2]
sage: maes.mix_column(B)

[        x + 1 x^3 + x^2 + x]
[            1           x^3]
```

We can also work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [10, 5, 2, 7]; P
[10, 5, 2, 7]
sage: P = MS(maes.integer_to_GF(P)); P

[    x^3 + x     x^2 + 1]
[          x x^2 + x + 1]
sage: maes.mix_column(P)

[x^3 + 1       1]
[      1   x + 1]
```

nibble_sub(*block*, *algorithm='encrypt'*)

Substitute a nibble (or a block of 4 bits) using the following S-box:

Input	Output	Input	Output
0000	1110	1000	0011
0001	0100	1001	1010
0010	1101	1010	0110
0011	0001	1011	1100
0100	0010	1100	0101
0101	1111	1101	1001
0110	1011	1110	0000
0111	1000	1111	0111

The values in the above S-box are taken from the first row of the first S-box of the Data Encryption Standard (DES). Each nibble can be thought of as an element of the finite field $\mathbf{F}_{2^4}$ of 16 elements. Thus in terms of $\mathbf{F}_{2^4}$, the S-box can also be specified as:

Input	Output
0	$x^3 + x^2 + x$
1	x^2
x	$x^3 + x^2 + 1$
$x + 1$	1
x^2	x
$x^2 + 1$	$x^3 + x^2 + x + 1$
$x^2 + x$	$x^3 + x + 1$
$x^2 + x + 1$	x^3
x^3	$x + 1$
$x^3 + 1$	$x^3 + x$
$x^3 + x$	$x^2 + x$
$x^3 + x + 1$	$x^3 + x^2$
$x^3 + x^2$	$x^2 + 1$
$x^3 + x^2 + 1$	$x^3 + 1$
$x^3 + x^2 + x$	0
$x^3 + x^2 + x + 1$	$x^2 + x + 1$

Note that the above S-box is used for encryption. The S-box for decryption is obtained from the above S-box by reversing the role of the Input and Output columns. Thus the previous Input column for encryption now becomes the Output column for decryption, and the previous Output column for encryption is now the Input column for decryption. The S-box used for decryption can be specified as:

Input	Output
0	$x^3 + x^2 + x$
1	$x + 1$
x	x^2
$x + 1$	x^3
x^2	1
$x^2 + 1$	$x^3 + x^2$
$x^2 + x$	$x^3 + x$
$x^2 + x + 1$	$x^3 + x^2 + x + 1$
x^3	$x^2 + x + 1$
$x^3 + 1$	$x^3 + x^2 + 1$
$x^3 + x$	$x^3 + 1$
$x^3 + x + 1$	$x^2 + x$
$x^3 + x^2$	$x^3 + x + 1$
$x^3 + x^2 + 1$	x
$x^3 + x^2 + x$	0
$x^3 + x^2 + x + 1$	$x^2 + 1$

INPUT:

- `block` – a 2×2 matrix with entries over $\mathbf{F}_{2^4}$
- `algorithm` – (default: `"encrypt"`) a string; a flag to signify whether this nibble-sub operation is used for encryption or decryption. The encryption flag is `"encrypt"` and the decryption flag is `"decrypt"`.

OUTPUT:

- A 2×2 matrix resulting from applying an S-box on entries of the 2×2 matrix `block`.

EXAMPLES:

Here we work with elements of the finite field $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")], [K("x^2 + x + 1"), K("x^3 +␣
↪x")]])
sage: maes.nibble_sub(mat, algorithm="encrypt")

[  x^2 + x + 1 x^3 + x^2 + x]
[          x^3       x^2 + x]
```

But we can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: B = bin.encoding("bi"); B
0110001001101001
sage: B = MS(maes.binary_to_GF(B)); B

[x^2 + x       x]
[x^2 + x x^3 + 1]
sage: maes.nibble_sub(B, algorithm="encrypt")

[  x^3 + x + 1 x^3 + x^2 + 1]
[  x^3 + x + 1       x^3 + x]
sage: maes.nibble_sub(B, algorithm="decrypt")

[      x^3 + x           x^2]
[      x^3 + x x^3 + x^2 + 1]
```

Here we work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [2, 6, 8, 14]; P
[2, 6, 8, 14]
sage: P = MS(maes.integer_to_GF(P)); P

[            x       x^2 + x]
[          x^3 x^3 + x^2 + x]
sage: maes.nibble_sub(P, algorithm="encrypt")

[x^3 + x^2 + 1   x^3 + x + 1]
[        x + 1             0]
sage: maes.nibble_sub(P, algorithm="decrypt")
```

(continues on next page)

(continued from previous page)

```
[         x^2      x^3 + x]
[x^2 + x + 1             0]
```

random_key()

A random key within the key space of this Mini-AES block cipher. Like the AES, Phan's Mini-AES is a symmetric-key block cipher. A Mini-AES key is a block of 16 bits, or a 2×2 matrix with entries over the finite field $\mathbf{F}_{2^4}$. Thus the number of possible keys is $2^{16} = 16^4$.

OUTPUT:

- A 2×2 matrix over the finite field $\mathbf{F}_{2^4}$, used as a secret key for this Mini-AES block cipher.

EXAMPLES:

Each nibble of a key is an element of the finite field $\mathbf{F}_{2^4}$:

```
sage: K = FiniteField(16, "x")
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: key = maes.random_key()
sage: [key[i][j] in K for i in range(key.nrows()) for j in range(key.ncols())]
[True, True, True, True]
```

Generate a random key, then perform encryption and decryption using that key:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: key = maes.random_key()
sage: P = MS.random_element()
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

round_key(*key*, *n*)

Return the round key for round `n`. Phan's Mini-AES is defined to have two rounds. The round key K_0 is generated and used prior to the first round, with round keys K_1 and K_2 being used in rounds 1 and 2 respectively. In total, there are three round keys, each generated from the secret key `key`.

INPUT:

- `key` – the secret key
- `n` – non-negative integer; the round number

OUTPUT:

- The n-th round key.

EXAMPLES:

Obtaining the round keys from the secret key:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0
→")] ])
```

(continues on next page)

(continued from previous page)

```
sage: maes.round_key(key, 0)

[        x^3 + x^2 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: key

[        x^3 + x^2 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: maes.round_key(key, 1)

[            x + 1 x^3 + x^2 + x + 1]
[                0 x^3 + x^2 + x + 1]
sage: maes.round_key(key, 2)

[x^2 + x x^3 + 1]
[x^2 + x x^2 + x]
```

sbox()

Return the S-box of Mini-AES.

EXAMPLES:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.sbox()
(14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7)
```

shift_row(*block*)

Rotate each row of `block` to the left by different nibble amounts. The first or zero-th row is left unchanged, while the second or row one is rotated left by one nibble. This has the effect of only interchanging the nibbles in the second row. Let b_0, b_1, b_2, b_3 be four nibbles arranged as the following 2×2 matrix

$$\begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix}$$

Then the operation of shift-row is the mapping

$$\begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} \longmapsto \begin{bmatrix} b_0 & b_2 \\ b_3 & b_1 \end{bmatrix}$$

INPUT:

- `block` – a 2×2 matrix with entries over $\mathbf{F}_{2^4}$

OUTPUT:

- A 2×2 matrix resulting from applying shift-row on `block`.

EXAMPLES:

Here we work with elements of the finite field $\mathbf{F}_{2^4}$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")], [K("x^2 + x + 1"), K("x^3 +␣
↪x")]])
sage: maes.shift_row(mat)
```

(continues on next page)

(continued from previous page)

```
[x^3 + x^2 + x + 1                 0]
[          x^3 + x       x^2 + x + 1]
sage: mat

[x^3 + x^2 + x + 1                 0]
[      x^2 + x + 1           x^3 + x]
```

But we can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: B = bin.encoding("Qt"); B
0101000101110100
sage: B = MS(maes.binary_to_GF(B)); B

[    x^2 + 1           1]
[x^2 + x + 1         x^2]
sage: maes.shift_row(B)

[    x^2 + 1           1]
[        x^2 x^2 + x + 1]
```

Here we work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [3, 6, 9, 12]; P
[3, 6, 9, 12]
sage: P = MS(maes.integer_to_GF(P)); P

[    x + 1   x^2 + x]
[  x^3 + 1 x^3 + x^2]
sage: maes.shift_row(P)

[    x + 1   x^2 + x]
[x^3 + x^2   x^3 + 1]
```

CHAPTER

SEVEN

BLUM-GOLDWASSER PROBABILISTIC ENCRYPTION

The Blum-Goldwasser probabilistic public-key encryption scheme. This scheme was originally described in [BG1985]. See also section 8.7.2 of [MvOV1996] and the Wikipedia article Blum-Goldwasser_cryptosystem on this scheme.

AUTHORS:

- Mike Hogan and David Joyner (2009-9-19): initial procedural version released as public domain software.
- Minh Van Nguyen (2009-12): integrate into Sage as a class and relicense under the GPLv2+. Complete rewrite of the original version to follow the description contained in [MvOV1996].

class sage.crypto.public_key.blum_goldwasser.**BlumGoldwasser**

Bases: *sage.crypto.cryptosystem.PublicKeyCryptosystem*

The Blum-Goldwasser probabilistic public-key encryption scheme.

The Blum-Goldwasser encryption and decryption algorithms as described in *encrypt()* and *decrypt()*, respectively, make use of the least significant bit of a binary string. A related concept is the k least significant bits of a binary string. For example, given a positive integer n, let $b = b_0 b_1 \cdots b_{m-1}$ be the binary representation of n so that b is a binary string of length m. Then the least significant bit of n is b_{m-1}. If $0 < k \leq m$, then the k least significant bits of n are $b_{m-1-k} b_{m-k} \cdots b_{m-1}$. The least significant bit of an integer is also referred to as its parity bit, because this bit determines whether the integer is even or odd. In the following example, we obtain the least significant bit of an integer:

```
sage: n = 123
sage: b = n.binary(); b
'1111011'
sage: n % 2
1
sage: b[-1]
'1'
```

Now find the 4 least significant bits of the integer $n = 123$:

```
sage: b
'1111011'
sage: b[-4:]
'1011'
```

The last two examples could be worked through as follows:

```
sage: from sage.crypto.util import least_significant_bits
sage: least_significant_bits(123, 1)
[1]
sage: least_significant_bits(123, 4)
[1, 0, 1, 1]
```

EXAMPLES:

The following encryption/decryption example is taken from Example 8.57, pages 309–310 of [MvOV1996]:

```
sage: from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
sage: bg = BlumGoldwasser(); bg
The Blum-Goldwasser public-key encryption scheme.
sage: p = 499; q = 547
sage: pubkey = bg.public_key(p, q); pubkey
272953
sage: prikey = bg.private_key(p, q); prikey
(499, 547, -57, 52)
sage: P = "10011100000100001100"
sage: C = bg.encrypt(P, pubkey, seed=159201); C
([[0, 0, 1, 0], [0, 0, 0, 0], [1, 1, 0, 0], [1, 1, 1, 0], [0, 1, 0, 0]], 139680)
sage: M = bg.decrypt(C, prikey); M
[[1, 0, 0, 1], [1, 1, 0, 0], [0, 0, 0, 1], [0, 0, 0, 0], [1, 1, 0, 0]]
sage: M = "".join(str(x) for x in flatten(M)); M
'10011100000100001100'
sage: M == P
True
```

Generate a pair of random public/private keys. Use the public key to encrypt a plaintext. Then decrypt the resulting ciphertext using the private key. Finally, compare the decrypted message with the original plaintext.

```
sage: from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
sage: from sage.crypto.util import bin_to_ascii
sage: bg = BlumGoldwasser()
sage: pubkey, prikey = bg.random_key(10**4, 10**6)
sage: P = "A fixed plaintext."
sage: C = bg.encrypt(P, pubkey)
sage: M = bg.decrypt(C, prikey)
sage: bin_to_ascii(flatten(M)) == P
True
```

If (p, q, a, b) is a private key, then $n = pq$ is the corresponding public key. Furthermore, we have $\gcd(p, q) = ap + bq = 1$.

```
sage: p, q, a, b = prikey
sage: pubkey == p * q
True
sage: gcd(p, q) == a*p + b*q == 1
True
```

decrypt(C, K)

Apply the Blum-Goldwasser scheme to decrypt the ciphertext `C` using the private key `K`.

INPUT:

- `C` – a ciphertext resulting from encrypting a plaintext using the Blum-Goldwasser encryption algorithm. The ciphertext C must be of the form $C = (c_1, c_2, \ldots, c_t, x_{t+1})$. Each c_i is a sub-block of binary string and x_{t+1} is the result of the $t + 1$-th iteration of the Blum-Blum-Shub algorithm.
- `K` – a private key (p, q, a, b) where p and q are distinct Blum primes and $\gcd(p, q) = ap + bq = 1$.

OUTPUT:

- The plaintext resulting from decrypting the ciphertext `C` using the Blum-Goldwasser decryption algorithm.

ALGORITHM:

The Blum-Goldwasser decryption algorithm is described in Algorithm 8.56, page 309 of [MvOV1996]. The algorithm works as follows:

1. Let C be the ciphertext $C = (c_1, c_2, \ldots, c_t, x_{t+1})$. Then t is the number of ciphertext sub-blocks and h is the length of each binary string sub-block c_i.
2. Let (p, q, a, b) be the private key whose corresponding public key is $n = pq$. Note that $\gcd(p, q) = ap + bq = 1$.
3. Compute $d_1 = ((p + 1)/4)^{t+1} \bmod (p - 1)$.
4. Compute $d_2 = ((q + 1)/4)^{t+1} \bmod (q - 1)$.
5. Let $u = x_{t+1}^{d_1} \bmod p$.
6. Let $v = x_{t+1}^{d_2} \bmod q$.
7. Compute $x_0 = vap + ubq \bmod n$.
8. For i from 1 to t, do:
   (a) Compute $x_i = x_{t-1}^2 \bmod n$.
   (b) Let p_i be the h least significant bits of x_i.
   (c) Compute $m_i = p_i \oplus c_i$.
9. The plaintext is $m = m_1 m_2 \cdots m_t$.

EXAMPLES:

The following decryption example is taken from Example 8.57, pages 309–310 of [MvOV1996]. Here we decrypt a binary string:

```
sage: from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
sage: bg = BlumGoldwasser()
sage: p = 499; q = 547
sage: C = ([[0, 0, 1, 0], [0, 0, 0, 0], [1, 1, 0, 0], [1, 1, 1, 0], [0, 1, 0,␣
↪0]], 139680)
sage: K = bg.private_key(p, q); K
(499, 547, -57, 52)
sage: P = bg.decrypt(C, K); P
[[1, 0, 0, 1], [1, 1, 0, 0], [0, 0, 0, 1], [0, 0, 0, 0], [1, 1, 0, 0]]
```

Convert the plaintext sub-blocks into a binary string:

```
sage: bin = BinaryStrings()
sage: bin(flatten(P))
10011100000100001100
```

Decrypt a longer ciphertext and convert the resulting plaintext into an ASCII string:

```
sage: from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
sage: from sage.crypto.util import bin_to_ascii
sage: bg = BlumGoldwasser()
sage: p = 78307; q = 412487
sage: K = bg.private_key(p, q)
sage: C = ([[1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0], \
....: [1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1], \
....: [0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0], \
....: [0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1], \
....: [1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0], \
```

(continues on next page)

(continued from previous page)

```
....: [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1], \
....: [1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0], \
....: [1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1], \
....: [0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0], \
....: [1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1], \
....: [1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1], \
....: [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0], \
....: [0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1]], 3479653279)
sage: P = bg.decrypt(C, K)
sage: bin_to_ascii(flatten(P))
'Blum-Goldwasser encryption'
```

encrypt(*P*, *K*, *seed=None*)

Apply the Blum-Goldwasser scheme to encrypt the plaintext `P` using the public key `K`.

INPUT:

- `P` – a non-empty string of plaintext. The string `""` is an empty string, whereas `" "` is a string consisting of one white space character. The plaintext can be a binary string or a string of ASCII characters. Where `P` is an ASCII string, then `P` is first encoded as a binary string prior to encryption.
- `K` – a public key, which is the product of two Blum primes.
- `seed` – (default: `None`) if p and q are Blum primes and $n = pq$ is a public key, then `seed` is a quadratic residue in the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$. If `seed=None`, then the function would generate its own random quadratic residue in $(\mathbf{Z}/n\mathbf{Z})^*$. Where a value for `seed` is provided, it is your responsibility to ensure that the seed is a quadratic residue in the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$.

OUTPUT:

- The ciphertext resulting from encrypting `P` using the public key `K`. The ciphertext C is of the form $C = (c_1, c_2, \ldots, c_t, x_{t+1})$. Each c_i is a sub-block of binary string and x_{t+1} is the result of the $t+1$-th iteration of the Blum-Blum-Shub algorithm.

ALGORITHM:

The Blum-Goldwasser encryption algorithm is described in Algorithm 8.56, page 309 of [MvOV1996]. The algorithm works as follows:

1. Let n be a public key, where $n = pq$ is the product of two distinct Blum primes p and q.
2. Let $k = \lfloor \log_2(n) \rfloor$ and $h = \lfloor \log_2(k) \rfloor$.
3. Let $m = m_1 m_2 \cdots m_t$ be the message (plaintext) where each m_i is a binary string of length h.
4. Choose a random seed x_0, which is a quadratic residue in the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$. That is, choose a random $r \in (\mathbf{Z}/n\mathbf{Z})^*$ and compute $x_0 = r^2 \bmod n$.
5. For i from 1 to t, do:
   (a) Let $x_i = x_{i-1}^2 \bmod n$.
   (b) Let p_i be the h least significant bits of x_i.
   (c) Let $c_i = p_i \oplus m_i$.
6. Compute $x_{t+1} = x_t^2 \bmod n$.
7. The ciphertext is $c = (c_1, c_2, \ldots, c_t, x_{t+1})$.

The value h in the algorithm is the sub-block length. If the binary string representing the message cannot be divided into blocks of length h each, then other sub-block lengths would be used instead. The sub-block lengths to fall back on are in the following order: 16, 8, 4, 2, 1.

EXAMPLES:

The following encryption example is taken from Example 8.57, pages 309–310 of [MvOV1996]. Here, we encrypt a binary string:

```
sage: from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
sage: bg = BlumGoldwasser()
sage: p = 499; q = 547; n = p * q
sage: P = "1001110000100001100"
sage: C = bg.encrypt(P, n, seed=159201); C
([[0, 0, 1, 0], [0, 0, 0, 0], [1, 1, 0, 0], [1, 1, 1, 0], [0, 1, 0, 0]],␣
↪139680)
```

Convert the ciphertext sub-blocks into a binary string:

```
sage: bin = BinaryStrings()
sage: bin(flatten(C[0]))
00100000110011100100
```

Now encrypt an ASCII string. The result is random; no seed is provided to the encryption function so the function generates its own random seed:

```
sage: from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
sage: bg = BlumGoldwasser()
sage: K = 32300619509
sage: P = "Blum-Goldwasser encryption"
sage: bg.encrypt(P, K)  # random
([[1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0], \
[1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1], \
[0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0], \
[0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1], \
[1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0], \
[0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1], \
[1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0], \
[1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1], \
[0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0], \
[1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1], \
[1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1], \
[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0], \
[0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1]], 3479653279)
```

private_key(*p*, *q*)

Return the Blum-Goldwasser private key corresponding to the distinct Blum primes `p` and `q`.

INPUT:

- `p` – a Blum prime.
- `q` – a Blum prime.

OUTPUT:

- The Blum-Goldwasser private key (p, q, a, b) where $\gcd(p, q) = ap + bq = 1$.

Both `p` and `q` must be distinct Blum primes. Let p be a positive prime. Then p is a Blum prime if p is congruent to 3 modulo 4, i.e. $p \equiv 3 \pmod 4$.

EXAMPLES:

Obtain two distinct Blum primes and compute the Blum-Goldwasser private key corresponding to those two Blum primes:

```
sage: from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
sage: from sage.crypto.util import is_blum_prime
sage: bg = BlumGoldwasser()
sage: P = primes_first_n(10); P
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: [is_blum_prime(_) for _ in P]
[False, True, False, True, True, False, False, True, True, False]
sage: bg.private_key(19, 23)
(19, 23, -6, 5)
```

Choose two distinct random Blum primes, compute the Blum-Goldwasser private key corresponding to those two primes, and test that the resulting private key (p, q, a, b) satisfies $\gcd(p, q) = ap + bq = 1$:

```
sage: from sage.crypto.util import random_blum_prime
sage: p = random_blum_prime(10**4, 10**5)
sage: q = random_blum_prime(10**4, 10**5)
sage: while q == p:
....:     q = random_blum_prime(10**4, 10**5)
sage: p, q, a, b = bg.private_key(p, q)
sage: gcd(p, q) == a*p + b*q == 1
True
```

public_key(p, q)

Return the Blum-Goldwasser public key corresponding to the distinct Blum primes `p` and `q`.

INPUT:

- `p` – a Blum prime.
- `q` – a Blum prime.

OUTPUT:

- The Blum-Goldwasser public key $n = pq$.

Both `p` and `q` must be distinct Blum primes. Let p be a positive prime. Then p is a Blum prime if p is congruent to 3 modulo 4, i.e. $p \equiv 3 \pmod 4$.

EXAMPLES:

Obtain two distinct Blum primes and compute the Blum-Goldwasser public key corresponding to those two Blum primes:

```
sage: from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
sage: from sage.crypto.util import is_blum_prime
sage: bg = BlumGoldwasser()
sage: P = primes_first_n(10); P
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: [is_blum_prime(_) for _ in P]
[False, True, False, True, True, False, False, True, True, False]
sage: bg.public_key(3, 7)
21
```

Choose two distinct random Blum primes, compute the Blum-Goldwasser public key corresponding to those two primes, and test that the public key factorizes into Blum primes:

```
sage: from sage.crypto.util import random_blum_prime
sage: p = random_blum_prime(10**4, 10**5)
sage: q = random_blum_prime(10**4, 10**5)
sage: while q == p:
```

(continues on next page)

(continued from previous page)

```
....:     q = random_blum_prime(10**4, 10**5)
...
sage: n = bg.public_key(p, q)
sage: f = factor(n)
sage: is_blum_prime(f[0][0]); is_blum_prime(f[1][0])
True
True
sage: p * q == f[0][0] * f[1][0]
True
```

random_key(*lbound*, *ubound*, *ntries=100*)

Return a pair of random public and private keys.

INPUT:

- `lbound` – positive integer; the lower bound on how small each random Blum prime p and q can be. So we have `0 < lower_bound <= p, q <= upper_bound`. The lower bound must be distinct from the upper bound.
- `ubound` – positive integer; the upper bound on how large each random Blum prime p and q can be. So we have `0 < lower_bound <= p, q <= upper_bound`. The lower bound must be distinct from the upper bound.
- `ntries` – (default: `100`) the number of attempts to generate a random public/private key pair. If `ntries` is a positive integer, then perform that many attempts at generating a random public/private key pair.

OUTPUT:

- A random public key and its corresponding private key. Each randomly chosen p and q are guaranteed to be Blum primes. The public key is $n = pq$, and the private key is (p, q, a, b) where $\gcd(p, q) = ap + bq = 1$.

ALGORITHM:

The key generation algorithm is described in Algorithm 8.55, page 308 of [MvOV1996]. The algorithm works as follows:

1. Let p and q be distinct large random primes, each congruent to 3 modulo 4. That is, p and q are Blum primes.
2. Let $n = pq$ be the product of p and q.
3. Use the extended Euclidean algorithm to compute integers a and b such that $\gcd(p, q) = ap + bq = 1$.
4. The public key is n and the corresponding private key is (p, q, a, b).

Note: Beware that there might not be any primes between the lower and upper bounds. So make sure that these two bounds are "sufficiently" far apart from each other for there to be primes congruent to 3 modulo 4. In particular, there should be at least two distinct primes within these bounds, each prime being congruent to 3 modulo 4.

EXAMPLES:

Choosing a random pair of public and private keys. We then test to see if they satisfy the requirements of the Blum-Goldwasser scheme:

```
sage: from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
sage: from sage.crypto.util import is_blum_prime
sage: bg = BlumGoldwasser()
sage: pubkey, prikey = bg.random_key(10**4, 10**5)
sage: p, q, a, b = prikey
sage: is_blum_prime(p); is_blum_prime(q)
True
True
sage: p == q
False
sage: pubkey == p*q
True
sage: gcd(p, q) == a*p + b*q == 1
True
```

CHAPTER

EIGHT

STREAM CRYPTOSYSTEMS

class sage.crypto.stream.**LFSRCryptosystem**(*field=None*)

Bases: *sage.crypto.cryptosystem.SymmetricKeyCryptosystem*

Linear feedback shift register cryptosystem class

encoding(*M*)

class sage.crypto.stream.**ShrinkingGeneratorCryptosystem**(*field=None*)

Bases: *sage.crypto.cryptosystem.SymmetricKeyCryptosystem*

Shrinking generator cryptosystem class

encoding(*M*)

sage.crypto.stream.**blum_blum_shub**(*length*, *seed=None*, *p=None*, *q=None*, *lbound=None*, *ubound=None*, *ntries=100*)

The Blum-Blum-Shub (BBS) pseudorandom bit generator.

See the original paper by Blum, Blum and Shub [BBS1986]. The BBS algorithm is also discussed in section 5.5.2 of [MvOV1996].

INPUT:

- `length` – positive integer; the number of bits in the output pseudorandom bit sequence.
- `seed` – (default: `None`) if p and q are Blum primes, then `seed` is a quadratic residue in the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$ where $n = pq$. If `seed=None`, then the function would generate its own random quadratic residue in $(\mathbf{Z}/n\mathbf{Z})^*$. If you provide a value for `seed`, then it is your responsibility to ensure that the seed is a quadratic residue in the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$.
- `p` – (default: `None`) a large positive prime congruent to 3 modulo 4. Both `p` and `q` must be distinct. If `p=None`, then a value for `p` will be generated, where `0 < lower_bound <= p <= upper_bound`.
- `q` – (default: `None`) a large positive prime congruence to 3 modulo 4. Both `p` and `q` must be distinct. If `q=None`, then a value for `q` will be generated, where `0 < lower_bound <= q <= upper_bound`.
- `lbound` – (positive integer, default: `None`) the lower bound on how small each random primes p and q can be. So we have `0 < lbound <= p, q <= ubound`. The lower bound must be distinct from the upper bound.
- `ubound` – (positive integer, default: `None`) the upper bound on how large each random primes p and q can be. So we have `0 < lbound <= p, q <= ubound`. The lower bound must be distinct from the upper bound.
- `ntries` – (default: `100`) the number of attempts to generate a random Blum prime. If `ntries` is a positive integer, then perform that many attempts at generating a random Blum prime. This might or might not result in a Blum prime.

OUTPUT:

- A pseudorandom bit sequence whose length is specified by `length`.

Here is a common use case for this function. If you want this function to use pre-computed values for p and q, you should pass those pre-computed values to this function. In that case, you only need to specify values for `length`, `p` and `q`, and you do not need to worry about doing anything with the parameters `lbound` and `ubound`. The pre-computed values p and q must be Blum primes. It is your responsibility to check that both p and q are Blum primes.

Here is another common use case. If you want the function to generate its own values for p and q, you must specify the lower and upper bounds within which these two primes must lie. In that case, you must specify values for `length`, `lbound` and `ubound`, and you do not need to worry about values for the parameters `p` and `q`. The parameter `ntries` is only relevant when you want this function to generate `p` and `q`.

Note: Beware that there might not be any primes between the lower and upper bounds. So make sure that these two bounds are "sufficiently" far apart from each other for there to be primes congruent to 3 modulo 4. In particular, there should be at least two distinct primes within these bounds, each prime being congruent to 3 modulo 4. This function uses the function *random_blum_prime()* to generate random primes that are congruent to 3 modulo 4.

ALGORITHM:

The BBS algorithm as described below is adapted from the presentation in Algorithm 5.40, page 186 of [MvOV1996].

1. Let L be the desired number of bits in the output bit sequence. That is, L is the desired length of the bit string.
2. Let p and q be two large distinct primes, each congruent to 3 modulo 4.
3. Let $n = pq$ be the product of p and q.
4. Select a random seed value $s \in (\mathbf{Z}/n\mathbf{Z})^*$, where $(\mathbf{Z}/n\mathbf{Z})^*$ is the multiplicative group of $\mathbf{Z}/n\mathbf{Z}$.
5. Let $x_0 = s^2 \bmod n$.
6. For i from 1 to L, do
   (a) Let $x_i = x_{i-1}^2 \bmod n$.
   (b) Let z_i be the least significant bit of x_i.
7. The output pseudorandom bit sequence is $z_1, z_2, \ldots, z_L$.

EXAMPLES:

A BBS pseudorandom bit sequence with a specified seed:

```
sage: from sage.crypto.stream import blum_blum_shub
sage: blum_blum_shub(length=6, seed=3, p=11, q=19)
110000
```

You could specify the length of the bit string, with given values for `p` and `q`:

```
sage: blum_blum_shub(length=6, p=11, q=19)  # random
001011
```

Or you could specify the length of the bit string, with given values for the lower and upper bounds:

```
sage: blum_blum_shub(length=6, lbound=10**4, ubound=10**5)  # random
110111
```

Under some reasonable hypotheses, Blum-Blum-Shub [BBS1982] sketch a proof that the period of the BBS stream cipher is equal to $\lambda(\lambda(n))$, where $\lambda(n)$ is the Carmichael function of n. This is verified below in a few examples by using the function *lfsr_connection_polynomial()* (written by Tim Brock) which computes the connection polynomial of a linear feedback shift register sequence. The degree of that polynomial is the period.

```
sage: from sage.crypto.stream import blum_blum_shub
sage: from sage.crypto.util import carmichael_lambda
sage: carmichael_lambda(carmichael_lambda(7*11))
4
sage: s = [GF(2)(int(str(x))) for x in blum_blum_shub(60, p=7, q=11, seed=13)]
sage: lfsr_connection_polynomial(s)
x^3 + x^2 + x + 1
sage: carmichael_lambda(carmichael_lambda(11*23))
20
sage: s = [GF(2)(int(str(x))) for x in blum_blum_shub(60, p=11, q=23, seed=13)]
sage: lfsr_connection_polynomial(s)
x^19 + x^18 + x^17 + x^16 + x^15 + x^14 + x^13 + x^12 + x^11 + x^10 + x^9 + x^8 +␣
↪x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
```

CHAPTER

NINE

STREAM CIPHERS

class sage.crypto.stream_cipher.**LFSRCipher**(*parent*, *poly*, *IS*)

Bases: *sage.crypto.cipher.SymmetricKeyCipher*

Create a linear feedback shift register (LFSR) cipher.

INPUT:

- `parent` - parent
- `poly` - connection polynomial
- `IS` - initial state

EXAMPLES:

```
sage: FF = FiniteField(2)
sage: P.<x> = PolynomialRing(FF)
sage: E = LFSRCryptosystem(FF)
sage: E
LFSR cryptosystem over Finite Field of size 2
sage: IS = [ FF(a) for a in [0,1,1,1,0,1,1] ]
sage: g = x^7 + x + 1
sage: e = E((g,IS))
sage: B = BinaryStrings()
sage: m = B.encoding("THECATINTHEHAT")
sage: e(m)
00100011011110101110101010100011000000001101000101010111000010111100100100000111111100100100011100
sage: FF = FiniteField(2)
sage: P.<x> = PolynomialRing(FF)
sage: LFSR = LFSRCryptosystem(FF)
sage: e = LFSR((x^2+x+1,[FF(0),FF(1)]))
sage: B = e.domain()
sage: m = B.encoding("The cat in the hat.")
sage: e(m)
00111001110111101011111001001101110101011011101000011001100101101011001000000011100101101010111
sage: m == e(e(m))
True
```

connection_polynomial()

The connection polynomial defining the LFSR of the cipher.

EXAMPLES:

```
sage: k = GF(2)
sage: P.<x> = PolynomialRing( k )
sage: LFSR = LFSRCryptosystem( k )
```

(continues on next page)

(continued from previous page)

```
sage: e = LFSR((x^2+x+1,[k(0), k(1)]))
sage: e.connection_polynomial()
x^2 + x + 1
```

initial_state()

The initial state of the LFSR cipher.

EXAMPLES:

```
sage: k = GF(2)
sage: P.<x> = PolynomialRing( k )
sage: LFSR = LFSRCryptosystem( k )
sage: e = LFSR((x^2+x+1,[k(0), k(1)]))
sage: e.initial_state()
[0, 1]
```

class sage.crypto.stream_cipher.**ShrinkingGeneratorCipher**(*parent*, *e1*, *e2*)

Bases: *sage.crypto.cipher.SymmetricKeyCipher*

Create a shrinking generator cipher.

INPUT:

- `parent` - parent
- `poly` - connection polynomial
- `IS` - initial state

EXAMPLES:

```
sage: FF = FiniteField(2)
sage: P.<x> = PolynomialRing(FF)
sage: LFSR = LFSRCryptosystem(FF)
sage: IS_1 = [ FF(a) for a in [0,1,0,1,0,0,0] ]
sage: e1 = LFSR((x^7 + x + 1,IS_1))
sage: IS_2 = [ FF(a) for a in [0,0,1,0,0,0,1,0,1] ]
sage: e2 = LFSR((x^9 + x^3 + 1,IS_2))
sage: E = ShrinkingGeneratorCryptosystem()
sage: e = E((e1,e2))
sage: e
Shrinking generator cipher on Free binary string monoid
```

decimating_cipher()

The LFSR cipher generating the decimating key stream.

EXAMPLES:

```
sage: FF = FiniteField(2)
sage: P.<x> = PolynomialRing(FF)
sage: LFSR = LFSRCryptosystem(FF)
sage: IS_1 = [ FF(a) for a in [0,1,0,1,0,0,0] ]
sage: e1 = LFSR((x^7 + x + 1,IS_1))
sage: IS_2 = [ FF(a) for a in [0,0,1,0,0,0,1,0,1] ]
sage: e2 = LFSR((x^9 + x^3 + 1,IS_2))
sage: E = ShrinkingGeneratorCryptosystem()
sage: e = E((e1,e2))
sage: e.decimating_cipher()
LFSR cipher on Free binary string monoid
```

keystream_cipher()

The LFSR cipher generating the output key stream.

EXAMPLES:

```
sage: FF = FiniteField(2)
sage: P.<x> = PolynomialRing(FF)
sage: LFSR = LFSRCryptosystem(FF)
sage: IS_1 = [ FF(a) for a in [0,1,0,1,0,0,0] ]
sage: e1 = LFSR((x^7 + x + 1,IS_1))
sage: IS_2 = [ FF(a) for a in [0,0,1,0,0,0,1,0,1] ]
sage: e2 = LFSR((x^9 + x^3 + 1,IS_2))
sage: E = ShrinkingGeneratorCryptosystem()
sage: e = E((e1,e2))
sage: e.keystream_cipher()
LFSR cipher on Free binary string monoid
```

CHAPTER

TEN

LINEAR FEEDBACK SHIFT REGISTER (LFSR) SEQUENCE COMMANDS

Stream ciphers have been used for a long time as a source of pseudo-random number generators.

S. Golomb [Go1967] gives a list of three statistical properties that a sequence of numbers $\mathbf{a} = \{a_n\}_{n=1}^{\infty}$, $a_n \in \{0, 1\}$ should display to be considered "random". Define the autocorrelation of $\mathbf{a}$ to be

$$C(k) = C(k, \mathbf{a}) = \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} (-1)^{a_n + a_{n+k}}.$$

In the case where $\mathbf{a}$ is periodic with period P, then this reduces to

$$C(k) = \frac{1}{P} \sum_{n=1}^{P} (-1)^{a_n + a_{n+k}}.$$

Assume $\mathbf{a}$ is periodic with period P.

- balance: $|\sum_{n=1}^{P} (-1)^{a_n}| \leq 1$.
- low autocorrelation:

$$C(k) = \begin{cases} 1, & k = 0, \\ \epsilon, & k \neq 0. \end{cases}$$

  (For sequences satisfying these first two properties, it is known that $\epsilon = -1/P$ must hold.)

- proportional runs property: In each period, half the runs have length 1, one-fourth have length 2, etc. Moreover, there are as many runs of 1's as there are of 0's.

A general feedback shift register is a map $f : \mathbf{F}_q^d \to \mathbf{F}_q^d$ of the form

$$f(x_0, ..., x_{n-1}) = (x_1, x_2, ..., x_n),$$
$$x_n = C(x_0, ..., x_{n-1}),$$

where $C : \mathbf{F}_q^d \to \mathbf{F}_q$ is a given function. When C is of the form

$$C(x_0, ..., x_{n-1}) = a_0 x_0 + ... + a_{n-1} x_{n-1},$$

for some given constants $a_i \in \mathbf{F}_q$, the map is called a linear feedback shift register (LFSR).

Example of an LFSR: Let

$$f(x) = a_0 + a_1 x + ... + a_n x^n + ...,$$

$$g(x) = b_0 + b_1 x + ... + b_n x^n + ...,$$

be given polynomials in $\mathbf{F}_2[x]$ and let

$$h(x) = \frac{f(x)}{g(x)} = c_0 + c_1 x + ... + c_n x^n + ...$$

We can compute a recursion formula which allows us to rapidly compute the coefficients of $h(x)$ (take $f(x) = 1$):

$$c_n = \sum_{i=1}^{n} \frac{-b_i}{b_0} c_{n-i}.$$

The coefficients of $h(x)$ can, under certain conditions on $f(x)$ and $g(x)$, be considered "random" from certain statistical points of view.

Example: For instance, if

$$f(x) = 1, \quad g(x) = x^4 + x + 1,$$

then

$$h(x) = 1 + x + x^2 + x^3 + x^5 + x^7 + x^8 + ...$$

The coefficients of h are

$$1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1,$$
$$1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, ...$$

The sequence of $0, 1$'s is periodic with period $P = 2^4 - 1 = 15$ and satisfies Golomb's three randomness conditions. However, this sequence of period 15 can be "cracked" (i.e., a procedure to reproduce $g(x)$) by knowing only 8 terms! This is the function of the Berlekamp-Massey algorithm [Mas1969], implemented in `berlekamp_massey.py`.

AUTHORS:

- David Joyner (2005-11-24): initial creation.
- Timothy Brock (2005-11): added `lfsr_sequence` with code modified from Python Cookbook, http://aspn.activestate.com/ASPN/Python/Cookbook/
- Timothy Brock (2006-04-17): added `lfsr_autocorrelation` and `lfsr_connection_polynomial`.

`sage.crypto.lfsr.`**`lfsr_autocorrelation`**(*L*, *p*, *k*)

INPUT:

- `L` – a periodic sequence of elements of ZZ or GF(2); must have length p
- `p` – the period of L
- `k` – an integer between 0 and p

OUTPUT: autocorrelation sequence of L

EXAMPLES:

```
sage: F = GF(2)
sage: o = F(0)
sage: l = F(1)
sage: key = [l,o,o,l]; fill = [l,l,o,l]; n = 20
sage: s = lfsr_sequence(key,fill,n)
sage: lfsr_autocorrelation(s,15,7)
4/15
sage: lfsr_autocorrelation(s,int(15),7)
4/15
```

sage.crypto.lfsr.**lfsr_connection_polynomial**(*s*)

INPUT:

- s – a sequence of elements of a finite field of even length

OUTPUT:

- C(x) – the connection polynomial of the minimal LFSR.

This implements the algorithm in section 3 of J. L. Massey's article [Mas1969].

EXAMPLES:

```
sage: F = GF(2)
sage: F
Finite Field of size 2
sage: o = F(0); l = F(1)
sage: key = [l,o,o,l]; fill = [l,l,o,l]; n = 20
sage: s = lfsr_sequence(key,fill,n); s
[1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0]
sage: lfsr_connection_polynomial(s)
x^4 + x + 1
sage: berlekamp_massey(s)
x^4 + x^3 + 1
```

Notice that berlekamp_massey returns the reverse of the connection polynomial (and is potentially must faster than this implementation).

sage.crypto.lfsr.**lfsr_sequence**(*key*, *fill*, *n*)

This function creates an LFSR sequence.

INPUT:

- key – a list of finite field elements, [c_0,c_1,...,c_k]
- fill – the list of the initial terms of the LFSR sequence, [x_0,x_1,...,x_k]
- n – number of terms of the sequence that the function returns

OUTPUT: The LFSR sequence defined by $x_{n+1} = c_k x_n + ... + c_0 x_{n-k}$ for $n \leq k$.

EXAMPLES:

```
sage: F = GF(2); l = F(1); o = F(0)
sage: F = GF(2); S = LaurentSeriesRing(F,'x'); x = S.gen()
sage: fill = [l,l,o,l]; key = [1,o,o,l]; n = 20
sage: L = lfsr_sequence(key,fill,20); L
[1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0]
sage: g = berlekamp_massey(L); g
x^4 + x^3 + 1
sage: (1)/(g.reverse()+O(x^20))
1 + x + x^2 + x^3 + x^5 + x^7 + x^8 + x^11 + x^15 + x^16 + x^17 + x^18 + O(x^20)
sage: (1+x^2)/(g.reverse()+O(x^20))
1 + x + x^4 + x^8 + x^9 + x^10 + x^11 + x^13 + x^15 + x^16 + x^19 + O(x^20)
sage: (1+x^2+x^3)/(g.reverse()+O(x^20))
1 + x + x^3 + x^5 + x^6 + x^9 + x^13 + x^14 + x^15 + x^16 + x^18 + O(x^20)
sage: fill = [l,l,o,l]; key = [l,o,o,o]; n = 20
sage: L = lfsr_sequence(key,fill,20); L
[1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1]
sage: g = berlekamp_massey(L); g
x^4 + 1
sage: (1+x)/(g.reverse()+O(x^20))
```

(continues on next page)

(continued from previous page)

```
1 + x + x^4 + x^5 + x^8 + x^9 + x^12 + x^13 + x^16 + x^17 + O(x^20)
sage: (1+x+x^3)/(g.reverse()+O(x^20))
1 + x + x^3 + x^4 + x^5 + x^7 + x^8 + x^9 + x^11 + x^12 + x^13 + x^15 + x^16 + x^
↪17 + x^19 + O(x^20)
```

CHAPTER

ELEVEN

UTILITY FUNCTIONS FOR CRYPTOGRAPHY

Miscellaneous utility functions for cryptographic purposes.

AUTHORS:

- Minh Van Nguyen (2009-12): initial version with the following functions: `ascii_integer`, `ascii_to_bin`, `bin_to_ascii`, `has_blum_prime`, `is_blum_prime`, `least_significant_bits`, `random_blum_prime`.

`sage.crypto.util.`**`ascii_integer`**`(B)`

Return the ASCII integer corresponding to the binary string `B`.

INPUT:

- `B` – a non-empty binary string or a non-empty list of bits. The number of bits in `B` must be 8.

OUTPUT:

- The ASCII integer corresponding to the 8-bit block `B`.

EXAMPLES:

The ASCII integers of some binary strings:

```
sage: from sage.crypto.util import ascii_integer
sage: bin = BinaryStrings()
sage: B = bin.encoding("A"); B
01000001
sage: ascii_integer(B)
65
sage: B = bin.encoding("C"); list(B)
[0, 1, 0, 0, 0, 0, 1, 1]
sage: ascii_integer(list(B))
67
sage: ascii_integer("01000100")
68
sage: ascii_integer([0, 1, 0, 0, 0, 1, 0, 1])
69
```

`sage.crypto.util.`**`ascii_to_bin`**`(A)`

Return the binary representation of the ASCII string `A`.

INPUT:

- `A` – a string or list of ASCII characters.

OUTPUT:

- The binary representation of `A`.

ALGORITHM:

Let $A = a_0 a_1 \cdots a_{n-1}$ be an ASCII string, where each a_i is an ASCII character. Let c_i be the ASCII integer corresponding to a_i and let b_i be the binary representation of c_i. The binary representation B of A is $B = b_0 b_1 \cdots b_{n-1}$.

EXAMPLES:

The binary representation of some ASCII strings:

```
sage: from sage.crypto.util import ascii_to_bin
sage: ascii_to_bin("A")
01000001
sage: ascii_to_bin("Abc123")
010000010110001001100011001100010011001000110011
```

The empty string is different from the string with one space character. For the empty string and the empty list, this function returns the same result:

```
sage: from sage.crypto.util import ascii_to_bin
sage: ascii_to_bin("")

sage: ascii_to_bin(" ")
00100000
sage: ascii_to_bin([])
```

This function also accepts a list of ASCII characters. You can also pass in a list of strings:

```
sage: from sage.crypto.util import ascii_to_bin
sage: ascii_to_bin(["A", "b", "c", "1", "2", "3"])
010000010110001001100011001100010011001000110011
sage: ascii_to_bin(["A", "bc", "1", "23"])
010000010110001001100011001100010011001000110011
```

sage.crypto.util.**bin_to_ascii**(*B*)

Return the ASCII representation of the binary string `B`.

INPUT:

- `B` – a non-empty binary string or a non-empty list of bits. The number of bits in `B` must be a multiple of 8.

OUTPUT:

- The ASCII string corresponding to `B`.

ALGORITHM:

Consider a block of bits $B = b_0 b_1 \cdots b_{n-1}$ where each sub-block b_i is a binary string of length 8. Then the total number of bits is a multiple of 8 and is given by $8n$. Let c_i be the integer representation of b_i. We can consider c_i as the integer representation of an ASCII character. Then the ASCII representation A of B is $A = a_0 a_1 \cdots a_{n-1}$.

EXAMPLES:

Convert some ASCII strings to their binary representations and recover the ASCII strings from the binary representations:

```
sage: from sage.crypto.util import ascii_to_bin
sage: from sage.crypto.util import bin_to_ascii
sage: A = "Abc"
sage: B = ascii_to_bin(A); B
010000010110001001100011
```

(continues on next page)

(continued from previous page)

```
sage: bin_to_ascii(B)
'Abc'
sage: bin_to_ascii(B) == A
True
```

```
sage: A = "123 \" #"
sage: B = ascii_to_bin(A); B
00110001001100100011001100100000001000100010000000100011
sage: bin_to_ascii(B)
'123 " #'
sage: bin_to_ascii(B) == A
True
```

This function also accepts strings and lists of bits:

```
sage: from sage.crypto.util import bin_to_ascii
sage: bin_to_ascii("010000010110001001100011")
'Abc'
sage: bin_to_ascii([0, 1, 0, 0, 0, 0, 0, 1])
'A'
```

sage.crypto.util.**carmichael_lambda**(*n*)

Return the Carmichael function of a positive integer `n`.

The Carmichael function of n, denoted $\lambda(n)$, is the smallest positive integer k such that $a^k \equiv 1 \pmod{n}$ for all $a \in \mathbf{Z}/n\mathbf{Z}$ satisfying $\gcd(a, n) = 1$. Thus, $\lambda(n) = k$ is the exponent of the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$.

INPUT:

- `n` – a positive integer.

OUTPUT:

- The Carmichael function of `n`.

ALGORITHM:

If $n = 2, 4$ then $\lambda(n) = \varphi(n)$. Let $p \geq 3$ be an odd prime and let k be a positive integer. Then $\lambda(p^k) = p^{k-1}(p-1) = \varphi(p^k)$. If $k \geq 3$, then $\lambda(2^k) = 2^{k-2}$. Now consider the case where $n > 3$ is composite and let $n = p_1^{k_1} p_2^{k_2} \cdots p_t^{k_t}$ be the prime factorization of n. Then

$$\lambda(n) = \lambda(p_1^{k_1} p_2^{k_2} \cdots p_t^{k_t}) = \text{lcm}(\lambda(p_1^{k_1}), \lambda(p_2^{k_2}), \ldots, \lambda(p_t^{k_t}))$$

EXAMPLES:

The Carmichael function of all positive integers up to and including 10:

```
sage: from sage.crypto.util import carmichael_lambda
sage: list(map(carmichael_lambda, [1..10]))
[1, 1, 2, 2, 4, 2, 6, 2, 6, 4]
```

The Carmichael function of the first ten primes:

```
sage: list(map(carmichael_lambda, primes_first_n(10)))
[1, 2, 4, 6, 10, 12, 16, 18, 22, 28]
```

Cases where the Carmichael function is equivalent to the Euler phi function:

```
sage: carmichael_lambda(2) == euler_phi(2)
True
sage: carmichael_lambda(4) == euler_phi(4)
True
sage: p = random_prime(1000, lbound=3, proof=True)
sage: k = randint(1, 1000)
sage: carmichael_lambda(p^k) == euler_phi(p^k)
True
```

A case where $\lambda(n) \neq \varphi(n)$:

```
sage: k = randint(1, 1000)
sage: carmichael_lambda(2^k) == 2^(k - 2)
True
sage: carmichael_lambda(2^k) == 2^(k - 2) == euler_phi(2^k)
False
```

Verifying the current implementation of the Carmichael function using another implementation. The other implementation that we use for verification is an exhaustive search for the exponent of the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$.

```
sage: from sage.crypto.util import carmichael_lambda
sage: n = randint(1, 500)
sage: c = carmichael_lambda(n)
sage: def coprime(n):
....:     return [i for i in range(n) if gcd(i, n) == 1]
sage: def znpower(n, k):
....:     L = coprime(n)
....:     return list(map(power_mod, L, [k]*len(L), [n]*len(L)))
sage: def my_carmichael(n):
....:     for k in range(1, n):
....:         L = znpower(n, k)
....:         ones = [1] * len(L)
....:         T = [L[i] == ones[i] for i in range(len(L))]
....:         if all(T):
....:             return k
sage: c == my_carmichael(n)
True
```

Carmichael's theorem states that $a^{\lambda(n)} \equiv 1 \pmod{n}$ for all elements a of the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$. Here, we verify Carmichael's theorem.

```
sage: from sage.crypto.util import carmichael_lambda
sage: n = randint(1, 1000)
sage: c = carmichael_lambda(n)
sage: ZnZ = IntegerModRing(n)
sage: M = ZnZ.list_of_elements_of_multiplicative_group()
sage: ones = [1] * len(M)
sage: P = [power_mod(a, c, n) for a in M]
sage: P == ones
True
```

REFERENCES:

- Wikipedia article Carmichael_function

sage.crypto.util.**has_blum_prime**(*lbound*, *ubound*)

Determine whether or not there is a Blum prime within the specified closed interval.

INPUT:

- `lbound` – positive integer; the lower bound on how small a Blum prime can be. The lower bound must be distinct from the upper bound.
- `ubound` – positive integer; the upper bound on how large a Blum prime can be. The lower bound must be distinct from the upper bound.

OUTPUT:

- `True` if there is a Blum prime `p` such that `lbound <= p <= ubound`. `False` otherwise.

ALGORITHM:

Let L and U be distinct positive integers. Let P be the set of all odd primes p such that $L \leq p \leq U$. Our main focus is on Blum primes, i.e. odd primes that are congruent to 3 modulo 4, so we assume that the lower bound $L > 2$. The closed interval $[L, U]$ has a Blum prime if and only if the set P has a Blum prime.

EXAMPLES:

Testing for the presence of Blum primes within some closed intervals. The interval $[4, 100]$ has a Blum prime, the smallest such prime being 7. The interval $[24, 28]$ has no primes, hence no Blum primes.

```
sage: from sage.crypto.util import has_blum_prime
sage: from sage.crypto.util import is_blum_prime
sage: has_blum_prime(4, 100)
True
sage: for n in range(4, 100):
....:     if is_blum_prime(n):
....:         print(n)
....:         break
7
sage: has_blum_prime(24, 28)
False
```

sage.crypto.util.**is_blum_prime**(*n*)

Determine whether or not `n` is a Blum prime.

INPUT:

- `n` a positive prime.

OUTPUT:

- `True` if `n` is a Blum prime; `False` otherwise.

Let n be a positive prime. Then n is a Blum prime if n is congruent to 3 modulo 4, i.e. $n \equiv 3 \pmod 4$.

EXAMPLES:

Testing some integers to see if they are Blum primes:

```
sage: from sage.crypto.util import is_blum_prime
sage: from sage.crypto.util import random_blum_prime
sage: is_blum_prime(101)
False
sage: is_blum_prime(7)
True
sage: p = random_blum_prime(10**3, 10**5)
sage: is_blum_prime(p)
True
```

sage.crypto.util.**least_significant_bits**(*n*, *k*)

Return the k least significant bits of n.

INPUT:

- n – an integer.
- k – a positive integer.

OUTPUT:

- The k least significant bits of the integer n. If k=1, then return the parity bit of the integer n. Let b be the binary representation of n, where m is the length of the binary string b. If $k \geq m$, then return the binary representation of n.

EXAMPLES:

Obtain the parity bits of some integers:

```
sage: from sage.crypto.util import least_significant_bits
sage: least_significant_bits(0, 1)
[0]
sage: least_significant_bits(2, 1)
[0]
sage: least_significant_bits(3, 1)
[1]
sage: least_significant_bits(-2, 1)
[0]
sage: least_significant_bits(-3, 1)
[1]
```

Obtain the 4 least significant bits of some integers:

```
sage: least_significant_bits(101, 4)
[0, 1, 0, 1]
sage: least_significant_bits(-101, 4)
[0, 1, 0, 1]
sage: least_significant_bits(124, 4)
[1, 1, 0, 0]
sage: least_significant_bits(-124, 4)
[1, 1, 0, 0]
```

The binary representation of 123:

```
sage: n = 123; b = n.binary(); b
'1111011'
sage: least_significant_bits(n, len(b))
[1, 1, 1, 1, 0, 1, 1]
```

sage.crypto.util.**random_blum_prime**(*lbound*, *ubound*, *ntries=100*)

A random Blum prime within the specified bounds.

Let p be a positive prime. Then p is a Blum prime if p is congruent to 3 modulo 4, i.e. $p \equiv 3 \pmod 4$.

INPUT:

- lbound – positive integer; the lower bound on how small a random Blum prime p can be. So we have 0 < lbound <= p <= ubound. The lower bound must be distinct from the upper bound.
- ubound – positive integer; the upper bound on how large a random Blum prime p can be. So we have 0 < lbound <= p <= ubound. The lower bound must be distinct from the upper bound.

- `ntries` – (default: `100`) the number of attempts to generate a random Blum prime. If `ntries` is a positive integer, then perform that many attempts at generating a random Blum prime. This might or might not result in a Blum prime.

OUTPUT:

- A random Blum prime within the specified lower and upper bounds.

Note: Beware that there might not be any primes between the lower and upper bounds. So make sure that these two bounds are "sufficiently" far apart from each other for there to be primes congruent to 3 modulo 4. In particular, there should be at least two distinct Blum primes within the specified bounds.

EXAMPLES:

Choose a random prime and check that it is a Blum prime:

```
sage: from sage.crypto.util import random_blum_prime
sage: p = random_blum_prime(10**4, 10**5)
sage: is_prime(p)
True
sage: mod(p, 4) == 3
True
```

CHAPTER

TWELVE

BOOLEAN FUNCTIONS

Those functions are used for example in LFSR based ciphers like the filter generator or the combination generator.

This module allows to study properties linked to spectral analysis, and also algebraic immunity.

EXAMPLES:

```
sage: R.<x>=GF(2^8,'a')[]
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction( x^254 ) # the Boolean function Tr(x^254)
sage: B
Boolean function with 8 variables
sage: B.nonlinearity()
112
sage: B.algebraic_immunity()
4
```

AUTHOR:

- Rusydi H. Makarim (2016-10-13): add functions related to linear structures
- Rusydi H. Makarim (2016-07-09): add is_plateaued()
- Yann Laigle-Chapuy (2010-02-26): add basic arithmetic
- Yann Laigle-Chapuy (2009-08-28): first implementation

class sage.crypto.boolean_function.**BooleanFunction**

Bases: sage.structure.sage_object.SageObject

This module implements Boolean functions represented as a truth table.

We can construct a Boolean Function from either:

- an integer - the result is the zero function with `x` variables;
- a list - it is expected to be the truth table of the result. Therefore it must be of length a power of 2, and its elements are interpreted as Booleans;
- a string - representing the truth table in hexadecimal;
- a Boolean polynomial - the result is the corresponding Boolean function;
- a polynomial P over an extension of GF(2) - the result is the Boolean function with truth table `( Tr(P(x)) for x in GF(2^k) )`

EXAMPLES:

from the number of variables:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: BooleanFunction(5)
Boolean function with 5 variables
```

from a truth table:

```
sage: BooleanFunction([1,0,0,1])
Boolean function with 2 variables
```

note that elements can be of different types:

```
sage: B = BooleanFunction([False, sqrt(2)])
sage: B
Boolean function with 1 variable
sage: [b for b in B]
[False, True]
```

from a string:

```
sage: BooleanFunction("111e")
Boolean function with 4 variables
```

from a sage.rings.polynomial.pbori.BooleanPolynomial:

```
sage: R.<x,y,z> = BooleanPolynomialRing(3)
sage: P = x*y
sage: BooleanFunction( P )
Boolean function with 3 variables
```

from a polynomial over a binary field:

```
sage: R.<x> = GF(2^8,'a')[]
sage: B = BooleanFunction( x^7 )
sage: B
Boolean function with 8 variables
```

two failure cases:

```
sage: BooleanFunction(sqrt(2))
Traceback (most recent call last):
...
TypeError: unable to init the Boolean function

sage: BooleanFunction([1, 0, 1])
Traceback (most recent call last):
...
ValueError: the length of the truth table must be a power of 2
```

absolut_indicator()

Return the absolut indicator of the function. Ths is the maximal absolut value of the autocorrelation.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction("7969817CC5893BA6AC326E47619F5AD0")
sage: B.absolut_indicator()
32
```

absolute_autocorrelation()
Return the absolute autocorrelation of the function.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction("7969817CC5893BA6AC326E47619F5AD0")
sage: sorted(B.absolute_autocorrelation().items())
[(0, 33), (8, 58), (16, 28), (24, 6), (32, 2), (128, 1)]
```

absolute_walsh_spectrum()
Return the absolute Walsh spectrum fo the function.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction("7969817CC5893BA6AC326E47619F5AD0")
sage: sorted(B.absolute_walsh_spectrum().items())
[(0, 64), (16, 64)]

sage: B = BooleanFunction("0113077C165E76A8")
sage: B.absolute_walsh_spectrum()
{8: 64}
```

algebraic_degree()
Return the algebraic degree of this Boolean function.

The algebraic degree of a Boolean function is defined as the degree of its algebraic normal form. Note that the degree of the constant zero function is defined to be equal to -1.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B.<x0, x1, x2, x3> = BooleanPolynomialRing()
sage: f = BooleanFunction(x1*x2 + x1*x2*x3 + x1)
sage: f.algebraic_degree()
3
sage: g = BooleanFunction([0, 0])
sage: g.algebraic_degree()
-1
```

algebraic_immunity(*annihilator=False*)
Returns the algebraic immunity of the Boolean function. This is the smallest integer i such that there exists a non trivial annihilator for $self$ or $self$.

INPUT:

- annihilator – a Boolean (default: False), if True, returns also an annihilator of minimal degree.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: R.<x0,x1,x2,x3,x4,x5> = BooleanPolynomialRing(6)
sage: B = BooleanFunction(x0*x1 + x1*x2 + x2*x3 + x3*x4 + x4*x5)
sage: B.algebraic_immunity(annihilator=True)
(2, x0*x1 + x1*x2 + x2*x3 + x3*x4 + x4*x5 + 1)
sage: B[0] +=1
sage: B.algebraic_immunity()
2
```

(continues on next page)

(continued from previous page)

```
sage: R.<x> = GF(2^8,'a')[]
sage: B = BooleanFunction(x^31)
sage: B.algebraic_immunity()
4
```

algebraic_normal_form()

Return the sage.rings.polynomial.pbori.BooleanPolynomial corresponding to the algebraic normal form.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction([0,1,1,0,1,0,1,1])
sage: P = B.algebraic_normal_form()
sage: P
x0*x1*x2 + x0 + x1*x2 + x1 + x2
sage: [ P(*ZZ(i).digits(base=2,padto=3)) for i in range(8) ]
[0, 1, 1, 0, 1, 0, 1, 1]
```

annihilator(*d*, *dim=False*)

Return (if it exists) an annihilator of the boolean function of degree at most d, that is a Boolean polynomial g such that

$$f(x)g(x) = 0 \forall x.$$

INPUT:

- d – an integer;
- dim – a Boolean (default: False), if True, return also the dimension of the annihilator vector space.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: f = BooleanFunction("7969817CC5893BA6AC326E47619F5AD0")
sage: f.annihilator(1) is None
True
sage: g = BooleanFunction( f.annihilator(3) )
sage: set([ fi*g(i) for i,fi in enumerate(f) ])
{0}
```

autocorrelation()

Return the autocorrelation of the function, defined by

$$\Delta_f(j) = \sum_{i \in \{0,1\}^n} (-1)^{f(i) \oplus f(i \oplus j)}.$$

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction("03")
sage: B.autocorrelation()
(8, 8, 0, 0, 0, 0, 0, 0)
```

correlation_immunity()

Return the maximum value m such that the function is correlation immune of order m.

A Boolean function is said to be correlation immune of order m , if the output of the function is statistically independent of the combination of any m of its inputs.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction("7969817CC5893BA6AC326E47619F5AD0")
sage: B.correlation_immunity()
2
```

has_linear_structure()

Return `True` if this function has a linear structure.

An n-variable Boolean function f has a linear structure if there exists a nonzero $a \in \mathbf{F}_2^n$ such that $f(x \oplus a) \oplus f(x)$ is a constant function.

See also:

is_linear_structure(), *linear_structures()*.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: f = BooleanFunction([0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0])
sage: f.has_linear_structure()
True
sage: f.autocorrelation()
(16, -16, 0, 0, 0, 0, 0, 0, -16, 16, 0, 0, 0, 0, 0, 0)
sage: g = BooleanFunction([0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1])
sage: g.has_linear_structure()
False
sage: g.autocorrelation()
(16, 4, 4, 4, 4, -4, -4, -4, -4, 4, -4, -4, -4, 4, -4, -4)
```

is_balanced()

Return True if the function takes the value True half of the time.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction(1)
sage: B.is_balanced()
False
sage: B[0] = True
sage: B.is_balanced()
True
```

is_bent()

Return True if the function is bent.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction("0113077C165E76A8")
sage: B.is_bent()
True
```

is_linear_structure(*val*)

Return `True` if `val` is a linear structure of this Boolean function.

INPUT:

- `val` – either an integer or a tuple/list of $\mathbf{F}_2$ elements of length equal to the number of variables

See also:

has_linear_structure(), *linear_structures()*.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: f = BooleanFunction([0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0])
sage: f.is_linear_structure(1)
True
sage: l = [1, 0, 0, 1]
sage: f.is_linear_structure(l)
True
sage: v = vector(GF(2), l)
sage: f.is_linear_structure(v)
True
sage: f.is_linear_structure(7)
False
sage: f.is_linear_structure(20) #parameter is out of range
Traceback (most recent call last):
...
IndexError: index out of range
sage: v = vector(GF(3), [1, 0, 1, 1])
sage: f.is_linear_structure(v)
Traceback (most recent call last):
...
TypeError: base ring of input vector must be GF(2)
sage: v = vector(GF(2), [1, 0, 1, 1, 1])
sage: f.is_linear_structure(v)
Traceback (most recent call last):
...
TypeError: input vector must be an element of a vector space with dimension 4
sage: f.is_linear_structure('X') #failure case
Traceback (most recent call last):
...
TypeError: cannot compute is_linear_structure() using parameter X
```

is_plateaued()

Return `True` if this function is plateaued, i.e. its Walsh transform takes at most three values 0 and $\pm\lambda$, where λ is some positive integer.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: R.<x0, x1, x2, x3> = BooleanPolynomialRing()
sage: f = BooleanFunction(x0*x1 + x2 + x3)
sage: f.walsh_hadamard_transform()
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 8, 8, -8)
sage: f.is_plateaued()
True
```

is_symmetric()

Return True if the function is symmetric, i.e. invariant under permutation of its input bits. Another way to see it is that the output depends only on the Hamming weight of the input.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction(5)
```

(continues on next page)

(continued from previous page)

```
sage: B[3] = 1
sage: B.is_symmetric()
False
sage: V_B = [0, 1, 1, 0, 1, 0]
sage: for i in srange(32): B[i] = V_B[i.popcount()]
sage: B.is_symmetric()
True
```

linear_structures()

Return all linear structures of this Boolean function as a vector subspace of $\mathbf{F}_2^n$.

See also:

is_linear_structure(), *has_linear_structure()*.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: f = BooleanFunction([0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0])
sage: LS = f.linear_structures()
sage: LS.dimension()
2
sage: LS.basis_matrix()
[1 0 0 0]
[0 0 0 1]
sage: LS.list()
[(0, 0, 0, 0), (1, 0, 0, 0), (0, 0, 0, 1), (1, 0, 0, 1)]
```

nonlinearity()

Return the nonlinearity of the function. This is the distance to the linear functions, or the number of output ones need to change to obtain a linear function.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction(5)
sage: B[1] = B[3] = 1
sage: B.nonlinearity()
2
sage: B = BooleanFunction("0113077C165E76A8")
sage: B.nonlinearity()
28
```

nvariables()

The number of variables of this function.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: BooleanFunction(4).nvariables()
4
```

resiliency_order()

Return the maximum value m such that the function is resilient of order m.

A Boolean function is said to be resilient of order m if it is balanced and correlation immune of order m.

If the function is not balanced, we return -1.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction(
↪"077CE5A2F8831A5DF8831A5D077CE5A2699669966969969666999965AA5A55A")
sage: B.resiliency_order()
3
```

sum_of_square_indicator()
Return the sum of square indicator of the function.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction("7969817CC5893BA6AC326E47619F5AD0")
sage: B.sum_of_square_indicator()
32768
```

truth_table(*format='bin'*)
The truth table of the Boolean function.

INPUT: a string representing the desired format, can be either

- 'bin' (default) : we return a tuple of Boolean values
- 'int' : we return a tuple of 0 or 1 values
- 'hex' : we return a string representing the truth_table in hexadecimal

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: R.<x,y,z> = BooleanPolynomialRing(3)
sage: B = BooleanFunction( x*y*z + z + y + 1 )
sage: B.truth_table()
(True, True, False, False, False, False, True, False)
sage: B.truth_table(format='int')
(1, 1, 0, 0, 0, 0, 1, 0)
sage: B.truth_table(format='hex')
'43'

sage: BooleanFunction('00ab').truth_table(format='hex')
'00ab'

sage: H = '0abbacadabbacad0'
sage: len(H)
16
sage: T = BooleanFunction(H).truth_table(format='hex')
sage: T == H
True
sage: H = H * 4
sage: T = BooleanFunction(H).truth_table(format='hex')
sage: T == H
True
sage: H = H * 4
sage: T = BooleanFunction(H).truth_table(format='hex')
sage: T == H
True
sage: len(T)
256
sage: B.truth_table(format='oct')
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: unknown output format
```

walsh_hadamard_transform()

Compute the Walsh Hadamard transform W of the function f.

$$W(j) = \sum_{i \in \{0,1\}^n} (-1)^{f(i) \oplus i \cdot j}$$

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: R.<x> = GF(2^3,'a')[]
sage: B = BooleanFunction( x^3 )
sage: B.walsh_hadamard_transform()
(0, -4, 0, 4, 0, 4, 0, 4)
```

class sage.crypto.boolean_function.**BooleanFunctionIterator**

Bases: object

Iterator through the values of a Boolean function.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction(3)
sage: type(B.__iter__())
<type 'sage.crypto.boolean_function.BooleanFunctionIterator'>
```

next()

x.next() -> the next value, or raise StopIteration

sage.crypto.boolean_function.**random_boolean_function**(*n*)

Returns a random Boolean function with n variables.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import random_boolean_function
sage: B = random_boolean_function(9)
sage: B.nvariables()
9
sage: B.nonlinearity()
217                     # 32-bit
222                     # 64-bit
```

sage.crypto.boolean_function.**unpickle_BooleanFunction**(*bool_list*)

Specific function to unpickle Boolean functions.

EXAMPLES:

```
sage: from sage.crypto.boolean_function import BooleanFunction
sage: B = BooleanFunction([0,1,1,0])
sage: loads(dumps(B)) == B # indirect doctest
True
```

CHAPTER

THIRTEEN

S-BOXES AND THEIR ALGEBRAIC REPRESENTATIONS

class sage.crypto.sbox.**SBox**(**args*, ***kwargs*)

Bases: sage.structure.sage_object.SageObject

A substitution box or S-box is one of the basic components of symmetric key cryptography. In general, an S-box takes `m` input bits and transforms them into `n` output bits. This is called an `mxn` S-box and is often implemented as a lookup table. These S-boxes are carefully chosen to resist linear and differential cryptanalysis [He2002].

This module implements an S-box class which allows an algebraic treatment and determine various cryptographic properties.

EXAMPLES:

We consider the S-box of the block cipher PRESENT [BKLPPRSV2007]:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2); S
(12, 5, 6, 11, 9, 0, 10, 13, 3, 14, 15, 8, 4, 7, 1, 2)
sage: S(1)
5
```

Note that by default bits are interpreted in big endian order. This is not consistent with the rest of Sage, which has a strong bias towards little endian, but is consistent with most cryptographic literature:

```
sage: S([0,0,0,1])
[0, 1, 0, 1]

sage: S = SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2, big_endian=False)
sage: S(1)
5
sage: S([0,0,0,1])
[1, 1, 0, 0]
```

Now we construct an `SBox` object for the 4-bit small scale AES S-Box (cf. *sage.crypto.mq.sr*):

```
sage: sr = mq.SR(1,1,1,4, allow_zero_inversions=True)
sage: S = SBox([sr.sub_byte(e) for e in list(sr.k)])
sage: S
(6, 5, 2, 9, 4, 7, 3, 12, 14, 15, 10, 0, 8, 1, 13, 11)
```

AUTHORS:

- Rusydi H. Makarim (2016-03-31) : added more functions to determine related cryptographic properties
- Yann Laigle-Chapuy (2009-07-01): improve linear and difference matrix computation
- Martin R. Albrecht (2008-03-12): initial implementation

REFERENCES:

- [He2002]
- [BKLPPRSV2007]
- [CDL2015]

autocorrelation_matrix()

Return autocorrelation matrix correspond to this S-Box.

for an $m \times n$ S-Box S, its autocorrelation matrix entry at row $a \in \mathbf{F}_2^m$ and column $b \in \mathbf{F}_2^n$ (considering their integer representation) is defined as:

$$\sum_{x \in \mathbf{F}_2^m} (-1)^{b \cdot S(x) \oplus b \cdot S(x \oplus a)}$$

Equivalently, the columns b of autocorrelation matrix correspond to the autocorrelation spectrum of component function $b \cdot S(x)$.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.autocorrelation_matrix()
[ 8  8  8  8  8  8  8  8]
[ 8  0  0  0  0  0  0 -8]
[ 8  0 -8  0  0  0  0  0]
[ 8  0  0  0  0 -8  0  0]
[ 8 -8  0  0  0  0  0  0]
[ 8  0  0  0  0  0 -8  0]
[ 8  0  0 -8  0  0  0  0]
[ 8  0  0  0 -8  0  0  0]
```

boomerang_connectivity_matrix()

Return the boomerang connectivity matrix for this S-Box.

Boomerang connectivity matrix of an invertible $m \times m$ S-Box S is an $2^m \times 2^m$ matrix with entry at row $\Delta_i \in \mathbb{F}_2^m$ and column $\Delta_o \in \mathbb{F}_2^m$ equal to

$$|\{x \in \mathbb{F}_2^m | S^{-1}(S(x) \oplus \Delta_o) \oplus S^{-1}(S(x \oplus \Delta_i) \oplus \Delta_o) = \Delta_i\}|.$$

For more results concering boomerang connectivity matrix, see [CHPSS18] .

EXAMPLES:

```
sage: from sage.crypto.sboxes import PRESENT
sage: PRESENT.boomerang_connectivity_matrix()
[16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16]
[16  0  4  4  0 16  4  4  4  4  0  0  4  4  0  0]
[16  0  0  6  0  4  6  0  0  0  2  0  2  2  2  0]
[16  2  0  6  2  4  4  2  0  0  2  2  0  0  0  0]
[16  0  0  0  0  4  2  2  0  6  2  0  6  0  2  0]
[16  2  0  0  2  4  0  0  0  6  2  2  4  2  0  0]
[16  4  2  0  4  0  2  0  2  0  0  4  2  0  4  8]
[16  4  2  0  4  0  2  0  2  0  0  4  2  0  4  8]
[16  4  0  2  4  0  0  2  0  2  0  4  0  2  4  8]
[16  4  2  0  4  0  2  0  2  0  0  4  2  0  4  8]
[16  0  2  2  0  4  0  0  6  0  2  0  0  6  2  0]
[16  2  0  0  2  4  0  0  4  2  2  2  0  6  0  0]
```

(continues on next page)

(continued from previous page)

```
[16  0  6  0  0  4  0  6  2  2  2  0  0  0  2  0]
[16  2  4  2  2  4  0  6  0  0  2  2  0  0  0  0]
[16  0  2  2  0  0  2  2  2  2  0  0  2  2  0  0]
[16  8  0  0  8  0  0  0  0  0  0  8  0  0  8 16]
```

cnf (*xi=None*, *yi=None*, *format=None*)

Return a representation of this S-Box in conjunctive normal form.

This function examines the truth tables for each output bit of the S-Box and thus has complexity $n * 2^m$ for an `m x n` S-Box.

INPUT:

- `xi` - indices for the input variables (default: `1...m`)
- `yi` - indices for the output variables (default: `m+1 ... m+n`)
- `format` - output format, see below (default: `None`)

FORMATS:

- `None` - return a list of tuples of integers where each tuple represents a clause, the absolute value of an integer represents a variable and the sign of an integer indicates inversion.
- `symbolic` - a string that can be parsed by the `SymbolicLogic` package.
- `dimacs` - a string in DIMACS format which is the gold standard for SAT-solver input (cf. http://www.satlib.org/).
- `dimacs_headless` - a string in DIMACS format, but without the header. This is useful for concatenation of outputs.

EXAMPLES:

We give a very small example to explain the output format:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(1,2,0,3); S
(1, 2, 0, 3)
sage: cnf = S.cnf(); cnf
[(1, 2, -3),  (1, 2, 4),
 (1, -2, 3),  (1, -2, -4),
 (-1, 2, -3), (-1, 2, -4),
 (-1, -2, 3), (-1, -2, 4)]
```

This output completely describes the S-Box. For instance, we can check that `S([0,1]) -> [1,0]` satisfies every clause if the first input bit corresponds to the index `1` and the last output bit corresponds to the index `3` in the output.

We can convert this representation to the DIMACS format:

```
sage: print(S.cnf(format='dimacs'))
p cnf 4 8
1 2 -3 0
1 2 4 0
1 -2 3 0
1 -2 -4 0
-1 2 -3 0
-1 2 -4 0
-1 -2 3 0
-1 -2 4 0
```

For concatenation we can strip the header:

```
sage: print(S.cnf(format='dimacs_headless'))
1 2 -3 0
1 2 4 0
1 -2 3 0
1 -2 -4 0
-1 2 -3 0
-1 2 -4 0
-1 -2 3 0
-1 -2 4 0
```

This might be helpful in combination with the `xi` and `yi` parameter to assign indices manually:

```
sage: print(S.cnf(xi=[10,20],yi=[30,40], format='dimacs_headless'))
10 20 -30 0
10 20 40 0
10 -20 30 0
10 -20 -40 0
-10 20 -30 0
-10 20 -40 0
-10 -20 30 0
-10 -20 40 0
```

We can also return a string which is parse-able by the `SymbolicLogic` package:

```
sage: log = SymbolicLogic()
sage: s = log.statement(S.cnf(format='symbolic'))
sage: log.truthtable(s)[1:]
[['False', 'False', 'False', 'False', 'False'],
 ['False', 'False', 'False', 'True', 'False'],
 ['False', 'False', 'True', 'False', 'False'],
 ['False', 'False', 'True', 'True', 'True'],
 ['False', 'True', 'False', 'False', 'True'],
 ['False', 'True', 'False', 'True', 'True'],
 ['False', 'True', 'True', 'False', 'True'],
 ['False', 'True', 'True', 'True', 'True'],
 ['True', 'False', 'False', 'False', 'True'],
 ['True', 'False', 'False', 'True', 'True'],
 ['True', 'False', 'True', 'False', 'True'],
 ['True', 'False', 'True', 'True', 'True'],
 ['True', 'True', 'False', 'False', 'True'],
 ['True', 'True', 'False', 'True', 'True'],
 ['True', 'True', 'True', 'False', 'True'],
 ['True', 'True', 'True', 'True', 'True']]
```

This function respects endianness of the S-Box:

```
sage: S = SBox(1,2,0,3, big_endian=False); S
(1, 2, 0, 3)
sage: cnf = S.cnf(); cnf
[(1, 2, -4), (1, 2, 3),
 (-1, 2, 4), (-1, 2, -3),
 (1, -2, -4), (1, -2, -3),
 (-1, -2, 4), (-1, -2, 3)]
```

S-Boxes with m!=n also work:

sage: o = list(range(8)) + list(range(8)) sage: shuffle(o) sage: S = SBox(o) sage:

S.is_permutation() False

sage: len(S.cnf()) == 3*2^4 True

component_function(*b*)

Return a Boolean function corresponding to the component function $b \cdot S(x)$.

If S is an $m \times n$ S-Box, then $b \in \mathbf{F}_2^n$ and $\cdot$ denotes dot product of two vectors.

INPUT:

- `b` – either an integer or a list/tuple/vector of $\mathbf{F}_2$ elements of length `self.output_size()`

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([7,6,0,4,2,5,1,3])
sage: f3 = S.component_function(3)
sage: f3.algebraic_normal_form()
x0*x1 + x0*x2 + x0 + x2

sage: f5 = S.component_function([1, 0, 1])
sage: f5.algebraic_normal_form()
x0*x2 + x0 + x1*x2
```

difference_distribution_matrix()

Return difference distribution matrix `A` for this S-box.

The rows of `A` encode the differences `Delta I` of the input and the columns encode the difference `Delta O` for the output. The bits are ordered according to the endianess of this S-box. The value at `A[Delta I,Delta O]` encodes how often `Delta O` is the actual output difference given `Delta I` as input difference.

See [He2002] for an introduction to differential cryptanalysis.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.difference_distribution_matrix()
[8 0 0 0 0 0 0 0]
[0 2 2 0 2 0 0 2]
[0 0 2 2 0 0 2 2]
[0 2 0 2 2 0 2 0]
[0 2 0 2 0 2 0 2]
[0 0 2 2 2 2 0 0]
[0 2 2 0 0 2 2 0]
[0 0 0 0 2 2 2 2]
```

differential_branch_number()

Return differential branch number of this S-Box.

The differential branch number of an S-Box S is defined as

$$\min_{v,w \neq v} \{\mathrm{wt}(v \oplus w) + \mathrm{wt}(S(v) \oplus S(w))\}$$

where $\mathrm{wt}(x)$ denotes the Hamming weight of vector x.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2])
sage: S.differential_branch_number()
3
```

fixed_points()

Return a list of all fixed points of this S-Box.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([0,1,3,6,7,4,5,2])
sage: S.fixed_points()
[0, 1]
```

from_bits(*x*, *n=None*)

Return integer for bitstring `x` of length `n`.

INPUT:

- `x` - a bitstring
- `n` - bit length (optional)

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.from_bits( [1,1,0])
6

sage: S( S.from_bits( [1,1,0] ) )
1
sage: S.from_bits( S( [1,1,0] ) )
1
```

has_linear_structure()

Return `True` if there exists a nonzero component function of this S-Box that has a linear structure.

See also:

is_linear_structure(), *linear_structures()*.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: S.has_linear_structure()
True
```

input_size()

Return the input size of this S-Box.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([0, 3, 2, 1, 1, 3, 2, 0])
sage: S.input_size()
3
```

interpolation_polynomial(*k=None*)

Return a univariate polynomial over an extension field representing this S-box.

If `m` is the input length of this S-box then the extension field is of degree `m`.

If the output length does not match the input length then a `TypeError` is raised.

INPUT:

- `k` - an instance of $\mathbf{F}_{2^m}$ (default: `None`)

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: f = S.interpolation_polynomial()
sage: f
x^6 + a*x^5 + (a + 1)*x^4 + (a^2 + a + 1)*x^3
  + (a^2 + 1)*x^2 + (a + 1)*x + a^2 + a + 1

sage: a = f.base_ring().gen()

sage: f(0), S(0)
(a^2 + a + 1, 7)

sage: f(a^2 + 1), S(5)
(a^2 + 1, 5)
```

inverse()

Return the inverse of this S-Box.

Note that the S-Box must be invertible, otherwise it will raise a `TypeError`.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([0, 1, 3, 6, 7, 4, 5, 2])
sage: Sinv = S.inverse()
sage: [Sinv(S(i)) for i in range(8)]
[0, 1, 2, 3, 4, 5, 6, 7]
```

is_almost_bent()

Return `True` if this S-Box is an almost bent (AB) function.

An $m \times m$ S-Box S, for m odd, is called almost bent if its nonlinearity is equal to $2^{m-1} - 2^{(m-1)/2}$.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([0,1,3,6,7,4,5,2])
sage: S.is_almost_bent()
True
```

is_apn()

Return `True` if this S-Box is an almost perfect nonlinear (APN) function.

An $m \times m$ S-Box S is called almost perfect nonlinear if for every nonzero $\alpha \in \mathbf{F}_2^m$ and every $\beta \in \mathbf{F}_2^m$, the equation $S(x) \oplus S(x \oplus \alpha) = \beta$ has 0 or 2 solutions. Equivalently, the differential uniformity of S is equal to 2.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([0,1,3,6,7,4,5,2])
sage: S.is_apn()
True
sage: S.differential_uniformity()
2
```

is_balanced()

Return `True` if this S-Box is balanced.

An S-Box is balanced if all its component functions are balanced.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2])
sage: S.is_balanced()
True
```

is_bent()

Return `True` if this S-Box is bent, i.e. its nonlinearity is equal to $2^{m-1} - 2^{m/2-1}$ where m is the input size of the S-Box.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: R.<x> = GF(2**2, 'a')[]
sage: base = R.base_ring()
sage: a = base.gen()
sage: G = a * x^2 + 1
sage: S = SBox([G(x * y**(14)) for x in sorted(base) for y in sorted(base)])
sage: S.is_bent()
True
sage: S.nonlinearity()
6
sage: S.linear_approximation_matrix()
[ 8 -2  2 -2]
[ 0 -2  2 -2]
[ 0 -2  2 -2]
[ 0 -2  2 -2]
[ 0 -2  2 -2]
[ 0 -2 -2  2]
[ 0  2  2  2]
[ 0  2 -2 -2]
[ 0 -2  2 -2]
[ 0  2 -2 -2]
[ 0 -2 -2  2]
[ 0  2  2  2]
[ 0 -2  2 -2]
[ 0  2  2  2]
[ 0  2 -2 -2]
[ 0 -2 -2  2]
```

is_involution()

Return `True` if this S-Box is an involution, i.e. the inverse S-Box is equal itself.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([x**254 for x in sorted(GF(2**8))])
sage: S.is_involution()
True
```

is_linear_structure(*a*, *b*)

Return `True` if a is a linear structure of the component function $b \cdot S(x)$ where S is this $m \times n$ S-Box.

INPUT:

- `a` – either an integer or a tuple of $\mathbf{F}_2$ elements of length equal to the input size of SBox
- `b` – either an integer or a tuple of $\mathbf{F}_2$ elements of length equal to the output size of SBox

See also:

linear_structures(), *has_linear_structure()*.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: S.component_function(1).autocorrelation()
(16, -16, 0, 0, 0, 0, 0, 0, -16, 16, 0, 0, 0, 0, 0, 0)
sage: S.is_linear_structure(1, 1)
True
sage: S.is_linear_structure([1, 0, 0, 1], [0, 0, 0, 1])
True
sage: S.is_linear_structure([0, 1, 1, 1], 1)
False
```

is_monomial_function()

Return `True` if this S-Box is a monomial/power function.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([0,1,3,6,7,4,5,2])
sage: S.is_monomial_function()
False
sage: S.interpolation_polynomial()
(a + 1)*x^6 + (a^2 + a + 1)*x^5 + (a^2 + 1)*x^3

sage: S = SBox(0,1,5,6,7,2,3,4)
sage: S.is_monomial_function()
True
sage: S.interpolation_polynomial()
x^6
```

is_permutation()

Return `True` if this S-Box is a permutation.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.is_permutation()
True

sage: S = SBox(3,2,0,0,2,1,1,3)
```

(continues on next page)

(continued from previous page)

```
sage: S.is_permutation()
False
```

is_plateaued()

Return `True` if this S-Box is plateaued, i.e. for all nonzero $b \in \mathbb{F}_2^n$ the Boolean function $b \cdot S(x)$ is plateaued.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(0, 3, 1, 2, 4, 6, 7, 5)
sage: S.is_plateaued()
True
```

linear_approximation_matrix(*scale='absolute_bias'*)

Return linear approximation matrix (LAM) A for this S-box.

The entry $A[\alpha, \beta]$ corresponds to the probability $Pr[\alpha \cdot x = \beta \cdot S(x)]$, where S is this S-box mapping n-bit inputs to m-bit outputs. There are three typical notations for this probability used in the literature:

- $Pr[\alpha \cdot x = \beta \cdot S(x)] = 1/2 + e(\alpha, \beta)$, where $e(\alpha, \beta)$ is called the bias,
- $2 \cdot Pr[\alpha \cdot x = \beta \cdot S(x)] = 1 + c(\alpha, \beta)$, where $c(\alpha, \beta) = 2 \cdot e(\alpha, \beta)$ is the correlation, and
- $2^{(m+1)} \cdot Pr[\alpha \cdot x = \beta \cdot S(x)] = 2^m + \hat{S}(\alpha, \beta)$, where $\hat{S}(\alpha, \beta)$ is the Fourier coefficient of S.

See [He2002] for an introduction to linear cryptanalysis.

INPUT:

- **scale - string to choose the scaling for the LAM, one of**
  - "bias": elements are $e(\alpha, \beta)$
  - "correlation": elements are $c(\alpha, \beta)$
  - "absolute_bias": elements are $2^m \cdot e(\alpha, \beta)$ (default)
  - "fourier_coefficient": elements are $\hat{S}(\alpha, \beta)$

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: lat_abs_bias = S.linear_approximation_matrix()
sage: lat_abs_bias
[ 4  0  0  0  0  0  0  0]
[ 0  0  0  0  2  2  2 -2]
[ 0  0 -2 -2 -2  2  0  0]
[ 0  0 -2  2  0  0 -2 -2]
[ 0  2  0  2 -2  0  2  0]
[ 0 -2  0  2  0  2  0  2]
[ 0 -2 -2  0  0 -2  2  0]
[ 0 -2  2  0 -2  0  0 -2]

sage: lat_abs_bias/(1<<S.m) == S.linear_approximation_matrix(scale="bias")
True

sage: lat_abs_bias/(1<<(S.m-1)) == S.linear_approximation_matrix(scale=
↪"correlation")
True
```

(continues on next page)

(continued from previous page)

```
sage: lat_abs_bias*2 == S.linear_approximation_matrix(scale="fourier_
↪coefficient")
True
```

According to this matrix the first bit of the input is equal to the third bit of the output 6 out of 8 times:

```
sage: for i in srange(8): print(S.to_bits(i)[0] == S.to_bits(S(i))[2])
False
True
True
True
False
True
True
True
```

linear_branch_number()

Return linear branch number of this S-Box.

The linear branch number of an S-Box S is defined as

$$\min_{\substack{\alpha\neq 0,\beta \\ \mathrm{LAM}(\alpha,\beta)\neq 0}} \{\mathrm{wt}(\alpha)+\mathrm{wt}(\beta)\}$$

where $\mathrm{LAM}(\alpha,\beta)$ is the entry at row α and column β of linear approximation matrix correspond to this S-Box. The $\mathrm{wt}(x)$ denotes the Hamming weight of x.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2])
sage: S.linear_branch_number()
2
```

linear_structures()

Return a list of 3-valued tuple (b,α,c) such that α is a c-linear structure of the component function $b\cdot S(x)$.

A Boolean function $f:\mathbf{F}_2^m\mapsto\mathbf{F}_2$ is said to have a c-linear structure if there exists a nonzero α such that $f(x)\oplus f(x\oplus\alpha)$ is a constant function c.

An $m\times n$ S-Box S has a linear structure if there exists a component function $b\cdot S(x)$ that has a linear structure.

The three valued tuple (b,α,c) shows that α is a c-linear structure of the component function $b\cdot S(x)$. This implies that for all output differences β of the S-Box correspond to input difference α, we have $b\cdot\beta=c$.

See also:

is_linear_structure(), *has_linear_structure()*.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([0,1,3,6,7,4,5,2])
sage: S.linear_structures()
[(1, 1, 1), (2, 2, 1), (3, 3, 1), (4, 4, 1), (5, 5, 1), (6, 6, 1), (7, 7, 1)]
```

linearity()

Return the linearity of this S-Box.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = mq.SR(1, 4, 4, 8).sbox()
sage: S.linearity()
32
```

max_degree()

Return the maximal algebraic degree of all its component functions.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2])
sage: S.max_degree()
3
```

maximal_difference_probability()

Return the difference probability of the difference with the highest probability in the range between 0.0 and 1.0 indicating 0% or 100% respectively.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.maximal_difference_probability()
0.25
```

maximal_difference_probability_absolute()

Return the difference probability of the difference with the highest probability in absolute terms, i.e. how often it occurs in total.

Equivalently, this is equal to the differential uniformity of this S-Box.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.maximal_difference_probability_absolute()
2
```

Note: This code is mainly called internally.

---

maximal_linear_bias_absolute()

Return maximal linear bias, i.e. how often the linear approximation with the highest bias is true or false minus 2^{n-1}.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.maximal_linear_bias_absolute()
2
```

maximal_linear_bias_relative()

Return maximal bias of all linear approximations of this S-box.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.maximal_linear_bias_relative()
0.25
```

min_degree()

Return the minimal algebraic degree of all its component functions.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2])
sage: S.min_degree()
2
```

nonlinearity()

Return the nonlinearity of this S-Box.

The nonlinearity of an S-Box is defined as the minimum nonlinearity of all its component functions.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = mq.SR(1,4,4,8).sbox()
sage: S.nonlinearity()
112
```

output_size()

Return the output size of this S-Box.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([0, 3, 2, 1, 1, 3, 2, 0])
sage: S.output_size()
2
```

polynomials(*X=None*, *Y=None*, *degree=2*, *groebner=False*)

Return a list of polynomials satisfying this S-box.

First, a simple linear fitting is performed for the given `degree` (cf. for example [BC2003]). If `groebner=True` a Groebner basis is also computed for the result of that process.

INPUT:

- `X` - input variables
- `Y` - output variables
- `degree` - integer > 0 (default: `2`)
- `groebner` - calculate a reduced Groebner basis of the spanning polynomials to obtain more polynomials (default: `False`)

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: P = S.ring()
```

By default, this method returns an indirect representation:

```
sage: S.polynomials()
[x0*x2 + x1 + y1 + 1,
 x0*x1 + x1 + x2 + y0 + y1 + y2 + 1,
 x0*y1 + x0 + x2 + y0 + y2,
 x0*y0 + x0*y2 + x1 + x2 + y0 + y1 + y2 + 1,
 x1*x2 + x0 + x1 + x2 + y2 + 1,
 x0*y0 + x1*y0 + x0 + x2 + y1 + y2,
 x0*y0 + x1*y1 + x1 + y1 + 1,
 x1*y2 + x1 + x2 + y0 + y1 + y2 + 1,
 x0*y0 + x2*y0 + x1 + x2 + y1 + 1,
 x2*y1 + x0 + y1 + y2,
 x2*y2 + x1 + y1 + 1,
 y0*y1 + x0 + x2 + y0 + y1 + y2,
 y0*y2 + x1 + x2 + y0 + y1 + 1,
 y1*y2 + x2 + y0]
```

We can get a direct representation by computing a lexicographical Groebner basis with respect to the right variable ordering, i.e. a variable ordering where the output bits are greater than the input bits:

```
sage: P.<y0,y1,y2,x0,x1,x2> = PolynomialRing(GF(2),6,order='lex')
sage: S.polynomials([x0,x1,x2],[y0,y1,y2], groebner=True)
[y0 + x0*x1 + x0*x2 + x0 + x1*x2 + x1 + 1,
 y1 + x0*x2 + x1 + 1,
 y2 + x0 + x1*x2 + x1 + x2 + 1]
```

ring()

Create, return and cache a polynomial ring for S-box polynomials.

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.ring()
Multivariate Polynomial Ring in x0, x1, x2, y0, y1, y2 over Finite Field of␣
↪size 2
```

solutions(*X=None*, *Y=None*)

Return a dictionary of solutions to this S-box.

INPUT:

- `X` - input variables (default: `None`)
- `Y` - output variables (default: `None`)

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox([7,6,0,4,2,5,1,3])
sage: F = S.polynomials()
sage: s = S.solutions()
sage: any(f.subs(_s) for f in F for _s in s)
False
```

to_bits(*x*, *n=None*)

Return bitstring of length `n` for integer `x`. The returned bitstring is guaranteed to have length `n`.

INPUT:

- `x` - an integer

- n - bit length (optional)

EXAMPLES:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(7,6,0,4,2,5,1,3)
sage: S.to_bits(6)
[1, 1, 0]

sage: S.to_bits( S(6) )
[0, 0, 1]

sage: S( S.to_bits( 6 ) )
[0, 0, 1]
```

sage.crypto.sbox.**feistel_construction**(**args*)

Return an S-Box constructed by Feistel structure using smaller S-Boxes in args. The number of round in the construction is equal to the number of S-Boxes provided as input. For more results concerning the differential uniformity and the nonlinearity of S-Boxes constructed by Feistel structures see [CDL2015] .

INPUT:

- args - a finite iterable SBox objects

EXAMPLES:

Suppose we construct an 8×8 S-Box with 3-round Feistel construction from the S-Box of PRESENT:

```
sage: from sage.crypto.sbox import SBox
sage: s = SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: from sage.crypto.sbox import feistel_construction
sage: S = feistel_construction(s, s, s)
```

The properties of the constructed S-Box can be easily examined:

```
sage: S.nonlinearity()
96
sage: S.differential_branch_number()
2
sage: S.linear_branch_number()
2
```

sage.crypto.sbox.**misty_construction**(**args*)

Return an S-Box constructed by MISTY structure using smaller S-Boxes in args. The number of round in the construction is equal to the number of S-Boxes provided as input. For further result related to the nonlinearity and differential uniformity of the constructed S-Box one may consult [CDL2015].

INPUT:

- args - a finite iterable SBox objects

EXAMPLES:

We construct an 8×8 S-Box using 3-round MISTY structure with the following 4×4 S-Boxes $S1, S2, S3$ (see Example 2 in [CDL2015]):

```
sage: from sage.crypto.sbox import SBox
sage: S1 = SBox([0x4,0x0,0x1,0xF,0x2,0xB,0x6,0x7,0x3,0x9,0xA,0x5,0xC,0xD,0xE,0x8])
sage: S2 = SBox([0x0,0x0,0x0,0x1,0x0,0xA,0x8,0x3,0x0,0x8,0x2,0xB,0x4,0x6,0xE,0xD])
sage: S3 = SBox([0x0,0x7,0xB,0xD,0x4,0x1,0xB,0xF,0x1,0x2,0xC,0xE,0xD,0xC,0x5,0x5])
```

(continues on next page)

(continued from previous page)

```
sage: from sage.crypto.sbox import misty_construction
sage: S = misty_construction(S1, S2, S3)
sage: S.differential_uniformity()
8
sage: S.linearity()
64
```

CHAPTER

FOURTEEN

ABSTRACT BASE CLASS FOR GENERATORS OF POLYNOMIAL SYSTEMS.

AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>

class sage.crypto.mq.mpolynomialsystemgenerator.**MPolynomialSystemGenerator**

Bases: sage.structure.sage_object.SageObject

Abstract base class for generators of polynomial systems.

block_order()

Return a block term ordering for the equation systems generated by self.

EXAMPLES:

```
sage: from sage.crypto.mq.mpolynomialsystemgenerator import␣
↪MPolynomialSystemGenerator
sage: msg = MPolynomialSystemGenerator()
sage: msg.block_order()
Traceback (most recent call last):
...
NotImplementedError
```

polynomial_system(*P=None*, *K=None*)

Return a tuple F,s for plaintext P and key K where F is an polynomial system and s a dictionary which maps key variables to their solutions.

INPUT: P – plaintext (vector, list) K – key (vector, list)

EXAMPLES:

```
sage: from sage.crypto.mq.mpolynomialsystemgenerator import␣
↪MPolynomialSystemGenerator
sage: msg = MPolynomialSystemGenerator()
sage: msg.polynomial_system()
Traceback (most recent call last):
...
NotImplementedError
```

random_element()

Return random element. Usually this is a list of elements in the base field of length 'blocksize'.

EXAMPLES:

```
sage: from sage.crypto.mq.mpolynomialsystemgenerator import␣
↪MPolynomialSystemGenerator
sage: msg = MPolynomialSystemGenerator()
```

(continues on next page)

(continued from previous page)

```
sage: msg.random_element()
Traceback (most recent call last):
...
NotImplementedError
```

ring()

Return the ring in which the system is defined.

EXAMPLES:

```
sage: from sage.crypto.mq.mpolynomialsystemgenerator import␣
↪MPolynomialSystemGenerator
sage: msg = MPolynomialSystemGenerator()
sage: msg.ring()
Traceback (most recent call last):
...
NotImplementedError
```

sbox()

Return SBox object for self.

EXAMPLES:

```
sage: from sage.crypto.mq.mpolynomialsystemgenerator import␣
↪MPolynomialSystemGenerator
sage: msg = MPolynomialSystemGenerator()
sage: msg.sbox()
Traceback (most recent call last):
...
AttributeError: '<class 'sage.crypto.mq.mpolynomialsystemgenerator.
↪MPolynomialSystemGenerator'>' object has no attribute '_sbox'
```

varformatstr(*name*)

Return format string for a given name 'name' which is understood by print et al.

Such a format string is used to construct variable names. Typically those format strings are somewhat like 'name%02d%02d' such that rounds and offset in a block can be encoded.

INPUT: name – string

EXAMPLES:

```
sage: from sage.crypto.mq.mpolynomialsystemgenerator import␣
↪MPolynomialSystemGenerator
sage: msg = MPolynomialSystemGenerator()
sage: msg.varformatstr('K')
Traceback (most recent call last):
...
NotImplementedError
```

vars(*name*, *round*)

Return a list of variables given a name 'name' and an index 'round'.

INPUT: name – string round – integer index

EXAMPLES:

```
sage: from sage.crypto.mq.mpolynomialsystemgenerator import␣
↪MPolynomialSystemGenerator
sage: msg = MPolynomialSystemGenerator()
sage: msg.vars('K',0)
Traceback (most recent call last):
...
NotImplementedError
```

varstrs(*name*, *round*)

Return a list of variable names given a name 'name' and an index 'round'.

This function is typically used by self._vars.

INPUT: name – string round – integer index

EXAMPLES:

```
sage: from sage.crypto.mq.mpolynomialsystemgenerator import␣
↪MPolynomialSystemGenerator
sage: msg = MPolynomialSystemGenerator()
sage: msg.varstrs('K', i)
Traceback (most recent call last):
...
NotImplementedError
```

CHAPTER

FIFTEEN

SMALL SCALE VARIANTS OF THE AES (SR) POLYNOMIAL SYSTEM GENERATOR

Sage supports polynomial system generation for small scale (and full scale) AES variants over $\mathbf{F}_2$ and $\mathbf{F}_{2^e}$. Also, Sage supports both the specification of SR as given in the papers [CMR2005] and [CMR2006] and a variant of SR* which is equivalent to AES.

SR is a family of parameterizable variants of the AES suitable as a framework for comparing different cryptanalytic techniques that can be brought to bear on the AES. It is different from *Mini-AES*, whose purpose is as a teaching tool to help beginners understand the basic structure and working of the full AES.

AUTHORS:

- Martin Albrecht (2008,2009-01): usability improvements
- Martin Albrecht (2007-09): initial version
- Niles Johnson (2010-08): (trac ticket #3893) `random_element()` should pass on `*args` and `**kwds`.

EXAMPLES:

We construct SR(1,1,1,4) and study its properties.

```
sage: sr = mq.SR(1, 1, 1, 4)
```

`n` is the number of rounds, `r` the number of rows in the state array, `c` the number of columns in the state array, and `e` the degree of the underlying field.

```
sage: sr.n, sr.r, sr.c, sr.e
(1, 1, 1, 4)
```

By default variables are ordered reverse to as they appear, e.g.:

```
sage: print(sr.R.repr_long())
Polynomial Ring
  Base Ring : Finite Field in a of size 2^4
       Size : 20 Variables
   Block  0 : Ordering : deglex
              Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100, w101,␣
↪w102, w103, s000, s001, s002, s003, k000, k001, k002, k003
```

However, this can be prevented by passing in `reverse_variables=False` to the constructor.

For SR(1, 1, 1, 4) the `ShiftRows` matrix isn't that interesting.:

```
sage: sr.ShiftRows
[1 0 0 0]
[0 1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0]
[0 0 0 1]
```

Also, the `MixColumns` matrix is the identity matrix.:

```
sage: sr.MixColumns
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

`Lin`, however, is not the identity matrix.:

```
sage: sr.Lin
[          a^2 + 1                 1         a^3 + a^2           a^2 + 1]
[               a                  a                 1 a^3 + a^2 + a + 1]
[          a^3 + a               a^2               a^2                 1]
[               1               a^3             a + 1             a + 1]
```

`M` and `Mstar` are identical for SR(1, 1, 1, 4):

```
sage: sr.M
[          a^2 + 1                 1         a^3 + a^2           a^2 + 1]
[               a                  a                 1 a^3 + a^2 + a + 1]
[          a^3 + a               a^2               a^2                 1]
[               1               a^3             a + 1             a + 1]
```

```
sage: sr.Mstar
[          a^2 + 1                 1         a^3 + a^2           a^2 + 1]
[               a                  a                 1 a^3 + a^2 + a + 1]
[          a^3 + a               a^2               a^2                 1]
[               1               a^3             a + 1             a + 1]
```

However, for larger instances of SR Mstar is not equal to M:

```
sage: sr = mq.SR(10,4,4,8)
sage: sr.Mstar == ~sr.MixColumns * sr.M
True
```

We can compute a Groebner basis for the ideals spanned by SR instances to recover all solutions to the system.:

```
sage: sr = mq.SR(1,1,1,4, gf2=True, polybori=True)
sage: K = sr.base_ring()
sage: a = K.gen()
sage: K = [a]
sage: P = [1]
sage: F,s = sr.polynomial_system(P=P, K=K)
sage: F.groebner_basis()
[k100, k101 + 1, k102, k103 + k003,
 x100 + 1, x101 + k003 + 1, x102 + k003 + 1,
 x103 + k003, w100, w101, w102 + 1, w103 + k003 + 1,
 s000 + 1, s001 + k003, s002 + k003, s003 + k003 + 1,
 k000, k001, k002 + 1]
```

Note that the order of `k000`, `k001`, `k002` and `k003` is little endian. Thus the result `k002 + 1, k001, k000` indicates that the key is either a or $a + 1$. We can verify that both keys encrypt P to the same ciphertext:

```
sage: sr(P,[a])
[0]
sage: sr(P,[a+1])
[0]
```

All solutions can easily be recovered using the variety function for ideals.:

```
sage: I = F.ideal()
sage: for V in I.variety():
....:     for k,v in sorted(V.items()):
....:         print("{} {}".format(k, v))
....:     print("\n")
k003 0
k002 1
k001 0
k000 0
s003 1
s002 0
s001 0
s000 1
w103 1
w102 1
w101 0
w100 0
x103 0
x102 1
x101 1
x100 1
k103 0
k102 0
k101 1
k100 0

k003 1
k002 1
k001 0
k000 0
s003 0
s002 1
s001 1
s000 1
w103 0
w102 1
w101 0
w100 0
x103 1
x102 0
x101 0
x100 1
k103 1
k102 0
k101 1
k100 0
```

We can also verify the correctness of the variety by evaluating all ideal generators on all points.:

```
sage: for V in I.variety():
```

(continues on next page)

(continued from previous page)

```
....:     for f in I.gens():
....:         if f.subs(V) != 0:
....:             print("epic fail")
```

Note that the S-Box object for SR can be constructed with a call to `sr.sbox()`:

```
sage: sr = mq.SR(1,1,1,4, gf2=True, polybori=True)
sage: S = sr.sbox()
```

For example, we can now study the difference distribution matrix of `S`:

```
sage: S.difference_distribution_matrix()
[16  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  2  2  2  2  0  0  0  2  0  0  0  2  4  0  0]
[ 0  2  0  4  2  2  2  0  0  2  0  0  0  0  0  2]
[ 0  2  4  0  0  2  0  0  2  2  0  2  0  0  2  0]
[ 0  0  2  0  4  2  0  0  0  0  2  0  2  0  2  2]
[ 0  0  0  2  0  0  0  2  4  2  0  0  2  0  2  2]
[ 0  4  0  0  0  2  0  2  0  2  2  0  2  2  0  0]
[ 0  2  0  0  0  0  2  0  0  0  0  2  4  2  2  2]
[ 0  2  2  0  0  0  2  2  2  0  2  0  0  0  0  4]
[ 0  0  2  2  0  0  0  0  0  2  2  4  0  2  0  2]
[ 0  0  2  0  2  0  2  2  0  4  0  2  2  0  0  0]
[ 0  0  0  0  2  0  2  0  2  2  4  0  0  2  2  0]
[ 0  0  0  2  0  4  2  0  2  0  2  2  2  0  0  0]
[ 0  0  0  0  2  2  0  4  2  0  0  2  0  2  0  2]
[ 0  0  2  2  0  2  4  2  0  0  0  0  0  2  2  0]
[ 0  2  0  2  2  0  0  2  0  0  2  2  0  0  4  0]
```

or use `S` to find alternative polynomial representations for the S-Box.:

```
sage: S.polynomials(degree=3)
[x0*x1 + x1*x2 + x0*x3 + x0*y2 + x1 + y0 + y1 + 1,
 x0*x1 + x0*x2 + x0*y0 + x0*y1 + x0*y2 + x1 + x2 + y0 + y1 + y2,
 x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x1*y0 + x0*y1 + x0*y3,
 x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x1*y1 + x0*y3 + x1 + y0 + y1 + 1,
 x0*x1 + x0*x2 + x0*y2 + x1*y2 + x0*y3 + x0 + x1,
 x0*x3 + x1*x3 + x0*y1 + x0*y2 + x1*y3 + x0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
 x0*x1 + x1*x3 + x2*x3 + x0*y0 + x0*y2 + x0*y3 + x2 + y0 + y3,
 x0*x1 + x0*x2 + x0*x3 + x1*x3 + x2*y0 + x0*y2 + x0 + x2 + x3 + y3,
 x0*x3 + x1*x3 + x0*y0 + x2*y1 + x0*y2 + x3 + y3,
 x0*x1 + x0*x2 + x0*y0 + x0*y1 + x2*y2 + x0*y3 + x1 + y0 + y1 + 1,
 x0*x3 + x1*x3 + x0*y0 + x0*y1 + x0*y3 + x2*y3 + y0 + y3,
 x0*x1 + x0*x2 + x3*y0 + x0*y1 + x0*y3 + y0,
 x0*y0 + x0*y1 + x3*y1 + x0 + x2 + y0 + y3,
 x0*y0 + x3*y2 + y0,
 x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x0*y2 + x3*y3 + y0,
 x0*x2 + x0*x3 + x0*y1 + y0*y1 + x0*y3 + x2 + x3 + y3,
 x0*x2 + x0*y0 + y0*y2 + x0*y3 + x0 + y0,
 x0*x1 + x0*x2 + x1*x3 + x0*y2 + y0*y3 + y0,
 x0*x1 + x0*y0 + y1*y2 + x0*y3 + x1 + x2 + y0 + 1,
 x0*x2 + x1*x3 + x0*y1 + x0*y2 + x0*y3 + y1*y3 + x0 + y0 + y3,
 x0*x1 + x0*x2 + x0*x3 + x0*y1 + x0*y2 + x0*y3 + y2*y3 + x0 + x1 + x2 + x3 + y1 + y3␣
↪+ 1,
 x0*x1*x2 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0,
 x0*x1*x3 + x0*x2 + x0*x3 + x0*y1 + x0*y3 + x0,
 x0*x1*y0 + x0*x1 + x0*y0 + x0,
```

(continues on next page)

(continued from previous page)

```
 x0*x1*y1,
 x0*x1*y2 + x0*x2 + x0*y2 + x0*y3 + x0,
 x0*x1*y3 + x0*x1 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0,
 x0*x2*x3 + x0*x1 + x0*x3 + x0*y1 + x0*y2 + x0*y3 + x0,
 x0*x2*y0 + x0*x1 + x0*x2 + x0*x3 + x0*y1 + x0*y2,
 x0*x2*y1 + x0*x2 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0,
 x0*x2*y2 + x0*x2 + x0*y3 + x0,
 x0*x2*y3 + x0*x2 + x0*y3 + x0,
 x0*x3*y0 + x0*x1 + x0*x2 + x0*y0 + x0*y1 + x0*y3,
 x0*x3*y1 + x0*x2 + x0*y1 + x0*y3 + x0,
 x0*x3*y2,
 x0*x3*y3 + x0*x1 + x0*y1 + x0*y2 + x0*y3 + x0,
 x0*y0*y1 + x0*y1,
 x0*y0*y2 + x0*x2 + x0*y3 + x0,
 x0*y0*y3 + x0*x1 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0*y3 + x0,
 x0*y1*y2 + x0*x2 + x0*y3 + x0,
 x0*y1*y3 + x0*x3 + x0*y0 + x0*y2 + x0*y3,
 x0*y2*y3 + x0*y2,
 x1*x2*x3 + x0*x1 + x1*x3 + x0*y0 + x0*y1 + x2 + x3 + y3,
 x1*x2*y0 + x0*x1 + x1*x3 + x0*y0 + x0*y1 + x2 + x3 + y3,
 x1*x2*y1 + x0*x1 + x1*x3 + x0*y0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
 x1*x2*y2 + x0*x1 + x0*y0 + x0*y1 + x0 + x1 + y0 + y1 + 1,
 x1*x2*y3 + x0*x1 + x1*x3 + x0*y0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
 x1*x3*y0 + x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x0*y1 + x0*y3,
 x1*x3*y1 + x0*x2 + x0*x3 + x0*y3 + x2 + x3 + y3,
 x1*x3*y2 + x0*x2 + x0*x3 + x1*x3 + x0*y1 + x0*y3 + x0,
 x1*x3*y3 + x0*x1 + x0*x2 + x0*x3 + x0*y0 + x0*y1 + x0*y3,
 x1*y0*y1 + x0*x2 + x0*x3 + x0*y3 + x2 + x3 + y3,
 x1*y0*y2 + x0*x2 + x0*x3 + x1*x3 + x0*y1 + x0*y3 + x0,
 x1*y0*y3,
 x1*y1*y2 + x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x0*y3 + x1 + y0 + y1 + 1,
 x1*y1*y3 + x0*x1 + x1*x3 + x0*y0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
 x1*y2*y3 + x0*x1 + x0*x2 + x1*x3 + x0*y0 + x0*y2 + x0*y3 + x0 + x1 + x2 + x3 + y0 +␣
↪y1 + y3 + 1,
 x2*x3*y0 + x0*x1 + x0*x3 + x1*x3 + x0*y2 + x0*y3 + x2 + x3 + y3,
 x2*x3*y1 + x0*y1 + x0*y2 + x0*y3 + x3 + y0,
 x2*x3*y2 + x1*x3 + x0*y1 + x0 + x2 + x3 + y3,
 x2*x3*y3,
 x2*y0*y1 + x0*x2 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0,
 x2*y0*y2 + x0*x2 + x1*x3 + x0*y1 + x0*y3 + x2 + x3 + y3,
 x2*y0*y3 + x0*x2 + x0*y3 + x0,
 x2*y1*y2 + x0*x1 + x0*x2 + x1*x3 + x0*y0 + x0*y3 + x0 + x1 + x2 + x3 + y0 + y1 + y3␣
↪+ 1,
 x2*y1*y3 + x0*x3 + x1*x3 + x0*y0 + x0*y1 + x0*y3 + y0 + y3,
 x2*y2*y3 + x0*x1 + x0*x2 + x1*x3 + x0*y0 + x0*y3 + x0 + x1 + x2 + x3 + y0 + y1 + y3␣
↪+ 1,
 x3*y0*y1 + x0*x3 + x0*y1 + x0 + x2 + x3 + y3,
 x3*y0*y2 + x0*y0 + y0,
 x3*y0*y3 + x1*x3 + x0*y1 + x0*y2 + x0*y3 + y0,
 x3*y1*y2 + x0*x2 + x0*x3 + x0*y3 + x2 + x3 + y3,
 x3*y1*y3 + x0*x2 + x0*x3 + x0*y0 + x0*y2 + x0,
 x3*y2*y3 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x0*y1 + x0*y3 + x0 + y0,
 y0*y1*y2 + x0*x3 + x0 + x2 + x3 + y3,
 y0*y1*y3 + x0*x3 + x0*y0 + x0*y2 + x0*y3,
 y0*y2*y3 + x0*x3 + x1*x3 + x0*y0 + x0*y1 + y0,
 y1*y2*y3 + x0*x1 + x0*x2 + x1*x3 + x0*y0 + x0*y3 + x0 + x1 + x2 + x3 + y0 + y1 + y3␣
↪+ 1]
```

(continues on next page)

(continued from previous page)

```
sage: S.interpolation_polynomial()
(a^2 + 1)*x^14 + x^13 + (a^3 + a^2)*x^11 + (a^2 + 1)*x^7 + a^2 + a
```

The *SR_gf2_2* gives an example how use alternative polynomial representations of the S-Box for construction of polynomial systems.

REFERENCES:

- [CMR2005]
- [CMR2006]
- [MR2002]

class sage.crypto.mq.sr.**AllowZeroInversionsContext**(*sr*)
Temporarily allow zero inversion.

sage.crypto.mq.sr.**SR**(*n=1*, *r=1*, *c=1*, *e=4*, *star=False*, ***kwargs*)
Return a small scale variant of the AES polynomial system constructor subject to the following conditions:

INPUT:

- `n` - the number of rounds (default: 1)
- `r` - the number of rows in the state array (default: 1)
- `c` - the number of columns in the state array (default: 1)
- `e` - the exponent of the finite extension field (default: 4)
- `star` - determines if SR* or SR should be constructed (default: `False`)
- `aes_mode` - as the SR key schedule specification differs slightly from the AES key schedule, this parameter controls which schedule to use (default: `True`)
- `gf2` - generate polynomial systems over $\mathbf{F}_2$ rather than over $\mathbf{F}_{2^e}$ (default: `False`)
- `polybori` - use the `BooleanPolynomialRing` as polynomial representation (default: `True`, $\mathbf{F}_2$ only)
- `order` - a string to specify the term ordering of the variables (default: `deglex`)
- `postfix` - a string which is appended after the variable name (default: '')
- `allow_zero_inversions` - a boolean to control whether zero inversions raise an exception (default: `False`)
- `correct_only` - only include correct inversion polynomials (default: `False`, $\mathbf{F}_2$ only)
- `biaffine_only` - only include bilinear and biaffine inversion polynomials (default: `True`, $\mathbf{F}_2$ only)

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 4)
sage: ShiftRows = sr.shift_rows_matrix()
sage: MixColumns = sr.mix_columns_matrix()
sage: Lin = sr.lin_matrix()
sage: M = MixColumns * ShiftRows * Lin
sage: print(sr.hex_str_matrix(M))
 5 1 C 5
 2 2 1 F
 A 4 4 1
 1 8 3 3
```

```
sage: sr = mq.SR(1, 2, 1, 4)
sage: ShiftRows = sr.shift_rows_matrix()
sage: MixColumns = sr.mix_columns_matrix()
sage: Lin = sr.lin_matrix()
sage: M = MixColumns * ShiftRows * Lin
sage: print(sr.hex_str_matrix(M))
 F 3 7 F A 2 B A
 A A 5 6 8 8 4 9
 7 8 8 2 D C C 3
 4 6 C C 5 E F F
 A 2 B A F 3 7 F
 8 8 4 9 A A 5 6
 D C C 3 7 8 8 2
 5 E F F 4 6 C C
```

```
sage: sr = mq.SR(1, 2, 2, 4)
sage: ShiftRows = sr.shift_rows_matrix()
sage: MixColumns = sr.mix_columns_matrix()
sage: Lin = sr.lin_matrix()
sage: M = MixColumns * ShiftRows * Lin
sage: print(sr.hex_str_matrix(M))
 F 3 7 F 0 0 0 0 0 0 0 0 A 2 B A
 A A 5 6 0 0 0 0 0 0 0 0 8 8 4 9
 7 8 8 2 0 0 0 0 0 0 0 0 D C C 3
 4 6 C C 0 0 0 0 0 0 0 0 5 E F F
 A 2 B A 0 0 0 0 0 0 0 0 F 3 7 F
 8 8 4 9 0 0 0 0 0 0 0 0 A A 5 6
 D C C 3 0 0 0 0 0 0 0 0 7 8 8 2
 5 E F F 0 0 0 0 0 0 0 0 4 6 C C
 0 0 0 0 A 2 B A F 3 7 F 0 0 0 0
 0 0 0 0 8 8 4 9 A A 5 6 0 0 0 0
 0 0 0 0 D C C 3 7 8 8 2 0 0 0 0
 0 0 0 0 5 E F F 4 6 C C 0 0 0 0
 0 0 0 0 F 3 7 F A 2 B A 0 0 0 0
 0 0 0 0 A A 5 6 8 8 4 9 0 0 0 0
 0 0 0 0 7 8 8 2 D C C 3 0 0 0 0
 0 0 0 0 4 6 C C 5 E F F 0 0 0 0
```

class sage.crypto.mq.sr.**SR_generic**(*n=1*, *r=1*, *c=1*, *e=4*, *star=False*, ***kwargs*)

Bases: *sage.crypto.mq.mpolynomialsystemgenerator.MPolynomialSystemGenerator*

Small Scale Variants of the AES.

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 4)
sage: ShiftRows = sr.shift_rows_matrix()
sage: MixColumns = sr.mix_columns_matrix()
sage: Lin = sr.lin_matrix()
sage: M = MixColumns * ShiftRows * Lin
sage: print(sr.hex_str_matrix(M))
 5 1 C 5
 2 2 1 F
 A 4 4 1
 1 8 3 3
```

add_round_key(*d*, *key*)

Perform the `AddRoundKey` operation on `d` using `key`.

INPUT:

- `d` - state array or something coercible to a state array
- `key` - state array or something coercible to a state array

EXAMPLES:

```
sage: sr = mq.SR(10, 4, 4, 4)
sage: D = sr.random_state_array()
sage: K = sr.random_state_array()
sage: sr.add_round_key(D, K) == K + D
True
```

base_ring()

Return the base field of self as determined by `self.e`.

EXAMPLES:

```
sage: sr = mq.SR(10, 2, 2, 4)
sage: sr.base_ring().polynomial()
a^4 + a + 1
```

The Rijndael polynomial:

```
sage: sr = mq.SR(10, 4, 4, 8)
sage: sr.base_ring().polynomial()
a^8 + a^4 + a^3 + a + 1
```

block_order()

Return a block order for self where each round is a block.

EXAMPLES:

```
sage: sr = mq.SR(2, 1, 1, 4)
sage: sr.block_order()
Block term order with blocks:
(Degree lexicographic term order of length 16,
 Degree lexicographic term order of length 16,
 Degree lexicographic term order of length 4)
```

```
sage: P = sr.ring(order='block')
sage: print(P.repr_long())
Polynomial Ring
  Base Ring : Finite Field in a of size 2^4
       Size : 36 Variables
   Block  0 : Ordering : deglex
              Names    : k200, k201, k202, k203, x200, x201, x202, x203, w200,
↪ w201, w202, w203, s100, s101, s102, s103
   Block  1 : Ordering : deglex
              Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100,
↪ w101, w102, w103, s000, s001, s002, s003
   Block  2 : Ordering : deglex
              Names    : k000, k001, k002, k003
```

hex_str(*M*, *typ='matrix'*)

Return a hex string for the provided AES state array/matrix.

INPUT:

- `M` - state array

- `typ` - controls what to return, either 'matrix' or 'vector' (default: `'matrix'`)

EXAMPLES:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: k = sr.base_ring()
sage: A = matrix(k, 2, 2, [1, k.gen(), 0, k.gen()^2])
sage: sr.hex_str(A)
' 1 2 \n 0 4 \n'
```

```
sage: sr.hex_str(A, typ='vector')
'1024'
```

hex_str_matrix(*M*)

Return a two-dimensional AES-like representation of the matrix M.

That is, show the finite field elements as hex strings.

INPUT:

- `M` - an AES state array

EXAMPLES:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: k = sr.base_ring()
sage: A = matrix(k, 2, 2, [1, k.gen(), 0, k.gen()^2])
sage: sr.hex_str_matrix(A)
' 1 2 \n 0 4 \n'
```

hex_str_vector(*M*)

Return a one-dimensional AES-like representation of the matrix M.

That is, show the finite field elements as hex strings.

INPUT:

- `M` - an AES state array

EXAMPLES:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: k = sr.base_ring()
sage: A = matrix(k, 2, 2, [1, k.gen(), 0, k.gen()^2])
sage: sr.hex_str_vector(A)
'1024'
```

is_state_array(*d*)

Return `True` if `d` is a state array, i.e. has the correct dimensions and base field.

EXAMPLES:

```
sage: sr = mq.SR(2, 2, 4, 8)
sage: k = sr.base_ring()
sage: sr.is_state_array( matrix(k, 2, 4) )
True
```

```
sage: sr = mq.SR(2, 2, 4, 8)
sage: k = sr.base_ring()
sage: sr.is_state_array( matrix(k, 4, 4) )
False
```

key_schedule(*kj*, *i*)

Return k_i for a given i and k_j with $j = i - 1$.

EXAMPLES:

```
sage: sr = mq.SR(10, 4, 4, 8, star=True, allow_zero_inversions=True)
sage: ki = sr.state_array()
sage: for i in range(10):
....:     ki = sr.key_schedule(ki, i+1)
sage: print(sr.hex_str_matrix(ki))
B4 3E 23 6F
EF 92 E9 8F
5B E2 51 18
CB 11 CF 8E
```

key_schedule_polynomials(*i*)

Return polynomials for the i-th round of the key schedule.

INPUT:

- `i` - round ($0 \leq i \leq n$)

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=False)
```

The 0-th subkey is the user provided key, so only conjugacy relations or field polynomials are added.:

```
sage: sr.key_schedule_polynomials(0)
(k000^2 + k000, k001^2 + k001, k002^2 + k002, k003^2 + k003)
```

The 1-th subkey is derived from the user provided key according to the key schedule which is non-linear.:

```
sage: sr.key_schedule_polynomials(1)
(k100 + s000 + s002 + s003,
 k101 + s000 + s001 + s003 + 1,
 k102 + s000 + s001 + s002 + 1,
 k103 + s001 + s002 + s003 + 1,
 k100^2 + k100, k101^2 + k101, k102^2 + k102, k103^2 + k103,
 s000^2 + s000, s001^2 + s001, s002^2 + s002, s003^2 + s003,
 s000*k000 + s000*k003 + s001*k002 + s002*k001 + s003*k000,
 s000*k000 + s000*k001 + s001*k000 + s001*k003 + s002*k002 + s003*k001,
 s000*k001 + s000*k002 + s001*k000 + s001*k001 + s002*k000 + s002*k003 +␣
↪s003*k002,
 s000*k000 + s000*k001 + s000*k003 + s001*k001 + s002*k000 + s002*k002 +␣
↪s003*k000 + k000,
 s000*k002 + s001*k000 + s001*k001 + s001*k003 + s002*k001 + s003*k000 +␣
↪s003*k002 + k001,
 s000*k000 + s000*k001 + s000*k002 + s001*k002 + s002*k000 + s002*k001 +␣
↪s002*k003 + s003*k001 + k002,
 s000*k001 + s001*k000 + s001*k002 + s002*k000 + s003*k001 + s003*k003 + k003,
 s000*k000 + s000*k002 + s000*k003 + s001*k000 + s001*k001 + s002*k002 +␣
↪s003*k000 + s000,
 s000*k001 + s000*k003 + s001*k001 + s001*k002 + s002*k000 + s002*k003 +␣
↪s003*k001 + s001,
 s000*k000 + s000*k002 + s001*k000 + s001*k002 + s001*k003 + s002*k000 +␣
↪s002*k001 + s003*k002 + s002,
 s000*k001 + s000*k002 + s001*k000 + s001*k003 + s002*k001 + s003*k003 + s003,
 s000*k002 + s001*k001 + s002*k000 + s003*k003 + 1)
```

mix_columns(*d*)

Perform the `MixColumns` operation on `d`.

INPUT:

- `d` - state array or something coercible to a state array

EXAMPLES:

```
sage: sr = mq.SR(10, 4, 4, 4)
sage: E = sr.state_array() + 1; E
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
sage: sr.mix_columns(E)
[    a a + 1     1     1]
[    1     a a + 1     1]
[    1     1     a a + 1]
[a + 1     1     1     a]
```

new_generator(***kwds*)

Return a new `SR` instance equal to this instance except for the parameters passed explicitly to this function.

INPUT:

- `**kwds` - see the `SR` constructor for accepted parameters

EXAMPLES:

```
sage: sr = mq.SR(2,1,1,4); sr
SR(2,1,1,4)
sage: sr.ring().base_ring()
Finite Field in a of size 2^4
```

```
sage: sr2 = sr.new_generator(gf2=True); sr2
SR(2,1,1,4)
sage: sr2.ring().base_ring()
Finite Field of size 2
sage: sr3 = sr2.new_generator(correct_only=True)
sage: len(sr2.inversion_polynomials_single_sbox())
20
sage: len(sr3.inversion_polynomials_single_sbox())
19
```

polynomial_system(*P=None*, *K=None*, *C=None*)

Return a polynomial system for this small scale AES variant for a given plaintext-key pair.

If neither `P`, `K` nor `C` are provided, a random pair (`P`, `K`) will be generated. If `P` and `C` are provided no `K` needs to be provided.

INPUT:

- `P` - vector, list, or tuple (default: `None`)
- `K` - vector, list, or tuple (default: `None`)
- `C` - vector, list, or tuple (default: `None`)

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: P = sr.vector([0, 0, 1, 0])
sage: K = sr.vector([1, 0, 0, 1])
sage: F, s = sr.polynomial_system(P, K)
```

This returns a polynomial system:

```
sage: F
Polynomial Sequence with 36 Polynomials in 20 Variables
```

and a solution:

```
sage: s # random -- maybe we need a better doctest here?
{k000: 1, k001: 0, k003: 1, k002: 0}
```

This solution is not the only solution that we can learn from the Groebner basis of the system.

```
sage: F.groebner_basis()[-3:]
[k000 + 1, k001, k003 + 1]
```

In particular we have two solutions:

```
sage: len(F.ideal().variety())
2
```

In the following example we provide `C` explicitly:

```
sage: C = sr(P,K)
sage: F,s = sr.polynomial_system(P=P, C=C)
sage: F
Polynomial Sequence with 36 Polynomials in 20 Variables
```

Alternatively, we can use symbols for the `P` and `C`. First, we have to create a polynomial ring:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: R = sr.R
sage: vn = sr.varstrs("P",0,1,4) + R.variable_names() + sr.varstrs("C",0,1,4)
sage: R = BooleanPolynomialRing(len(vn),vn)
sage: sr.R = R
```

Now, we can construct the purely symbolic equation system:

```
sage: C = sr.vars("C",0); C
(C000, C001, C002, C003)
sage: P = sr.vars("P",0)
sage: F,s = sr.polynomial_system(P=P,C=C)
sage: [(k,v) for k,v in sorted(s.items())] # this can be ignored
[(k003, 1), (k002, 1), (k001, 0), (k000, 1)]
sage: F
Polynomial Sequence with 36 Polynomials in 28 Variables
sage: F.part(0)
(P000 + w100 + k000, P001 + w101 + k001, P002 + w102 + k002, P003 + w103 +␣
↪k003)
sage: F.part(-2)
(k100 + x100 + x102 + x103 + C000, k101 + x100 + x101 + x103 + C001 + 1, ...)
```

We show that the (returned) key is a solution to the returned system:

```
sage: sr = mq.SR(3,4,4,8, star=True, gf2=True, polybori=True)
sage: F,s = sr.polynomial_system()
sage: F.subs(s).groebner_basis() # long time
Polynomial Sequence with 1248 Polynomials in 1248 Variables
```

random_element(*elem_type='vector'*, **args*, ***kwds*)

Return a random element for self. Other arguments and keywords are passed to random_* methods.

INPUT:

- `elem_type` - either 'vector' or 'state array' (default: `'vector'`)

EXAMPLES:

```
sage: sr = mq.SR()
sage: sr.random_element()
[    a^2]
[  a + 1]
[a^2 + 1]
[      a]
sage: sr.random_element('state_array')
[a^3 + a + 1]
```

Passes extra positional or keyword arguments through:

```
sage: sr.random_element(density=0)
[0]
[0]
[0]
[0]
```

random_state_array(**args*, ***kwds*)

Return a random element in `MatrixSpace(self.base_ring(), self.r, self.c)`.

EXAMPLES:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: sr.random_state_array()
[              a^2       a^3 + a + 1]
[a^3 + a^2 + a + 1             a + 1]
```

random_vector(**args*, ***kwds*)

Return a random vector as it might appear in the algebraic expression of self.

EXAMPLES:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: sr.random_vector()
[              a^2]
[            a + 1]
[          a^2 + 1]
[                a]
[a^3 + a^2 + a + 1]
[          a^3 + a]
[              a^3]
[        a^3 + a^2]
[      a^3 + a + 1]
[          a^3 + 1]
[    a^3 + a^2 + 1]
```

(continues on next page)

(continued from previous page)

```
[    a^3 + a^2 + a]
[           a + 1]
[         a^2 + 1]
[             a]
[            a^2]
```

Note: ϕ was already applied to the result.

ring(*order=None*, *reverse_variables=None*)

Construct a ring as a base ring for the polynomial system.

By default, variables are ordered in the reverse of their natural ordering, i.e. the reverse of as they appear.

INPUT:

- `order` - a monomial ordering (default: `None`)
- `reverse_variables` - reverse rounds of variables (default: `True`)

The variable assignment is as follows:

- $k_{i,j,l}$ - subkey round i word j conjugate/bit l
- $s_{i,j,l}$ - subkey inverse round i word j conjugate/bit l
- $w_{i,j,l}$ - inversion input round i word j conjugate/bit l
- $x_{i,j,l}$ - inversion output round i word j conjugate/bit l

Note that the variables are ordered in column major ordering in the state array and that the bits are ordered in little endian ordering.

For example, if $x_{0,1,0}$ is a variable over $\mathbf{F}_2$ for $r = 2$ and $c = 2$ then refers to the *most* significant bit of the entry in the position (1,0) in the state array matrix.

EXAMPLES:

```
sage: sr = mq.SR(2, 1, 1, 4)
sage: P = sr.ring(order='block')
sage: print(P.repr_long())
Polynomial Ring
  Base Ring : Finite Field in a of size 2^4
       Size : 36 Variables
   Block  0 : Ordering : deglex
              Names    : k200, k201, k202, k203, x200, x201, x202, x203, w200,
↪ w201, w202, w203, s100, s101, s102, s103
   Block  1 : Ordering : deglex
              Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100,
↪ w101, w102, w103, s000, s001, s002, s003
   Block  2 : Ordering : deglex
              Names    : k000, k001, k002, k003
```

round_polynomials(*i*, *plaintext=None*, *ciphertext=None*)

Return list of polynomials for a given round i.

If `i == 0` a plaintext must be provided, if `i == n` a ciphertext must be provided.

INPUT:

- `i` - round number

- `plaintext` - optional plaintext (mandatory in first round)
- `ciphertext` - optional ciphertext (mandatory in last round)

OUTPUT: tuple

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 4)
sage: k = sr.base_ring()
sage: p = [k.random_element() for _ in range(sr.r*sr.c)]
sage: sr.round_polynomials(0, plaintext=p)
(w100 + k000 + (a^2 + 1), w101 + k001 + (a), w102 + k002 + (a^2), w103 + k003␣
↪+ (a + 1))
```

sbox(*inversion_only=False*)

Return an S-Box object for this SR instance.

INPUT:

- `inversion_only` - do not include the $\mathbf{F}_2$ affine map when computing the S-Box (default: `False`)

EXAMPLES:

```
sage: sr = mq.SR(1,2,2,4, allow_zero_inversions=True)
sage: S = sr.sbox(); S
(6, 11, 5, 4, 2, 14, 7, 10, 9, 13, 15, 12, 3, 1, 0, 8)

sage: sr.sub_byte(0)
a^2 + a
sage: sage_eval(str(sr.sub_byte(0)), {'a':2})
6
sage: S(0)
6

sage: sr.sub_byte(1)
a^3 + a + 1
sage: sage_eval(str(sr.sub_byte(1)), {'a':2})
11
sage: S(1)
11

sage: sr = mq.SR(1,2,2,8, allow_zero_inversions=True)
sage: S = sr.sbox(); S
(99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43,
254, 215, 171, 118, 202, 130, 201, 125, 250, 89, 71, 240,
173, 212, 162, 175, 156, 164, 114, 192, 183, 253, 147, 38,
54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21, 4,
199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39,
178, 117, 9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214,
179, 41, 227, 47, 132, 83, 209, 0, 237, 32, 252, 177, 91,
106, 203, 190, 57, 74, 76, 88, 207, 208, 239, 170, 251,
67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168, 81,
163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33, 16,
255, 243, 210, 205, 12, 19, 236, 95, 151, 68, 23, 196,
167, 126, 61, 100, 93, 25, 115, 96, 129, 79, 220, 34, 42,
144, 136, 70, 238, 184, 20, 222, 94, 11, 219, 224, 50, 58,
10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121,
231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234,
101, 122, 174, 8, 186, 120, 37, 46, 28, 166, 180, 198,
```

(continues on next page)

(continued from previous page)

```
232, 221, 116, 31, 75, 189, 139, 138, 112, 62, 181, 102,
72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158, 225,
248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206,
85, 40, 223, 140, 161, 137, 13, 191, 230, 66, 104, 65,
153, 45, 15, 176, 84, 187, 22)

sage: sr.sub_byte(0)
a^6 + a^5 + a + 1

sage: sage_eval(str(sr.sub_byte(0)), {'a':2})
99
sage: S(0)
99

sage: sr.sub_byte(1)
a^6 + a^5 + a^4 + a^3 + a^2

sage: sage_eval(str(sr.sub_byte(1)), {'a':2})
124

sage: S(1)
124

sage: sr = mq.SR(1,2,2,4, allow_zero_inversions=True)
sage: S = sr.sbox(inversion_only=True); S
(0, 1, 9, 14, 13, 11, 7, 6, 15, 2, 12, 5, 10, 4, 3, 8)

sage: S(0)
0
sage: S(1)
1

sage: S(sr.k.gen())
a^3 + 1
```

sbox_constant()

Return the S-Box constant which is added after $L(x^{-1})$ was performed. That is `0x63` if `e == 8` or `0x6` if `e == 4`.

EXAMPLES:

```
sage: sr = mq.SR(10, 1, 1, 8)
sage: sr.sbox_constant()
a^6 + a^5 + a + 1
```

shift_rows(d)

Perform the `ShiftRows` operation on `d`.

INPUT:

- `d` - state array or something coercible to a state array

EXAMPLES:

```
sage: sr = mq.SR(10, 4, 4, 4)
sage: E = sr.state_array() + 1; E
[1 0 0 0]
[0 1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0]
[0 0 0 1]
```

```
sage: sr.shift_rows(E)
[1 0 0 0]
[1 0 0 0]
[1 0 0 0]
[1 0 0 0]
```

state_array(*d=None*)

Convert the parameter to a state array.

INPUT:

- `d` - a matrix, a list, or a tuple (default: `None`)

EXAMPLES:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: k = sr.base_ring()
sage: e1 = [k.fetch_int(e) for e in range(2*2)]; e1
[0, 1, a, a + 1]
sage: e2 = sr.phi( Matrix(k, 2*2, 1, e1) )
sage: sr.state_array(e1) # note the column major ordering
[    0     a]
[    1 a + 1]
sage: sr.state_array(e2)
[    0     a]
[    1 a + 1]
```

```
sage: sr.state_array()
[0 0]
[0 0]
```

sub_byte(*b*)

Perform `SubByte` on a single byte/halfbyte `b`.

A `ZeroDivision` exception is raised if an attempt is made to perform an inversion on the zero element. This can be disabled by passing `allow_zero_inversion=True` to the constructor. A zero inversion can result in an inconsistent equation system.

INPUT:

- `b` - an element in `self.base_ring()`

EXAMPLES:

The S-Box table for $\mathbf{F}_{2^4}$:

```
sage: sr = mq.SR(1, 1, 1, 4, allow_zero_inversions=True)
sage: for e in sr.base_ring():
....:     print('% 20s % 20s'%(e, sr.sub_byte(e)))
                   0                a^2 + a
                   a                a^2 + 1
                 a^2                    a
                 a^3                a^3 + 1
               a + 1                    a^2
            a^2 + a            a^2 + a + 1
```

(continues on next page)

(continued from previous page)

```
        a^3 + a^2                 a + 1
      a^3 + a + 1             a^3 + a^2
          a^2 + 1         a^3 + a^2 + a
          a^3 + a     a^3 + a^2 + a + 1
      a^2 + a + 1               a^3 + a
    a^3 + a^2 + a                   0
a^3 + a^2 + a + 1                 a^3
    a^3 + a^2 + 1                   1
          a^3 + 1         a^3 + a^2 + 1
                1           a^3 + a + 1
```

sub_bytes(*d*)

Perform the non-linear transform on `d`.

INPUT:

- `d` - state array or something coercible to a state array

EXAMPLES:

```
sage: sr = mq.SR(2, 1, 2, 8, gf2=True)
sage: k = sr.base_ring()
sage: A = Matrix(k, 1, 2 , [k(1), k.gen()])
sage: sr.sub_bytes(A)
[  a^6 + a^5 + a^4 + a^3 + a^2 a^6 + a^5 + a^4 + a^2 + a + 1]
```

varformatstr(*name*, *n=None*, *rc=None*, *e=None*)

Return a format string which is understood by print et al.

If a numerical value is omitted, the default value of `self` is used. The numerical values (`n`, `rc`, `e`) are used to determine the width of the respective fields in the format string.

INPUT:

- `name` - name of the variable
- `n` - number of rounds (default: `None`)
- `rc` - number of rows * number of cols (default: `None`)
- `e` - exponent of base field (default: `None`)

EXAMPLES:

```
sage: sr = mq.SR(1, 2, 2, 4)
sage: sr.varformatstr('x')
'x%01d%01d%01d'
sage: sr.varformatstr('x', n=1000)
'x%03d%03d%03d'
```

variable_dict()

Return a dictionary to access variables in `self.R` by their names.

EXAMPLES:

```
sage: sr = mq.SR(1,1,1,4)
sage: sr.variable_dict()
{'k000': k000,
 'k001': k001,
 'k002': k002,
```

(continues on next page)

(continued from previous page)

```
 'k003': k003,
 'k100': k100,
 'k101': k101,
 'k102': k102,
 'k103': k103,
 's000': s000,
 's001': s001,
 's002': s002,
 's003': s003,
 'w100': w100,
 'w101': w101,
 'w102': w102,
 'w103': w103,
 'x100': x100,
 'x101': x101,
 'x102': x102,
 'x103': x103}

sage: sr = mq.SR(1,1,1,4,gf2=True)
sage: sr.variable_dict()
{'k000': k000,
 'k001': k001,
 'k002': k002,
 'k003': k003,
 'k100': k100,
 'k101': k101,
 'k102': k102,
 'k103': k103,
 's000': s000,
 's001': s001,
 's002': s002,
 's003': s003,
 'w100': w100,
 'w101': w101,
 'w102': w102,
 'w103': w103,
 'x100': x100,
 'x101': x101,
 'x102': x102,
 'x103': x103}
```

vars(*name*, *nr*, *rc=None*, *e=None*)

Return a list of variables in `self`.

INPUT:

- `name` - variable name
- `nr` - number of round to create variable strings for
- `rc` - number of rounds * number of columns in the state array (default: `None`)
- `e` - exponent of base field (default: `None`)

EXAMPLES:

```
sage: sr = mq.SR(10, 1, 2, 4)
sage: sr.vars('x', 2)
(x200, x201, x202, x203, x210, x211, x212, x213)
```

varstr(*name*, *nr*, *rc*, *e*)

Return a string representing a variable for the small scale AES subject to the given constraints.

INPUT:

- `name` - variable name
- `nr` - number of round to create variable strings for
- `rc` - row*column index in state array
- `e` - exponent of base field

EXAMPLES:

```
sage: sr = mq.SR(10, 1, 2, 4)
sage: sr.varstr('x', 2, 1, 1)
'x211'
```

varstrs(*name*, *nr*, *rc=None*, *e=None*)

Return a list of strings representing variables in `self`.

INPUT:

- `name` - variable name
- `nr` - number of round to create variable strings for
- `rc` - number of rows * number of columns in the state array (default: `None`)
- `e` - exponent of base field (default: `None`)

EXAMPLES:

```
sage: sr = mq.SR(10, 1, 2, 4)
sage: sr.varstrs('x', 2)
('x200', 'x201', 'x202', 'x203', 'x210', 'x211', 'x212', 'x213')
```

class sage.crypto.mq.sr.**SR_gf2**(*n=1*, *r=1*, *c=1*, *e=4*, *star=False*, ***kwargs*)

Bases: *sage.crypto.mq.sr.SR_generic*

Small Scale Variants of the AES polynomial system constructor over $\mathbf{F}_2$. See help for SR.

EXAMPLES:

```
sage: sr = mq.SR(gf2=True)
sage: sr
SR(1,1,1,4)
```

antiphi(*l*)

The operation ϕ^{-1} from [MR2002] or the inverse of `self.phi`.

INPUT:

- `l` - a vector in the sense of `self.is_vector`

EXAMPLES:

```
sage: sr = mq.SR(gf2=True)
sage: A = sr.random_state_array()
sage: A
[a^2]
sage: sr.antiphi(sr.phi(A)) == A
True
```

field_polynomials(*name*, *i*, *l=None*)

Return list of field polynomials for a given round `i` and name `name`.

INPUT:

- `name` - variable name
- `i` - round number
- `l` - length of variable list (default: `None` = r*c)

EXAMPLES:

```
sage: sr = mq.SR(3, 1, 1, 8, gf2=True, polybori=False)
sage: sr.field_polynomials('x', 2)
[x200^2 + x200, x201^2 + x201,
 x202^2 + x202, x203^2 + x203,
 x204^2 + x204, x205^2 + x205,
 x206^2 + x206, x207^2 + x207]
```

```
sage: sr = mq.SR(3, 1, 1, 8, gf2=True, polybori=True)
sage: sr.field_polynomials('x', 2)
[]
```

inversion_polynomials(*xi*, *wi*, *length*)

Return polynomials to represent the inversion in the AES S-Box.

INPUT:

- `xi` - output variables
- `wi` - input variables
- `length` - length of both lists

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 8, gf2=True)
sage: xi = sr.vars('x', 1)
sage: wi = sr.vars('w', 1)
sage: sr.inversion_polynomials(xi, wi, len(xi))[:3]
[x100*w100 + x100*w102 + x100*w103 + x100*w107 + x101*w101 + x101*w102 +␣
↪x101*w106 + x102*w100 + x102*w101 + x102*w105 + x103*w100 + x103*w104 +␣
↪x104*w103 + x105*w102 + x106*w101 + x107*w100,
 x100*w101 + x100*w103 + x100*w104 + x101*w100 + x101*w102 + x101*w103 +␣
↪x101*w107 + x102*w101 + x102*w102 + x102*w106 + x103*w100 + x103*w101 +␣
↪x103*w105 + x104*w100 + x104*w104 + x105*w103 + x106*w102 + x107*w101,
 x100*w102 + x100*w104 + x100*w105 + x101*w101 + x101*w103 + x101*w104 +␣
↪x102*w100 + x102*w102 + x102*w103 + x102*w107 + x103*w101 + x103*w102 +␣
↪x103*w106 + x104*w100 + x104*w101 + x104*w105 + x105*w100 + x105*w104 +␣
↪x106*w103 + x107*w102]
```

inversion_polynomials_single_sbox(*x=None*, *w=None*, *biaffine_only=None*, *correct_only=None*)

Return inversion polynomials of a single S-Box.

INPUT:

- `xi` - output variables
- `wi` - input variables
- `length` - length of both lists

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 8, gf2=True)
sage: len(sr.inversion_polynomials_single_sbox())
24
sage: len(sr.inversion_polynomials_single_sbox(correct_only=True))
23
sage: len(sr.inversion_polynomials_single_sbox(biaffine_only=False))
40
sage: len(sr.inversion_polynomials_single_sbox(biaffine_only=False, correct_
↪only=True))
39


sage: sr = mq.SR(1, 1, 1, 8, gf2=True)
sage: l0 = sr.inversion_polynomials_single_sbox(); len(l0)
24
sage: l1 = sr.inversion_polynomials_single_sbox(correct_only=True); len(l1)
23
sage: l2 = sr.inversion_polynomials_single_sbox(biaffine_only=False); len(l2)
40
sage: l3 = sr.inversion_polynomials_single_sbox(biaffine_only=False, correct_
↪only=True); len(l3)
39

sage: set(l0) == set(sr._inversion_polynomials_single_sbox())
True
sage: set(l1) == set(sr._inversion_polynomials_single_sbox(correct_only=True))
True
sage: set(l2) == set(sr._inversion_polynomials_single_sbox(biaffine_
↪only=False))
True
sage: set(l3) == set(sr._inversion_polynomials_single_sbox(biaffine_
↪only=False, correct_only=True))
True

sage: sr = mq.SR(1, 1, 1, 4, gf2=True)
sage: l0 = sr.inversion_polynomials_single_sbox(); len(l0)
12
sage: l1 = sr.inversion_polynomials_single_sbox(correct_only=True); len(l1)
11
sage: l2 = sr.inversion_polynomials_single_sbox(biaffine_only=False); len(l2)
20
sage: l3 = sr.inversion_polynomials_single_sbox(biaffine_only=False, correct_
↪only=True); len(l3)
19

sage: set(l0) == set(sr._inversion_polynomials_single_sbox())
True
sage: set(l1) == set(sr._inversion_polynomials_single_sbox(correct_only=True))
True
sage: set(l2) == set(sr._inversion_polynomials_single_sbox(biaffine_
↪only=False))
True
sage: set(l3) == set(sr._inversion_polynomials_single_sbox(biaffine_
↪only=False, correct_only=True))
True
```

is_vector(*d*)

Return `True` if the given matrix satisfies the conditions for a vector as it appears in the algebraic expression of `self`.

INPUT:

- `d` - matrix

EXAMPLES:

```
sage: sr = mq.SR(gf2=True)
sage: sr
SR(1,1,1,4)
sage: k = sr.base_ring()
sage: A = Matrix(k, 1, 1, [k.gen()])
sage: B = sr.vector(A)
sage: sr.is_vector(A)
False
sage: sr.is_vector(B)
True
```

lin_matrix(*length=None*)

Return the `Lin` matrix.

If no `length` is provided, the standard state space size is used. The key schedule calls this method with an explicit length argument because only `self.r` S-Box applications are performed in the key schedule.

INPUT:

- `length` - length of state space (default: `None`)

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True)
sage: sr.lin_matrix()
[1 0 1 1]
[1 1 0 1]
[1 1 1 0]
[0 1 1 1]
```

mix_columns_matrix()

Return the `MixColumns` matrix.

EXAMPLES:

```
sage: sr = mq.SR(1, 2, 2, 4, gf2=True)
sage: s = sr.random_state_array()
sage: r1 = sr.mix_columns(s)
sage: r2 = sr.state_array(sr.mix_columns_matrix() * sr.vector(s))
sage: r1 == r2
True
```

phi(*l*, *diffusion_matrix=False*)

The operation ϕ from [MR2002]

Given a list/matrix of elements in $\mathbf{F}_{2^e}$, return a matching list/matrix of elements in $\mathbf{F}_2$.

INPUT:

- `l` - element to perform ϕ on.
- `diffusion_matrix` - if `True`, the given matrix `l` is transformed to a matrix which performs the same operation over $\mathbf{F}_2$ as `l` over $\mathbf{F}_{2^n}$ (default: `False`).

EXAMPLES:

```
sage: sr = mq.SR(2, 1, 2, 4, gf2=True)
sage: k = sr.base_ring()
sage: A = matrix(k, 1, 2, [k.gen(), 0] )
sage: sr.phi(A)
[0 0]
[0 0]
[1 0]
[0 0]
```

shift_rows_matrix()

Return the ShiftRows matrix.

EXAMPLES:

```
sage: sr = mq.SR(1, 2, 2, 4, gf2=True)
sage: s = sr.random_state_array()
sage: r1 = sr.shift_rows(s)
sage: r2 = sr.state_array( sr.shift_rows_matrix() * sr.vector(s) )
sage: r1 == r2
True
```

vector(*d=None*)

Constructs a vector suitable for the algebraic representation of SR.

INPUT:

- d - values for vector (default: None)

EXAMPLES:

```
sage: sr = mq.SR(gf2=True)
sage: sr
SR(1,1,1,4)
sage: k = sr.base_ring()
sage: A = Matrix(k, 1, 1, [k.gen()])
sage: sr.vector(A)
[0]
[0]
[1]
[0]
```

class sage.crypto.mq.sr.**SR_gf2_2**(*n=1*, *r=1*, *c=1*, *e=4*, *star=False*, ***kwargs*)

Bases: *sage.crypto.mq.sr.SR_gf2*

This is an example how to customize the SR constructor.

In this example, we replace the S-Box inversion polynomials by the polynomials generated by the S-Box class.

inversion_polynomials_single_sbox(*x=None*, *w=None*, *biaffine_only=None*, *correct_only=None*, *groebner=False*)

Return inversion polynomials of a single S-Box.

INPUT:

- x - output variables (default: None)
- w - input variables (default: None)
- biaffine_only - ignored (always False)
- correct_only - ignored (always True)

- `groebner` - precompute the Groebner basis for this S-Box (default: `False`).

EXAMPLES:

```
sage: from sage.crypto.mq.sr import SR_gf2_2
sage: e = 4
sage: sr = SR_gf2_2(1, 1, 1, e)
sage: P = PolynomialRing(GF(2),['x%d'%i for i in range(e)] + ['w%d'%i for i␣
↪in range(e)],order='lex')
sage: X,W = P.gens()[:e],P.gens()[e:]
sage: sr.inversion_polynomials_single_sbox(X, W, groebner=True)
[x0 + w0*w1*w2 + w0*w1 + w0*w2 + w0*w3 + w0 + w1 + w2,
 x1 + w0*w1*w3 + w0*w3 + w0 + w1*w3 + w1 + w2*w3,
 x2 + w0*w2*w3 + w0*w2 + w0 + w1*w2 + w1*w3 + w2*w3,
 x3 + w0*w1*w2 + w0 + w1*w2*w3 + w1*w2 + w1*w3 + w1 + w2 + w3]

sage: from sage.crypto.mq.sr import SR_gf2_2
sage: e = 4
sage: sr = SR_gf2_2(1, 1, 1, e)
sage: sr.inversion_polynomials_single_sbox()
[w3*w1 + w3*w0 + w3*x2 + w3*x1 + w3 + w2*w1 + w1 + x3 + x2 + x1,
 w3*w2 + w3*w1 + w3*x3 + w2 + w1 + x3,
 w3*w2 + w3*w1 + w3*x2 + w3 + w2*x3 + x2 + x1,
 w3*w2 + w3*w1 + w3*x3 + w3*x2 + w3*x1 + w3 + w2*x2 + w0 + x3 + x2 + x1 + x0,
 w3*w2 + w3*w1 + w3*x1 + w3*x0 + w2*x1 + w0 + x3 + x0,
 w3*w2 + w3*w1 + w3*w0 + w3*x2 + w3*x1 + w2*w0 + w2*x0 + w0 + x3 + x2 + x1 +␣
↪x0,
 w3*w2 + w3*x1 + w3 + w2*w0 + w1*w0 + w1 + x3 + x2,
 w3*w2 + w3*w1 + w3*x1 + w1*x3 + x3 + x2 + x1,
 w3*x3 + w3*x2 + w3*x0 + w3 + w1*x2 + w1 + w0 + x2 + x0,
 w3*w2 + w3*w1 + w3*x2 + w3*x1 + w1*x1 + w1 + w0 + x2 + x0,
 w3*w2 + w3*w1 + w3*w0 + w3*x3 + w3*x1 + w2*w0 + w1*x0 + x3 + x2,
 w3*w2 + w3*w1 + w3*x2 + w3*x1 + w3*x0 + w3 + w1 + w0*x3 + x3 + x2,
 w3*w2 + w3*w1 + w3*w0 + w3*x3 + w3 + w2*w0 + w1 + w0*x2 + x3 + x2,
 w3*w0 + w3*x2 + w2*w0 + w0*x1 + w0 + x3 + x1 + x0,
 w3*w0 + w3*x3 + w3*x0 + w2*w0 + w1 + w0*x0 + w0 + x3 + x2,
 w3*w2 + w3 + w1 + x3*x2 + x3 + x1,
 w3*w2 + w3*x3 + w1 + x3*x1 + x3 + x2,
 w3*w2 + w3*w0 + w3*x3 + w3*x2 + w3*x1 + w0 + x3*x0 + x1 + x0,
 w3*w2 + w3*w1 + w3*w0 + w3*x3 + w1 + w0 + x2*x1 + x2 + x0,
 w3*w2 + w2*w0 + w1 + x3 + x2*x0,
 w3*x3 + w3*x1 + w2*w0 + w1 + x3 + x2 + x1*x0 + x1]
```

class sage.crypto.mq.sr.**SR_gf2n**(*n=1*, *r=1*, *c=1*, *e=4*, *star=False*, ***kwargs*)

Bases: *sage.crypto.mq.sr.SR_generic*

Small Scale Variants of the AES polynomial system constructor over $\mathbf{F}_{2^n}$.

antiphi(*l*)

The operation ϕ^{-1} from [MR2002] or the inverse of `self.phi`.

INPUT:

- `l` – a vector in the sense of *is_vector()*

EXAMPLES:

```
sage: sr = mq.SR()
sage: A = sr.random_state_array()
sage: A
```

(continues on next page)

(continued from previous page)

```
[a^2]
sage: sr.antiphi(sr.phi(A)) == A
True
```

field_polynomials(*name*, *i*, *l=None*)

Return list of conjugacy polynomials for a given round `i` and name `name`.

INPUT:

- `name` - variable name
- `i` - round number
- `l` - r*c (default: `None`)

EXAMPLES:

```
sage: sr = mq.SR(3, 1, 1, 8)
sage: sr.field_polynomials('x', 2)
[x200^2 + x201,
x201^2 + x202,
x202^2 + x203,
x203^2 + x204,
x204^2 + x205,
x205^2 + x206,
x206^2 + x207,
x207^2 + x200]
```

inversion_polynomials(*xi*, *wi*, *length*)

Return polynomials to represent the inversion in the AES S-Box.

INPUT:

- `xi` - output variables
- `wi` - input variables
- `length` - length of both lists

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 8)
sage: R = sr.ring()
sage: xi = Matrix(R, 8, 1, sr.vars('x', 1))
sage: wi = Matrix(R, 8, 1, sr.vars('w', 1))
sage: sr.inversion_polynomials(xi, wi, 8)
[x100*w100 + 1,
x101*w101 + 1,
x102*w102 + 1,
x103*w103 + 1,
x104*w104 + 1,
x105*w105 + 1,
x106*w106 + 1,
x107*w107 + 1]
```

is_vector(*d*)

Return `True` if `d` can be used as a vector for `self`.

EXAMPLES:

```
sage: sr = mq.SR()
sage: sr
SR(1,1,1,4)
sage: k = sr.base_ring()
sage: A = Matrix(k, 1, 1, [k.gen()])
sage: B = sr.vector(A)
sage: sr.is_vector(A)
False
sage: sr.is_vector(B)
True
```

lin_matrix(*length=None*)

Return the `Lin` matrix.

If no `length` is provided, the standard state space size is used. The key schedule calls this method with an explicit length argument because only `self.r` S-Box applications are performed in the key schedule.

INPUT:

- `length` - length of state space (default: `None`)

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 4)
sage: sr.lin_matrix()
[          a^2 + 1                 1          a^3 + a^2            a^2 + 1]
[               a                  a                1 a^3 + a^2 + a + 1]
[          a^3 + a               a^2               a^2                  1]
[               1               a^3             a + 1              a + 1]
```

mix_columns_matrix()

Return the `MixColumns` matrix.

EXAMPLES:

```
sage: sr = mq.SR(1, 2, 2, 4)
sage: s = sr.random_state_array()
sage: r1 = sr.mix_columns(s)
sage: r2 = sr.state_array(sr.mix_columns_matrix() * sr.vector(s))
sage: r1 == r2
True
```

phi(*l*)

The operation ϕ from [MR2002]

Projects state arrays to their algebraic representation.

INPUT:

- `l` - element to perform ϕ on.

EXAMPLES:

```
sage: sr = mq.SR(2, 1, 2, 4)
sage: k = sr.base_ring()
sage: A = matrix(k, 1, 2, [k.gen(), 0] )
sage: sr.phi(A)
[      a       0]
[    a^2       0]
[  a + 1       0]
[a^2 + 1       0]
```

shift_rows_matrix()
Return the ShiftRows matrix.

EXAMPLES:

```
sage: sr = mq.SR(1, 2, 2, 4)
sage: s = sr.random_state_array()
sage: r1 = sr.shift_rows(s)
sage: r2 = sr.state_array( sr.shift_rows_matrix() * sr.vector(s) )
sage: r1 == r2
True
```

vector(*d=None*)
Constructs a vector suitable for the algebraic representation of SR, i.e. BES.

INPUT:

- d - values for vector, must be understood by self.phi (default:None)

EXAMPLES:

```
sage: sr = mq.SR()
sage: sr
SR(1,1,1,4)
sage: k = sr.base_ring()
sage: A = Matrix(k, 1, 1, [k.gen()])
sage: sr.vector(A)
[      a]
[    a^2]
[  a + 1]
[a^2 + 1]
```

sage.crypto.mq.sr.**test_consistency**(*max_n=2*, ***kwargs*)
Test all combinations of r, c, e and n in (1, 2) for consistency of random encryptions and their polynomial systems. $\mathbf{F}_2$ and $\mathbf{F}_{2^e}$ systems are tested. This test takes a while.

INPUT:

- max_n – maximal number of rounds to consider (default: 2)
- kwargs – are passed to the SR constructor

CHAPTER

SIXTEEN

RIJNDAEL-GF

Rijndael-GF is an algebraic implementation of the AES cipher which seeks to provide a fully generalized algebraic representation of both the whole AES cipher as well as its individual components.

This class is an algebraic implementation of the Rijndael-GF extension of the AES cipher, as described in [DR2002]. The AES cipher itself is defined to operate on a state in $(\mathbf{F}_2)^{8n_t}$ where $n_t \in \{16, 20, 24, 28, 32\}$. Rijndael-GF is a generalization of AES which allows for operations in $(\mathbf{F}_{2^8})^{n_t}$, enabling more algebraically sophisticated study of AES and its variants. This implementation of Rijndael-GF is suitable for learning purposes, for comparison to other algebraic ciphers, and for studying various techniques of algebraic cryptanalysis of AES. This cipher is different from *Mini-AES*, which is a teaching tool for beginners to understand the basic structure of AES.

An algebraic implementation of Rijndael-GF is achieved by recognizing that for each round component function ϕ of AES (SubBytes, ShiftRows, etc.) operating on state matrices, every entry of the output matrix $B = \phi(A)$ is representable as a polynomial with variables being the entries of the input state matrix A. Correspondingly, this implementation of Rijndael-GF provides a `RijndaelGF.Round_Component_Poly_Constr` class which allows for creation of these such polynomials. For each round component function ϕ of Rijndael-GF there exists a `Round_Component_Poly_Constr` object with a `__call__` method of the form `__call__(i, j)` which returns a polynomial representing $\phi(A)_{i,j}$ in terms of the entries of A. There additionally are various methods provided which allow for easy polynomial evaluation and for simple creation of `Round_Component_Poly_Constr` objects representing more complex aspects of the cipher.

This approach to implementing Rijndael-GF bears some similarity to the multivariate quadratic (MQ) systems utilized in *SR*, in that the MQ systems also seek to describe the AES cipher as a system of algebraic equations. Despite this initial similarity though, Rijndael-GF and *SR* are quite different as this implementation seeks to provide a fully generalized algebraic representation of both the whole AES cipher as well as its individual components, while *SR* is instead a family of parameterizable variants of the AES suitable as a framework for comparing different cryptanalytic techniques that can be brought to bear on the AES.

AUTHORS:

- Thomas Gagne (2015-06): initial version

EXAMPLES

We build Rijndael-GF with a block length of 4 and a key length of 6:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 6)
```

We can encrypt plaintexts and decrypt and ciphertexts by calling the `encrypt` and `decrypt` methods or by calling the Rijndael-GF object explicitly. Note that the default input format is a hex string.

```
sage: plaintext = '00112233445566778899aabbccddeeff'
sage: key = '000102030405060708090a0b0c0d0e0f1011121314151617'
sage: rgf.encrypt(plaintext, key)
'dda97ca4864cdfe06eaf70a0ec0d7191'
```

(continues on next page)

(continued from previous page)

```
sage: rgf.decrypt('dda97ca4864cdfe06eaf70a0ec0d7191', key)
'00112233445566778899aabbccddeeff'
```

We can also use binary strings as input and output.

```
sage: plain = '11101011100111110000000111001100' * 4
sage: key = '01100010111101101000110010111010' * 6
sage: ciphertext = rgf(plain, key, format='binary')
sage: ciphertext

↪'11010011000010011010110001000011101110110100110100110010011011111100011011100111111001110011101001
↪'
sage: rgf(ciphertext, key, algorithm='decrypt', format='binary') == plain
True
```

[DR2002] demonstrates an example of encryption which takes the plaintext '3243f6a8885a308d313198a2e0370734' and the key '2b7e151628aed2a6abf7158809cf4f3c' and returns the ciphertext '3902dc1925dc116a8409850b1dfb9732'. We can use this example to demonstrate the correctness of this implementation:

```
sage: rgf = RijndaelGF(4, 4) # change dimensions for this example
sage: plain = '3243f6a8885a308d313198a2e0370734'
sage: key = '2b7e151628aed2a6abf7158809cf4f3c'
sage: expected_ciphertext = '3925841d02dc09fbdc118597196a0b32'
sage: rgf.encrypt(plain, key) == expected_ciphertext
True
```

```
sage: rgf = RijndaelGF(4, 6) # revert to previous dimensions
```

To build polynomials representing entries of the output matrix $B = \phi(A)$ for any round component function ϕ, each of the round component functions (SubBytes, ShiftRows, and MixColumns) have a `Round_Component_Poly_Constr` object associated with it for building polynomials. These objects can be accessed by calling their getter functions: `rgf.sub_bytes_poly()`, `rgf.shift_rows_poly()`, and `rgf.mix_columns_poly()`. Each returned object has a `__call__` method which takes an index `i, j` and an `algorithm` flag ('encrypt' or 'decrypt') and returns a polynomial representing $\phi(A)_{i,j}$ in terms of the entries of A, where A is an arbitrary state matrix and ϕ is the round component function associated with that particular `Round_Component_Poly_Constr` object. Some of these objects' `__call__` methods also have additional keywords to modify their behavior, and so we describe the usage of each object below.

`rgf.shift_rows_poly()` and `rgf.mix_columns_poly()` do not have any additional keywords for their `__call__` methods and we can call them as such:

```
sage: sr_pc = rgf.shift_rows_poly_constr()
sage: sr_pc(1, 2)
a13
sage: sr_pc(2, 3, algorithm='decrypt')
a21
```

```
sage: mc_pc = rgf.mix_columns_poly_constr()
sage: mc_pc(1, 2)
a02 + (x)*a12 + (x + 1)*a22 + a32
sage: mc_pc(2, 3, algorithm='decrypt')
(x^3 + x^2 + 1)*a03 + (x^3 + 1)*a13 + (x^3 + x^2 + x)*a23 + (x^3 + x + 1)*a33
```

`rgf.sub_bytes_poly()` has a single keyword `no_inversion=False`, which when set to `True` returns only the affine transformation step of SubBytes. Below describes the usage of `rgf.sub_bytes_poly()`

```
sage: sb_pc = rgf.sub_bytes_poly_constr()
sage: sb_pc(1, 2)
(x^2 + 1)*a12^254 +
(x^3 + 1)*a12^253 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*a12^251 +
(x^5 + x^2 + 1)*a12^247 +
(x^7 + x^6 + x^5 + x^4 + x^2)*a12^239 +
a12^223 +
(x^7 + x^5 + x^4 + x^2 + 1)*a12^191 +
(x^7 + x^3 + x^2 + x + 1)*a12^127 +
(x^6 + x^5 + x + 1)
sage: sb_pc(2, 3, no_inversion=True)
(x^7 + x^3 + x^2 + x + 1)*a23^128 +
(x^7 + x^5 + x^4 + x^2 + 1)*a23^64 +
a23^32 +
(x^7 + x^6 + x^5 + x^4 + x^2)*a23^16 +
(x^5 + x^2 + 1)*a23^8 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*a23^4 +
(x^3 + 1)*a23^2 +
(x^2 + 1)*a23 +
(x^6 + x^5 + x + 1)
```

Because of the order of the affine transformation and the inversion step in SubBytes, calling `rgf.sub_bytes_poly()(i, j, algorithm='decrypt')` results in a polynomial with thousands of terms which takes a very long time to compute. Hence, when using the decryption version of `rgf.sub_bytes_poly()` with the intention of evaluating the polynomials it constructs, it is recommended to first call `rgf.sub_bytes_poly()(i, j, algorithm='decrypt', no_inversion=True)` to get a polynomial representing only the inverse affine transformation, evaluate this polynomial for a particular input block, then finally perform the inversion step after the affine transformation polynomial has been evaluated.

```
sage: inv_affine = sb_pc(1, 2, algorithm='decrypt',
....: no_inversion=True)
sage: state = rgf._hex_to_GF('ff87968431d86a51645151fa773ad009')
sage: evaluated = inv_affine(state.list())
sage: result = evaluated * -1
sage: rgf._GF_to_hex(result)
'79'
```

We can see how the variables of these polynomials are organized in A:

```
sage: rgf.state_vrs
[a00 a01 a02 a03]
[a10 a11 a12 a13]
[a20 a21 a22 a23]
[a30 a31 a32 a33]
```

The final `Round_Component_Poly_Constr` object we have not discussed yet is `add_round_key_poly`, which corresponds to the AddRoundKey round component function. This object differs from the other `Round_Component_Poly_Constr` objects in that it returns polynomials with variables being entries of an input state A as well as entries of various subkeys. Since there are N_r subkeys to choose from, `add_round_key_poly` has a keyword of `round=0` to select which subkey to use variables from.

```
sage: ark_pc = rgf.add_round_key_poly_constr()
sage: ark_pc(1, 2)
a12 + k012
sage: ark_pc(1, 2, algorithm='decrypt')
```

(continues on next page)

(continued from previous page)

```
a12 + k012
sage: ark_pc(2, 3, round=7)
a23 + k723
```

We can see how key variables are organized in the original key (the key used to build the rest of the subkeys) below. Note that because key variables are subkey entries, if the key length is longer than the block length we will have entries from multiple subkeys in the original key matrix.

```
sage: rgf.key_vrs
[k000 k001 k002 k003 k100 k101]
[k010 k011 k012 k013 k110 k111]
[k020 k021 k022 k023 k120 k121]
[k030 k031 k032 k033 k130 k131]
```

We can evaluate any of these constructed polynomials for a particular input state (in essence, calculate $\phi(A)_{i,j}$) as such:

```
sage: rgf = RijndaelGF(4, 6)
sage: state = rgf._hex_to_GF('fe7b5170fe7c8e93477f7e4bf6b98071')
sage: poly = mc_pc(3, 2, algorithm='decrypt')
sage: poly(state.list())
x^7 + x^6 + x^5 + x^2 + x
```

We can use the `apply_poly` method to build a matrix whose i, j th entry equals the polynomial `phi_poly(i, j)` evaluated for a particular input state, where `phi_poly` is the `Round_Component_Poly_Constr` object associated with the round component function ϕ. Essentially, `apply_poly` calculates $\phi(A)$, where A is our input state. Calling `apply_poly` is equivalent to applying the round component function associated this `Round_Component_Poly_Constr` object to A.

```
sage: state = rgf._hex_to_GF('c4cedcabe694694e4b23bfdd6fb522fa')
sage: result = rgf.apply_poly(state, rgf.sub_bytes_poly_constr())
sage: rgf._GF_to_hex(result)
'1c8b86628e22f92fb32608c1a8d5932d'
sage: result == rgf.sub_bytes(state)
True
```

Alternatively, we can pass a matrix of polynomials as input to `apply_poly`, which will then return another matrix of polynomials. For example, `rgf.state_vrs` can be used as input to make each `i,j` th entry of the output matrix equal `phi_poly_constr(i, j)`, where `phi_poly_constr` is our inputted `Round_Component_Poly_Constr` object. This matrix can then be passed through again and so on, demonstrating how one could potentially build a matrix of polynomials representing the entire cipher.

```
sage: state = rgf.apply_poly(rgf.state_vrs, rgf.shift_rows_poly_constr())
sage: state
[a00 a01 a02 a03]
[a11 a12 a13 a10]
[a22 a23 a20 a21]
[a33 a30 a31 a32]
sage: rgf.apply_poly(state, rgf.add_round_key_poly_constr())
[a00 + k000 a01 + k001 a02 + k002 a03 + k003]
[a11 + k010 a12 + k011 a13 + k012 a10 + k013]
[a22 + k020 a23 + k021 a20 + k022 a21 + k023]
[a33 + k030 a30 + k031 a31 + k032 a32 + k033]
```

For any of these `Round_Component_Poly_Constr` objects, we can change the keywords of its `__call__` method when `apply_poly` invokes it by passing `apply_poly` a dictionary mapping keywords to their values.

```
sage: rgf.apply_poly(rgf.state_vrs, rgf.add_round_key_poly_constr(),
....: poly_constr_attr={'round' : 5})
[a00 + k500 a01 + k501 a02 + k502 a03 + k503]
[a10 + k510 a11 + k511 a12 + k512 a13 + k513]
[a20 + k520 a21 + k521 a22 + k522 a23 + k523]
[a30 + k530 a31 + k531 a32 + k532 a33 + k533]
```

We can build our own `Round_Component_Poly_Constr` objects which correspond to the composition of multiple round component functions with the `compose` method. To do this, if we pass two `Round_Component_Poly_Constr` objects to `compose` where the first object corresponds to the round component function f and the second to the round component function g, `compose` will return a new `Round_Component_Poly_Constr` object corresponding to the function $g \circ f$. This returned `Round_Component_Poly_Constr` object will have the arguments of `__call__(row, col, algorithm='encrypt')` and when passed an index `i, j` will return $g(f(A))_{i,j}$ in terms of the entries of A.

```
sage: rcpc = rgf.compose(rgf.shift_rows_poly_constr(),
....: rgf.mix_columns_poly_constr())
sage: rcpc
A polynomial constructor of a round component of Rijndael-GF block cipher with block␣
↪length 4, key length 6, and 12 rounds.
sage: rcpc(2, 1)
a01 + a12 + (x)*a23 + (x + 1)*a30

sage: state = rgf._hex_to_GF('afb73eeb1cd1b85162280f27fb20d585')
sage: result = rgf.apply_poly(state, rcpc)
sage: new_state = rgf.shift_rows(state)
sage: new_state = rgf.mix_columns(new_state)
sage: result == new_state
True

sage: rcpc = rgf.compose(rgf.mix_columns_poly_constr(),
....: rgf.shift_rows_poly_constr())
sage: result = rgf.apply_poly(state, rcpc, algorithm='decrypt')
sage: new_state = rgf.mix_columns(state, algorithm='decrypt')
sage: new_state = rgf.shift_rows(new_state, algorithm='decrypt')
sage: new_state == result
True
```

Alternatively, we can use `compose` to build the polynomial output of a `Round_Component_Poly_Constr` object corresponding to the composition of multiple round functions like above without having to explicitly build our own `Round_Component_Poly_Constr` object. To do this, we simply make the first input a `Round_Component_Poly_Constr` object corresponding to a round component function f and make the second input a polynomial representing $g(A)_{i,j}$ for a round component function g. Given this, `compose` will return a polynomial representing $g(f(A))_{i,j}$ in terms of the entries of A.

```
sage: poly = rgf.mix_columns_poly_constr()(0, 3)
sage: poly
(x)*a03 + (x + 1)*a13 + a23 + a33
sage: rgf.compose(rgf.sub_bytes_poly_constr(), poly)
(x^3 + x)*a03^254 +
(x^3 + x^2 + x + 1)*a13^254 +
(x^2 + 1)*a23^254 +
(x^2 + 1)*a33^254 +
(x^4 + x)*a03^253 +
(x^4 + x^3 + x + 1)*a13^253 +
(x^3 + 1)*a23^253 +
```

(continues on next page)

(continued from previous page)

```
(x^3 + 1)*a33^253 +
(x^7 + x^6 + x^5 + x^3 + 1)*a03^251 +
(x^4)*a13^251 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*a23^251 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*a33^251 +
(x^6 + x^3 + x)*a03^247 +
(x^6 + x^5 + x^3 + x^2 + x + 1)*a13^247 +
(x^5 + x^2 + 1)*a23^247 +
(x^5 + x^2 + 1)*a33^247 +
(x^7 + x^6 + x^5 + x^4 + x + 1)*a03^239 +
(x^2 + x + 1)*a13^239 +
(x^7 + x^6 + x^5 + x^4 + x^2)*a23^239 +
(x^7 + x^6 + x^5 + x^4 + x^2)*a33^239 +
(x)*a03^223 +
(x + 1)*a13^223 +
a23^223 +
a33^223 +
(x^6 + x^5 + x^4 + 1)*a03^191 +
(x^7 + x^6 + x^2)*a13^191 +
(x^7 + x^5 + x^4 + x^2 + 1)*a23^191 +
(x^7 + x^5 + x^4 + x^2 + 1)*a33^191 +
(x^2 + 1)*a03^127 +
(x^7 + x^3 + x)*a13^127 +
(x^7 + x^3 + x^2 + x + 1)*a23^127 +
(x^7 + x^3 + x^2 + x + 1)*a33^127 +
(x^6 + x^5 + x + 1)
```

If we use `algorithm='decrypt'` as an argument to `compose`, then the value of `algorithm` will be passed directly to the first argument of `compose` (a `Round_Component_Poly_Constr` object) when it is called, provided the second argument is a polynomial. Setting this flag does nothing if both arguments are `Round_Component_Poly_Constr` objects, since the returned `Round_Component_Poly_Constr` object's `__call__` method must have its own `algorithm` keyword defaulted to 'encrypt'.

```
sage: poly = rgf.shift_rows_poly_constr()(2, 1)
sage: rgf.compose(rgf.mix_columns_poly_constr(), poly, algorithm='decrypt')
(x^3 + x^2 + 1)*a03 + (x^3 + 1)*a13 + (x^3 + x^2 + x)*a23 + (x^3 + x + 1)*a33

sage: state = rgf._hex_to_GF('80121e0776fd1d8a8d8c31bc965d1fee')
sage: with_decrypt = rgf.compose(rgf.sub_bytes_poly_constr(),
....: rgf.shift_rows_poly_constr(), algorithm='decrypt')
sage: result_wd = rgf.apply_poly(state, with_decrypt)
sage: no_decrypt = rgf.compose(rgf.sub_bytes_poly_constr(),
....: rgf.shift_rows_poly_constr())
sage: result_nd = rgf.apply_poly(state, no_decrypt)
sage: result_wd == result_nd
True
```

We can also pass keyword dictionaries of `f_attr` and `g_attr` to `compose` to make `f` and `g` use those keywords during polynomial creation.

```
sage: rcpc = rgf.compose(rgf.add_round_key_poly_constr(),
....: rgf.add_round_key_poly_constr(),
....: f_attr={'round' : 4}, g_attr={'round' : 7})
sage: rcpc(1, 2)
a12 + k412 + k712
```

In addition to building polynomial representations of state matrices, we can also build polynomial representations of

elements of the expanded key with the `expand_key_poly` method. However, since the key schedule is defined recursively, it is impossible to build polynomials for the key schedule in the same manner as we do for the round component functions. Consequently, `expand_round_key_poly()` is not a `Round_Component_Poly_Constr` object. Instead, `expand_key_poly` is a method which takes an index `i, j` and a round number `round`, and returns a polynomial representing the i, j th entry of the `round` th round key. This polynomial's variables are entries of the original key we built above.

```
sage: rgf.expand_key_poly(1, 2, 0)
k012
sage: rgf.expand_key_poly(1, 1, 1)
k111
sage: rgf.expand_key_poly(1, 2, 1)
(x^2 + 1)*k121^254 +
(x^3 + 1)*k121^253 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*k121^251 +
(x^5 + x^2 + 1)*k121^247 +
(x^7 + x^6 + x^5 + x^4 + x^2)*k121^239 +
k121^223 +
(x^7 + x^5 + x^4 + x^2 + 1)*k121^191 +
(x^7 + x^3 + x^2 + x + 1)*k121^127 +
k010 +
(x^6 + x^5 + x)
```

Since `expand_key_poly` is not actually a `Round_Component_Poly_Constr` object, we cannot use it as input to `apply_poly` or `compose`.

```
sage: rgf.apply_poly(state, rgf.expand_key_poly)
Traceback (most recent call last):
...
TypeError: keyword 'poly_constr' must be a Round_Component_Poly_Constr
sage: rgf.compose(rgf.expand_key_poly, rgf.sub_bytes_poly_constr())
Traceback (most recent call last):
...
TypeError: keyword 'f' must be a Round_Component_Poly_Constr
```

class sage.crypto.mq.rijndael_gf.**RijndaelGF**(*Nb*, *Nk*, *state_chr='a'*, *key_chr='k'*)

Bases: `sage.structure.sage_object.SageObject`

An algebraically generalized version of the AES cipher.

INPUT:

- `Nb` – The block length of this instantiation. Must be between 4 and 8.
- `Nk` – The key length of this instantiation. Must be between 4 and 8.
- `state_chr` – The variable name for polynomials representing elements from state matrices.
- `key_chr` – The variable name for polynomials representing elements of the key schedule.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(6, 8)
sage: rgf
Rijndael-GF block cipher with block length 6, key length 8, and 14 rounds.
```

By changing `state_chr` we can alter the names of variables in polynomials representing elements from state matrices.

```
sage: rgf = RijndaelGF(4, 6, state_chr='myChr')
sage: rgf.mix_columns_poly_constr()(3, 2)
(x + 1)*myChr02 + myChr12 + myChr22 + (x)*myChr32
```

We can also alter the name of variables in polynomials representing elements from round keys by changing `key_chr`.

```
sage: rgf = RijndaelGF(4, 6, key_chr='myKeyChr')
sage: rgf.expand_key_poly(1, 2, 1)
(x^2 + 1)*myKeyChr121^254 +
(x^3 + 1)*myKeyChr121^253 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*myKeyChr121^251 +
(x^5 + x^2 + 1)*myKeyChr121^247 +
(x^7 + x^6 + x^5 + x^4 + x^2)*myKeyChr121^239 +
myKeyChr121^223 +
(x^7 + x^5 + x^4 + x^2 + 1)*myKeyChr121^191 +
(x^7 + x^3 + x^2 + x + 1)*myKeyChr121^127 +
myKeyChr010 +
(x^6 + x^5 + x)
```

class Round_Component_Poly_Constr(*polynomial_constr*, *rgf*, *round_component_name=None*)

Bases: `sage.structure.sage_object.SageObject`

An object which constructs polynomials representing round component functions of a RijndaelGF object.

INPUT:

- `polynomial_constr` – A function which takes an index `row,col` and returns a polynomial representing the `row,col` th entry of a matrix after a specific round component function has been applied to it. This polynomial must be in terms of entries of the input matrix to that round component function and of entries of various subkeys. `polynomial_constr` must have arguments of the form `polynomial_constr(row, col, algorithm='encrypt', **kwargs)` and must be able to be called as `polynomial_constr(row, col)`.
- `rgf` – The RijndaelGF object whose state entries are represented by polynomials returned from `polynomial_constr`.
- `round_component_name` – The name of the round component function this object corresponds to as a string. Used solely for display purposes.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import \
....: RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: rcpc = RijndaelGF.Round_Component_Poly_Constr(
....: rgf._shift_rows_pc, rgf, "Shift Rows")
sage: rcpc
A polynomial constructor for the function 'Shift Rows' of Rijndael-GF block␣
↪cipher with block length 4, key length 4, and 10 rounds.
```

If ϕ is the round component function to which this object corresponds to, then `__call__(i,j)` $= \phi(A)_{i,j}$, where A is an arbitrary input matrix. Note that the polynomial returned by `__call__(i,j)` will be in terms of the entries of A.

```
sage: rcpc = RijndaelGF.Round_Component_Poly_Constr(
....: rgf._mix_columns_pc, rgf, "Mix Columns")
```

(continues on next page)

(continued from previous page)

```
sage: poly = rcpc(1, 2); poly
a02 + (x)*a12 + (x + 1)*a22 + a32
sage: state = rgf._hex_to_GF('d1876c0f79c4300ab45594add66ff41f')
sage: result = rgf.mix_columns(state)
sage: result[1,2] == poly(state.list())
True
```

Invoking this objects `__call__` method passes its arguments directly to `polynomial_constr` and returns the result. In a sense, `Round_Component_Poly_Constr` acts as a wrapper for the `polynomial_constr` method and helps ensure that each `Round_Component_Poly_Constr` object will act similarly.

```
sage: all([rgf._mix_columns_pc(i, j) == rcpc(i, j)
....: for i in range(4) for j in range(4)])
True
```

Since all keyword arguments of `polynomial_constr` must have a default value except for `row` and `col`, we can always call a `Round_Component_Poly_Constr` object by `__call__(row, col)`. Because of this, methods such as `apply_poly` and `compose` will only call `__call__(row, col)` when passed a `Round_Component_Poly_Constr` object. In order to change this object's behavior and force methods such as `apply_poly` to use non-default values for keywords we can pass dictionaries mapping keywords to non-default values as input to `apply_poly` and `compose`.

```
sage: rgf.apply_poly(rgf.state_vrs,
....: rgf.add_round_key_poly_constr(),
....: poly_constr_attr={'round' : 9})
[a00 + k900 a01 + k901 a02 + k902 a03 + k903]
[a10 + k910 a11 + k911 a12 + k912 a13 + k913]
[a20 + k920 a21 + k921 a22 + k922 a23 + k923]
[a30 + k930 a31 + k931 a32 + k932 a33 + k933]
```

```
sage: fn = rgf.compose(rgf.add_round_key_poly_constr(),
....: rgf.add_round_key_poly_constr(),
....: f_attr={'round' : 3}, g_attr={'round' : 7})
sage: fn(2, 3)
a23 + k323 + k723
```

Because all `Round_Component_Poly_Constr` objects are callable as `__call__(row, col, algorithm)`, `__call__` will check the validity of these three arguments automatically. Any other keywords, however, must be checked in `polynomial_constr`.

```
sage: def my_poly_constr(row, col, algorithm='encrypt'):
....:     return x * rgf._F.one() # example body with no checks
....:
sage: rcpc = RijndaelGF.Round_Component_Poly_Constr(
....: my_poly_constr, rgf, "My Poly Constr")
sage: rcpc(-1, 2)
Traceback (most recent call last):
...
ValueError: keyword 'row' must be in range 0 - 3
sage: rcpc(1, 2, algorithm=5)
Traceback (most recent call last):
...
ValueError: keyword 'algorithm' must be either 'encrypt' or 'decrypt'
```

add_round_key(*state*, *round_key*)

Returns the round-key addition of matrices `state` and `round_key`.

INPUT:

- `state` – The state matrix to have `round_key` added to.
- `round_key` – The round key to add to `state`.

OUTPUT:

- A state matrix which is the round key addition of `state` and `round_key`. This transformation is simply the entrywise addition of these two matrices.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: state = rgf._hex_to_GF('36339d50f9b539269f2c092dc4406d23')
sage: key = rgf._hex_to_GF('7CC78D0E22754E667E24573F454A6531')
sage: key_schedule = rgf.expand_key(key)
sage: result = rgf.add_round_key(state, key_schedule[0])
sage: rgf._GF_to_hex(result)
'4af4105edbc07740e1085e12810a0812'
```

add_round_key_poly_constr()

Return the `Round_Component_Poly_Constr` object corresponding to AddRoundKey.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: ark_pc = rgf.add_round_key_poly_constr()
sage: ark_pc
A polynomial constructor for the function 'Add Round Key' of Rijndael-GF␣
↪block cipher with block length 4, key length 4, and 10 rounds.
sage: ark_pc(0, 1)
a01 + k001
```

When invoking the returned object's `__call__` method, changing the value of `algorithm='encrypt'` does nothing, since the AddRoundKey round component function is its own inverse.

```
sage: with_encrypt = ark_pc(1, 1, algorithm='encrypt')
sage: with_decrypt = ark_pc(1, 1, algorithm='decrypt')
sage: with_encrypt == with_decrypt
True
```

When invoking the returned object's `__call__` method, one can change the round subkey used in the returned polynomial by changing the `round=0` keyword.

```
sage: ark_pc(2, 1, round=7)
a21 + k721
```

When passing the returned object to methods such as `apply_poly` and `compose`, we can make these methods use a non-default value for `round=0` by passing in a dictionary mapping `round` to a different value.

```
sage: rgf.apply_poly(rgf.state_vrs, ark_pc,
....: poly_constr_attr={'round' : 6})
[a00 + k600 a01 + k601 a02 + k602 a03 + k603]
```

(continues on next page)

(continued from previous page)

```
[a10 + k610 a11 + k611 a12 + k612 a13 + k613]
[a20 + k620 a21 + k621 a22 + k622 a23 + k623]
[a30 + k630 a31 + k631 a32 + k632 a33 + k633]
```

```
sage: rcpc = rgf.compose(ark_pc, ark_pc,
....: f_attr={'round' : 3}, g_attr={'round' : 5})
sage: rcpc(3, 1)
a31 + k331 + k531
```

apply_poly(*state*, *poly_constr*, *algorithm='encrypt'*, *keys=None*, *poly_constr_attr=None*)

Returns a state matrix where `poly_method` is applied to each entry.

INPUT:

- `state` – The state matrix over $\mathbf{F}_{2^8}$ to which `poly_method` is applied to.
- `poly_constr` – The `Round_Component_Poly_Constr` object to build polynomials during evaluation.
- `algorithm` – (default: "encrypt") Passed directly to `rcpc` to select encryption or decryption. The encryption flag is "encrypt" and the decrypt flag is "decrypt".
- `keys` – (default: None) An array of N_r subkey matrices to replace any key variables in any polynomials returned by `poly_method`. Must be identical to the format returned by `expand_key`. If any polynomials have key variables and `keys` is not supplied, the key variables will remain as-is.
- `poly_constr_attr` – (default:None) A dictionary of keyword attributes to pass to `rcpc` when it is called.

OUTPUT:

- A state matrix in $\mathbf{F}_{2^8}$ whose i, j th entry equals the polynomial `poly_constr(i, j, algorithm, **poly_constr_attr)` evaluated by setting its variables equal to the corresponding entries of `state`.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: state = rgf._hex_to_GF('3b59cb73fcd90ee05774222dc067fb68')
sage: result = rgf.apply_poly(state, rgf.shift_rows_poly_constr())
sage: rgf._GF_to_hex(result)
'3bd92268fc74fb735767cbe0c0590e2d'
```

Calling `apply_poly` with the `Round_Component_Poly_Constr` object of a round component (e.g. `sub_bytes_poly`) is identical to calling that round component function itself.

```
sage: state = rgf._hex_to_GF('4915598f55e5d7a0daca94fa1f0a63f7')
sage: apply_poly_result = rgf.apply_poly(state,
....: rgf.sub_bytes_poly_constr())
sage: direct_result = rgf.sub_bytes(state)
sage: direct_result == apply_poly_result
True
```

If the `Round_Component_Poly_Constr` object's `__call__` method returns a polynomial with state variables as well as key variables, we can supply a list of N_r round keys `keys` whose elements are evaluated as the key variables. If this is not provided, the key variables will remain as is.:

```
sage: state = rgf._hex_to_GF('14f9701ae35fe28c440adf4d4ea9c026')
sage: key = rgf._hex_to_GF('54d990a16ba09ab596bbf40ea111702f')
sage: keys = rgf.expand_key(key)
sage: result = rgf.apply_poly(state,
....: rgf.add_round_key_poly_constr(), keys=keys)
sage: result == rgf.add_round_key(state, key)
True

sage: rgf.apply_poly(state, rgf.add_round_key_poly_constr())[0,0]
k000 + (x^4 + x^2)
```

We can change the value of the keywords of `poly_constr` 's `__call__` method when `apply_poly` calls it by passing in a dictionary `poly_constr_attr` mapping keywords to their values.

```
sage: rgf.apply_poly(rgf.state_vrs,
....: rgf.add_round_key_poly_constr(),
....: poly_constr_attr={'round' : 5})
[a00 + k500 a01 + k501 a02 + k502 a03 + k503]
[a10 + k510 a11 + k511 a12 + k512 a13 + k513]
[a20 + k520 a21 + k521 a22 + k522 a23 + k523]
[a30 + k530 a31 + k531 a32 + k532 a33 + k533]
```

block_length()

Returns the block length of this instantiation of Rijndael-GF.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 6)
sage: rgf.block_length()
4
```

compose(*f*, *g*, *algorithm='encrypt'*, *f_attr=None*, *g_attr=None*)

Return a `Round_Component_Poly_Constr` object corresponding to $g \circ f$ or the polynomial output of this object's `__call__` method.

INPUT:

- `f` – A `Round_Component_Poly_Constr` object corresponding to a round component function f.
- `g` – A `Round_Component_Poly_Constr` object corresponding to a round component function g or a polynomial output of this object's `__call__` method.
- `algorithm` – (default: "encrypt") Whether `f` and `g` should use their encryption transformations or their decryption transformations. Does nothing if `g` is a `Round_Component_Poly_Constr` object. The encryption flag is "encrypt" and the decryption flag is "decrypt".
- `f_attr` – (default: None) A dictionary of keyword attributes to pass to `f` when it is called.
- `g_attr` – (default: None) A dictionary of keyword attributes to pass to `g` when it is called. Does nothing if `g` is a polynomial.

OUTPUT:

- If `g` is a `Round_Component_Poly_Constr` object corresponding to a round component function g, then `compose` returns a `Round_Component_Poly_Constr` corresponding to the round component function $g \circ f$, where f is the round component function corresponding to the first argument `f`. On the other hand, if `g` $= g(A)_{i,j}$ for a round component function g, then `compose` returns $g(f(A))_{i,j}$, where A is an arbitrary input state matrix.

EXAMPLES

This function allows us to determine the polynomial representations of entries across multiple round functions. For example, if we wanted a polynomial representing the `1, 3` entry of a matrix after we first apply ShiftRows and then MixColumns to that matrix, we do:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: mcp = rgf.mix_columns_poly_constr()(1, 3); mcp
a03 + (x)*a13 + (x + 1)*a23 + a33
sage: result = rgf.compose(rgf.shift_rows_poly_constr(), mcp)
sage: result
a03 + (x)*a10 + (x + 1)*a21 + a32
```

We can test the correctness of this:

```
sage: state = rgf._hex_to_GF('fa636a2825b339c940668a3157244d17')
sage: new_state = rgf.shift_rows(state)
sage: new_state = rgf.mix_columns(new_state)
sage: result(state.list()) == new_state[1,3]
True
```

We can also use `compose` to build a new `Round_Component_Poly_Constr` object corresponding to the composition of multiple round functions as such:

```
sage: fn = rgf.compose(rgf.shift_rows_poly_constr(),
....: rgf.mix_columns_poly_constr())
sage: fn(1, 3)
a03 + (x)*a10 + (x + 1)*a21 + a32
```

If we use `compose` to make a new `Round_Component_Poly_Constr` object, we can use that object as input to `apply_poly` and `compose`:

```
sage: state = rgf._hex_to_GF('36400926f9336d2d9fb59d23c42c3950')
sage: result = rgf.apply_poly(state, fn)
sage: rgf._GF_to_hex(result)
'f4bcd45432e554d075f1d6c51dd03b3c'

sage: new_state = rgf.shift_rows(state)
sage: new_state = rgf.mix_columns(new_state)
sage: result == new_state
True
```

```
sage: fn2 = rgf.compose(rgf.sub_bytes_poly_constr(), fn)
```

If the second argument is a polynomial, then the value of `algorithm` is passed directly to the first argument f during evaluation. However, if the second argument is a `Round_Component_Poly_Constr` object, changing `algorithm` does nothing since the returned object has its own `algorithm='encrypt'` keyword.

```
sage: f = rgf.compose(rgf.sub_bytes_poly_constr(),
....: rgf.mix_columns_poly_constr(), algorithm='decrypt')
sage: g = rgf.compose(rgf.sub_bytes_poly_constr(),
....: rgf.mix_columns_poly_constr())
sage: all([f(i,j) == g(i,j) for i in range(4) for j in range(4)])
True
```

We can change the keyword attributes of the `__call__` methods of `f` and `g` by passing dictionaries `f_attr` and `g_attr` to `compose`.

```
sage: fn = rgf.compose(rgf.add_round_key_poly_constr(),
....: rgf.add_round_key_poly_constr(),
....: f_attr={'round' : 4}, g_attr={'round' : 7})
sage: fn(1, 2)
a12 + k412 + k712
```

decrypt (*ciphertext*, *key*, *format='hex'*)

Returns the ciphertext `ciphertext` decrypted with the key `key`.

INPUT:

- `ciphertext` – The ciphertext to be decrypted.
- `key` – The key to decrypt `ciphertext` with.
- `format` – (default: `hex`) The string format that both `ciphertext` and `key` must be in, either "hex" or "binary".

OUTPUT:

- A string in the format `format` of `ciphertext` decrypted with key `key`.

EXAMPLES

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: key = '2dfb02343f6d12dd09337ec75b36e3f0'
sage: ciphertext = '54d990a16ba09ab596bbf40ea111702f'
sage: expected_plaintext = '1e1d913b7274ad9b5a4ab1a5f9133b93'
sage: rgf.decrypt(ciphertext, key) == expected_plaintext
True
```

We can also decrypt messages using binary strings.

```
sage: key = '00011010000011100011000000111101' * 4
sage: ciphertext = '00110010001110000111110110000001' * 4
sage: expected_plaintext = ('10111111101001110011110010101010100111'
....: '11110100001011011000011010000000000000000010000000100111011'
....: '01000011111000110100011011010011011')
sage: result = rgf.decrypt(ciphertext, key, format='binary')
sage: result == expected_plaintext
True
```

encrypt (*plain*, *key*, *format='hex'*)

Returns the plaintext `plain` encrypted with the key `key`.

INPUT:

- `plain` – The plaintext to be encrypted.
- `key` – The key to encrypt `plain` with.
- `format` – (default: `hex`) The string format of `key` and `plain`, either "hex" or "binary".

OUTPUT:

- A string of the plaintext `plain` encrypted with the key `key`.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: key = 'c81677bc9b7ac93b25027992b0261996'
sage: plain = 'fde3bad205e5d0d73547964ef1fe37f1'
sage: expected_ciphertext = 'e767290ddfc6414e3c50a444bec081f0'
sage: rgf.encrypt(plain, key) == expected_ciphertext
True
```

We can encrypt binary strings as well.

```
sage: key = '10010111110000011111011011010001' * 4
sage: plain = '00000000101000000000000001111011' * 4
sage: expected_ciphertext = ('11010111100100001010001011110010111'
....: '11100110000000111111001000110111001010000000010001111000010'
....: '00100111011011001000111101111110100')
sage: result = rgf.encrypt(plain, key, format='binary')
sage: result == expected_ciphertext
True
```

expand_key (*key*)

Returns the expanded key schedule from `key`.

INPUT:

- `key` – The key to build a key schedule from. Must be a matrix over $\mathbf{F}_{2^8}$ of dimensions $4 \times N_k$.

OUTPUT:

- A length Nr list of $4 \times N_b$ matrices corresponding to the expanded key. The n th entry of the list corresponds to the matrix used in the `add_round_key` step of the n th round.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 6)
sage: key = '331D0084B176C3FB59CAA0EDA271B565BB5D9A2D1E4B2892'
sage: key_state = rgf._hex_to_GF(key)
sage: key_schedule = rgf.expand_key(key_state)
sage: rgf._GF_to_hex(key_schedule[0])
'331d0084b176c3fb59caa0eda271b565'
sage: rgf._GF_to_hex(key_schedule[6])
'5c5d51c4121f018d0f4f3e408ae9f78c'
```

expand_key_poly (*row*, *col*, *round*)

Returns a polynomial representing the `row,col` th entry of the `round` th round key.

INPUT:

- `row` – The row position of the element represented by this polynomial.
- `col` – The column position of the element represented by this polynomial.

OUTPUT:

- A polynomial representing the `row,col` th entry of the `round` th round key in terms of entries of the input key.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
```

(continues on next page)

(continued from previous page)

```
sage: rgf.expand_key_poly(1, 2, 0)
k012
sage: rgf.expand_key_poly(1, 2, 1)
(x^2 + 1)*k023^254 +
(x^3 + 1)*k023^253 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*k023^251 +
(x^5 + x^2 + 1)*k023^247 +
(x^7 + x^6 + x^5 + x^4 + x^2)*k023^239 +
k023^223 +
(x^7 + x^5 + x^4 + x^2 + 1)*k023^191 +
(x^7 + x^3 + x^2 + x + 1)*k023^127 +
k010 +
k011 +
k012 +
(x^6 + x^5 + x)
```

It should be noted that `expand_key_poly` cannot be used with `apply_poly` or `compose`, since `expand_key_poly` is not a `Round_Component_Poly_Constr` object.

```
sage: rgf.compose(rgf.sub_bytes_poly_constr(), rgf.expand_key_poly)
Traceback (most recent call last):
...
TypeError: keyword 'g' must be a Round_Component_Poly_Constr or a polynomial␣
↪over Finite Field in x of size 2^8

sage: state = rgf._hex_to_GF('00000000000000000000000000000000')
sage: rgf.apply_poly(state, rgf.expand_key_poly)
Traceback (most recent call last):
...
TypeError: keyword 'poly_constr' must be a Round_Component_Poly_Constr
```

key_length()

Returns the key length of this instantiation of Rijndael-GF.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 8)
sage: rgf.key_length()
8
```

mix_columns(*state*, *algorithm='encrypt'*)

Returns the application of MixColumns to the state matrix `state`.

INPUT:

- `state` – The state matrix to apply MixColumns to.
- `algorithm` – (default: "encrypt") Whether to perform the encryption version of MixColumns, or its decryption inverse. The encryption flag is "encrypt" and the decryption flag is "decrypt".

OUTPUT:

- The state matrix over $\mathbf{F}_{2^8}$ which is the result of applying MixColumns to `state`.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
```

(continues on next page)

(continued from previous page)

```
sage: state = rgf._hex_to_GF('cd54c7283864c0c55d4c727e90c9a465')
sage: result = rgf.mix_columns(state)
sage: rgf._GF_to_hex(result)
'921f748fd96e937d622d7725ba8ba50c'
sage: decryption = rgf.mix_columns(result, algorithm='decrypt')
sage: decryption == state
True
```

mix_columns_poly_constr()

Return a Round_Component_Poly_Constr object corresponding to MixColumns.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: mc_pc = rgf.mix_columns_poly_constr()
sage: mc_pc
A polynomial constructor for the function 'Mix Columns' of Rijndael-GF block␣
↪cipher with block length 4, key length 4, and 10 rounds.
sage: mc_pc(1, 2)
a02 + (x)*a12 + (x + 1)*a22 + a32
sage: mc_pc(1, 0, algorithm='decrypt')
(x^3 + 1)*a00 + (x^3 + x^2 + x)*a10 + (x^3 + x + 1)*a20 + (x^3 + x^2 + 1)*a30
```

The returned object's __call__ method has no additional keywords, unlike sub_bytes_poly_constr() and add_round_key_poly_constr().

number_rounds()

Returns the number of rounds used in this instantiation of Rijndael-GF.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(5, 4)
sage: rgf.number_rounds()
11
```

shift_rows(*state*, *algorithm='encrypt'*)

Returns the application of ShiftRows to the state matrix state.

INPUT:

- state – A state matrix over $\mathbf{F}_{2^8}$ to which ShiftRows is applied to.
- algorithm – (default: "encrypt") Whether to perform the encryption version of ShiftRows or its decryption inverse. The encryption flag is "encrypt" and the decryption flag is "decrypt".

OUTPUT:

- A state matrix over $\mathbf{F}_{2^8}$ which is the application of ShiftRows to state.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: state = rgf._hex_to_GF('adcb0f257e9c63e0bc557e951c15ef01')
sage: result = rgf.shift_rows(state)
sage: rgf._GF_to_hex(result)
'ad9c7e017e55ef25bc150fe01ccb6395'
```

(continues on next page)

(continued from previous page)

```
sage: decryption = rgf.shift_rows(result, algorithm='decrypt')
sage: decryption == state
True
```

shift_rows_poly_constr()

Return a Round_Component_Poly_Constr object corresponding to ShiftRows.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: sr_pc = rgf.shift_rows_poly_constr()
sage: sr_pc(3, 0)
a33
sage: sr_pc(2, 1, algorithm='decrypt')
a23
```

The returned object's __call__ method has no additional keywords, unlike sub_bytes_poly_constr() and add_round_key_poly_constr.

sub_bytes(*state*, *algorithm='encrypt'*)

Returns the application of SubBytes to the state matrix state.

INPUT:

- state – The state matrix to apply SubBytes to.
- algorithm – (default: "encrypt") Whether to apply the encryption step of SubBytes or its decryption inverse. The encryption flag is "encrypt" and the decryption flag is "decrypt".

OUTPUT:

- The state matrix over $\mathbf{F}_{2^8}$ where SubBytes has been applied to every entry of state.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: state = rgf._hex_to_GF('d1c4941f7955f40fb46f6c0ad68730ad')
sage: result = rgf.sub_bytes(state)
sage: rgf._GF_to_hex(result)
'3e1c22c0b6fcbf768da85067f6170495'
sage: decryption = rgf.sub_bytes(result, algorithm='decrypt')
sage: decryption == state
True
```

sub_bytes_poly_constr()

Return the Round_Component_Poly_Constr object corresponding to SubBytes.

EXAMPLES:

```
sage: from sage.crypto.mq.rijndael_gf import RijndaelGF
sage: rgf = RijndaelGF(4, 4)
sage: sb_pc = rgf.sub_bytes_poly_constr()
sage: sb_pc
A polynomial constructor for the function 'SubBytes' of Rijndael-GF block␣
↪cipher with block length 4, key length 4, and 10 rounds.
sage: sb_pc(2, 3)
(x^2 + 1)*a23^254 +
```

(continues on next page)

(continued from previous page)

```
(x^3 + 1)*a23^253 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*a23^251 +
(x^5 + x^2 + 1)*a23^247 +
(x^7 + x^6 + x^5 + x^4 + x^2)*a23^239 +
a23^223 +
(x^7 + x^5 + x^4 + x^2 + 1)*a23^191 +
(x^7 + x^3 + x^2 + x + 1)*a23^127 +
(x^6 + x^5 + x + 1)
```

The returned object's `__call__` method has an additional keyword of `no_inversion=False`, which causes the returned polynomial to represent only the affine transformation step of SubBytes.

```
sage: sb_pc(1, 0, no_inversion=True)
(x^7 + x^3 + x^2 + x + 1)*a10^128 +
(x^7 + x^5 + x^4 + x^2 + 1)*a10^64 +
a10^32 +
(x^7 + x^6 + x^5 + x^4 + x^2)*a10^16 +
(x^5 + x^2 + 1)*a10^8 +
(x^7 + x^6 + x^5 + x^4 + x^3 + 1)*a10^4 +
(x^3 + 1)*a10^2 +
(x^2 + 1)*a10 +
(x^6 + x^5 + x + 1)
```

We can build a polynomial representing the inverse transformation by setting the keyword `algorithm='decrypt'`. However, the order of the affine transformation and the inversion step in SubBytes means that this polynomial has thousands of terms and is very slow to compute. Hence, if one wishes to build the decryption polynomial with the intention of evaluating that polynomial for a particular input, it is strongly recommended to first call `sb_pc(i, j, algorithm='decrypt', no_inversion=True)` to build a polynomial representing only the inverse affine transformation, evaluate this polynomial for your intended input, then finally calculate the inverse of the result.

```
sage: poly = sb_pc(1, 2, algorithm='decrypt', no_inversion=True)
sage: state = rgf._hex_to_GF('39daee38f4f1a82aaf432410c36d45b9')
sage: result = poly(state.list())
sage: rgf._GF_to_hex(result * -1)
'49'
```

When passing the returned object to `apply_poly` and `compose`, we can make those methods change the keyword `no_inversion` of this object's `__call__` method by passing the dictionary `{'no_inversion' : True}` to them.

```
sage: result = rgf.apply_poly(state, sb_pc,
....: poly_constr_attr={'no_inversion' : True})
sage: rgf._GF_to_hex(result)
'961c72894526f746aa85fc920adcc719'
```

```
sage: rcpc = rgf.compose(sb_pc, rgf.shift_rows_poly_constr(),
....: f_attr={'no_inversion' : True})
```

Note that if we set `algorithm='decrypt'` for `apply_poly`, it will perform the necessary performance enhancement described above automatically. The structure of `compose`, however, unfortunately does not allow this enhancement to be employed.

CHAPTER

SEVENTEEN

HARD LATTICE GENERATOR

This module contains lattice related functions relevant in cryptography.

Feel free to add more functionality.

AUTHORS:

- Richard Lindner <rlindner@cdc.informatik.tu-darmstadt.de>
- Michael Schneider <mischnei@cdc.informatik.tu-darmstadt.de>

sage.crypto.lattice.**gen_lattice**(*type='modular'*, *n=4*, *m=8*, *q=11*, *seed=None*, *quotient=None*, *dual=False*, *ntl=False*, *lattice=False*)

This function generates different types of integral lattice bases of row vectors relevant in cryptography.

Randomness can be set either with `seed`, or by using `sage.misc.randstate.set_random_seed()`.

INPUT:

- **`type` – one of the following strings**
  - `'modular'` (default) – A class of lattices for which asymptotic worst-case to average-case connections hold. For more refer to [Aj1996].
  - `'random'` – Special case of modular (n=1). A dense class of lattice used for testing basis reduction algorithms proposed by Goldstein and Mayer [GM2002].
  - `'ideal'` – Special case of modular. Allows for a more compact representation proposed by [LM2006].
  - `'cyclotomic'` – Special case of ideal. Allows for efficient processing proposed by [LM2006].
- `n` – Determinant size, primal:$det(L) = q^n$, dual:$det(L) = q^{m-n}$. For ideal lattices this is also the degree of the quotient polynomial.
- `m` – Lattice dimension, $L \subseteq Z^m$.
- `q` – Coefficient size, $q - Z^m \subseteq L$.
- `seed` – Randomness seed.
- `quotient` – For the type ideal, this determines the quotient polynomial. Ignored for all other types.
- `dual` – Set this flag if you want a basis for $q - dual(L)$, for example for Regev's LWE bases [Reg2005].
- `ntl` – Set this flag if you want the lattice basis in NTL readable format.
- `lattice` – Set this flag if you want a `FreeModule_submodule_with_basis_integer` object instead of an integer matrix representing the basis.

OUTPUT: `B` a unique size-reduced triangular (primal: lower_left, dual: lower_right) basis of row vectors for the lattice in question.

EXAMPLES:

Modular basis:

```
sage: sage.crypto.gen_lattice(m=10, seed=42)
[11  0  0  0  0  0  0  0  0  0]
[ 0 11  0  0  0  0  0  0  0  0]
[ 0  0 11  0  0  0  0  0  0  0]
[ 0  0  0 11  0  0  0  0  0  0]
[ 2  4  3  5  1  0  0  0  0  0]
[ 1 -5 -4  2  0  1  0  0  0  0]
[-4  3 -1  1  0  0  1  0  0  0]
[-2 -3 -4 -1  0  0  0  1  0  0]
[-5 -5  3  3  0  0  0  0  1  0]
[-4 -3  2 -5  0  0  0  0  0  1]
```

Random basis:

```
sage: sage.crypto.gen_lattice(type='random', n=1, m=10, q=11^4, seed=42)
[14641     0     0     0     0     0     0     0     0     0]
[  431     1     0     0     0     0     0     0     0     0]
[-4792     0     1     0     0     0     0     0     0     0]
[ 1015     0     0     1     0     0     0     0     0     0]
[-3086     0     0     0     1     0     0     0     0     0]
[-5378     0     0     0     0     1     0     0     0     0]
[ 4769     0     0     0     0     0     1     0     0     0]
[-1159     0     0     0     0     0     0     1     0     0]
[ 3082     0     0     0     0     0     0     0     1     0]
[-4580     0     0     0     0     0     0     0     0     1]
```

Ideal bases with quotient x^n-1, m=2*n are NTRU bases:

```
sage: sage.crypto.gen_lattice(type='ideal', seed=42, quotient=x^4-1)
[11  0  0  0  0  0  0  0]
[ 0 11  0  0  0  0  0  0]
[ 0  0 11  0  0  0  0  0]
[ 0  0  0 11  0  0  0  0]
[ 4 -2 -3 -3  1  0  0  0]
[-3  4 -2 -3  0  1  0  0]
[-3 -3  4 -2  0  0  1  0]
[-2 -3 -3  4  0  0  0  1]
```

Ideal bases also work with polynomials:

```
sage: R.<t> = PolynomialRing(ZZ)
sage: sage.crypto.gen_lattice(type='ideal', seed=1234, quotient=t^4-1)
[11  0  0  0  0  0  0  0]
[ 0 11  0  0  0  0  0  0]
[ 0  0 11  0  0  0  0  0]
[ 0  0  0 11  0  0  0  0]
[ 4  1  4 -3  1  0  0  0]
[-3  4  1  4  0  1  0  0]
[ 4 -3  4  1  0  0  1  0]
[ 1  4 -3  4  0  0  0  1]
```

Cyclotomic bases with n=2^k are SWIFFT bases:

```
sage: sage.crypto.gen_lattice(type='cyclotomic', seed=42)
[11  0  0  0  0  0  0  0]
[ 0 11  0  0  0  0  0  0]
[ 0  0 11  0  0  0  0  0]
[ 0  0  0 11  0  0  0  0]
[ 4 -2 -3 -3  1  0  0  0]
[ 3  4 -2 -3  0  1  0  0]
[ 3  3  4 -2  0  0  1  0]
[ 2  3  3  4  0  0  0  1]
```

Dual modular bases are related to Regev's famous public-key encryption [Reg2005]:

```
sage: sage.crypto.gen_lattice(type='modular', m=10, seed=42, dual=True)
[ 0  0  0  0  0  0  0  0  0 11]
[ 0  0  0  0  0  0  0  0 11  0]
[ 0  0  0  0  0  0  0 11  0  0]
[ 0  0  0  0  0  0 11  0  0  0]
[ 0  0  0  0  0 11  0  0  0  0]
[ 0  0  0  0 11  0  0  0  0  0]
[ 0  0  0  1 -5 -2 -1  1 -3  5]
[ 0  0  1  0 -3  4  1  4 -3 -2]
[ 0  1  0  0 -4  5 -3  3  5  3]
[ 1  0  0  0 -2 -1  4  2  5  4]
```

Relation of primal and dual bases:

```
sage: B_primal=sage.crypto.gen_lattice(m=10, q=11, seed=42)
sage: B_dual=sage.crypto.gen_lattice(m=10, q=11, seed=42, dual=True)
sage: B_dual_alt=transpose(11*B_primal.inverse()).change_ring(ZZ)
sage: B_dual_alt.hermite_form() == B_dual.hermite_form()
True
```

CHAPTER

EIGHTEEN

(RING-)LWE ORACLE GENERATORS

The Learning with Errors problem (LWE) is solving linear systems of equations where the right hand side has been disturbed 'slightly' where 'slightly' is made precise by a noise distribution - typically a discrete Gaussian distribution. See [Reg09] for details.

The Ring Learning with Errors problem (LWE) is solving a set of univariate polynomial equations - typically in a cyclotomic field - where the right hand side was disturbed 'slightly'. See [LPR2010] for details.

This module implements generators of LWE samples where parameters are chosen following proposals in the cryptographic literature.

EXAMPLES:

We get 30 samples from an LWE oracle parameterised by security parameter `n=20` and where the modulus and the standard deviation of the noise are chosen as in [Reg09]:

```
sage: from sage.crypto.lwe import samples
sage: samples(30, 20, 'Regev')
[((360, 264, 123, 368, 398, 392, 41, 84, 25, 389, 311, 68, 322, 41, 161, 372, 222,␣
↪153, 243, 381), 122),
...
((155, 22, 357, 312, 87, 298, 182, 163, 296, 181, 219, 135, 164, 308, 248, 320, 64,␣
↪166, 214, 104), 152)]
```

We may also pass classes to the samples function, which is useful for users implementing their own oracles:

```
sage: from sage.crypto.lwe import samples, LindnerPeikert
sage: samples(30, 20, LindnerPeikert)
[((1275, 168, 1529, 2024, 1874, 1309, 16, 1869, 1114, 1696, 1645, 618, 1372, 1273,␣
↪683, 237, 1526, 879, 1305, 1355), 950),
...
((1787, 2033, 1677, 331, 1562, 49, 796, 1002, 627, 98, 91, 711, 1712, 418, 2024, 163,␣
↪1773, 184, 1548, 3), 1815)]
```

Finally, *samples()* also accepts instances of classes:

```
sage: from sage.crypto.lwe import LindnerPeikert
sage: lwe = LindnerPeikert(20)
sage: samples(30, 20, lwe)
[((465, 180, 440, 706, 1367, 106, 1380, 614, 1162, 1354, 1098, 2036, 1974, 1417, 1502,
↪ 1431, 863, 1894, 1368, 1771), 618),
...
((1050, 1017, 1314, 1310, 1941, 2041, 484, 104, 1199, 1744, 161, 1905, 679, 1663, 531,
↪ 1630, 168, 1559, 1040, 1719), 1006)]
```

Note that Ring-LWE samples are returned as vectors:

```
sage: from sage.crypto.lwe import RingLWE
sage: from sage.stats.distributions.discrete_gaussian_polynomial import␣
↪DiscreteGaussianDistributionPolynomialSampler
sage: D = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], euler_phi(16), 5)
sage: ringlwe = RingLWE(16, 257, D, secret_dist='uniform')
sage: samples(30, euler_phi(16), ringlwe)
[((41, 78, 232, 79, 223, 85, 26, 68), (195, 99, 106, 57, 93, 113, 23, 68)),
...
((185, 89, 244, 122, 249, 140, 173, 142), (98, 196, 70, 49, 55, 8, 158, 57))]
```

One technical issue when working with these generators is that by default they return vectors and scalars over/in rings modulo some q. These are represented as elements in $(0, q-1)$ by Sage. However, it usually is more natural to think of these entries as integers in $(-q//2, q//2)$. To allow for this, this module provides the option to balance the representation. In this case vectors and scalars over/in the integers are returned:

```
sage: from sage.crypto.lwe import samples
sage: samples(30, 20, 'Regev', balanced=True)
[((-105, 43, -25, -16, 57, 141, -108, 92, -173, 4, 179, -191, 164, 101, -16, -175,␣
↪172, 10, 147, 1), 114),
...
((-166, -147, 120, -56, 130, 163, 83, 17, -125, -159, -124, 19, 198, -181, -124, -155,
↪ 84, -15, -113, 113), 39)]
```

AUTHORS:

- Martin Albrecht
- Robert Fitzpatrick
- Daniel Cabracas
- Florian Göpfert
- Michael Schneider

REFERENCES:

- [Reg09]
- [LP2011]
- [LPR2010]
- [CGW2013]

class sage.crypto.lwe.**LWE**(*n*, *q*, *D*, *secret_dist='uniform'*, *m=None*)

Bases: sage.structure.sage_object.SageObject

Learning with Errors (LWE) oracle.

__init__(*n*, *q*, *D*, *secret_dist='uniform'*, *m=None*)

Construct an LWE oracle in dimension `n` over a ring of order `q` with noise distribution `D`.

INPUT:

- `n` - dimension (integer > 0)
- `q` - modulus typically > n (integer > 0)
- `D` - an error distribution such as an instance of `DiscreteGaussianDistributionIntegerSampler` or *`UniformSampler`*
- `secret_dist` - distribution of the secret (default: 'uniform'); one of

- "uniform" - secret follows the uniform distribution in $\mathbf{Z}/q\mathbf{Z}$
- "noise" - secret follows the noise distribution
- `(lb,ub)` - the secret is chosen uniformly from `[lb,...,ub]` including both endpoints

• `m` - number of allowed samples or `None` if no such limit exists (default: `None`)

EXAMPLES:

First, we construct a noise distribution with standard deviation 3.0:

```
sage: from sage.stats.distributions.discrete_gaussian_integer import␣
↪DiscreteGaussianDistributionIntegerSampler
sage: D = DiscreteGaussianDistributionIntegerSampler(3.0)
```

Next, we construct our oracle:

```
sage: from sage.crypto.lwe import LWE
sage: lwe = LWE(n=20, q=next_prime(400), D=D); lwe
LWE(20, 401, Discrete Gaussian sampler over the Integers with sigma = 3.
↪000000 and c = 0, 'uniform', None)
```

and sample 1000 samples:

```
sage: L = [lwe() for _ in range(1000)]
```

To test the oracle, we use the internal secret to evaluate the samples in the secret:

```
sage: S = [ZZ(a.dot_product(lwe._LWE__s) - c) for (a,c) in L]
```

However, while Sage represents finite field elements between 0 and q-1 we rely on a balanced representation of those elements here. Hence, we fix the representation and recover the correct standard deviation of the noise:

```
sage: sqrt(variance([e if e <= 200 else e-401 for e in S]).n())
3.0...
```

If `m` is not `None` the number of available samples is restricted:

```
sage: from sage.crypto.lwe import LWE
sage: lwe = LWE(n=20, q=next_prime(400), D=D, m=30)
sage: _ = [lwe() for _ in range(30)]
sage: lwe() # 31
Traceback (most recent call last):
...
IndexError: Number of available samples exhausted.
```

__call__()

EXAMPLES:

```
sage: from sage.crypto.lwe import DiscreteGaussianDistributionIntegerSampler,␣
↪LWE
sage: LWE(10, 401, DiscreteGaussianDistributionIntegerSampler(3))()
((309, 347, 198, 194, 336, 360, 264, 123, 368, 398), 198)
```

class sage.crypto.lwe.**LindnerPeikert**(*n*, *delta=0.01*, *m=None*)

Bases: *sage.crypto.lwe.LWE*

LWE oracle with parameters as in [LP2011].

__init__(*n*, *delta=0.01*, *m=None*)

Construct LWE instance parameterised by security parameter `n` where the modulus `q` and the `stddev` of the noise is chosen as in [LP2011].

INPUT:

- `n` - security parameter (integer > 0)
- `delta` - error probability per symbol (default: 0.01)
- `m` - number of allowed samples or `None` in which case `m=2*n + 128` as in [LP2011] (default: `None`)

EXAMPLES:

```
sage: from sage.crypto.lwe import LindnerPeikert
sage: LindnerPeikert(n=20)
LWE(20, 2053, Discrete Gaussian sampler over the Integers with sigma = 3.
↪600954 and c = 0, 'noise', 168)
```

class sage.crypto.lwe.**Regev**(*n*, *secret_dist='uniform'*, *m=None*)

Bases: *sage.crypto.lwe.LWE*

LWE oracle with parameters as in [Reg09].

__init__(*n*, *secret_dist='uniform'*, *m=None*)

Construct LWE instance parameterised by security parameter `n` where the modulus `q` and the `stddev` of the noise are chosen as in [Reg09].

INPUT:

- `n` - security parameter (integer > 0)
- `secret_dist` - distribution of the secret. See documentation of *LWE* for details (default='uniform')
- `m` - number of allowed samples or `None` if no such limit exists (default: `None`)

EXAMPLES:

```
sage: from sage.crypto.lwe import Regev
sage: Regev(n=20)
LWE(20, 401, Discrete Gaussian sampler over the Integers with sigma = 1.
↪915069 and c = 401, 'uniform', None)
```

class sage.crypto.lwe.**RingLWE**(*N*, *q*, *D*, *poly=None*, *secret_dist='uniform'*, *m=None*)

Bases: sage.structure.sage_object.SageObject

Ring Learning with Errors oracle.

__init__(*N*, *q*, *D*, *poly=None*, *secret_dist='uniform'*, *m=None*)

Construct a Ring-LWE oracle in dimension `n=phi(N)` over a ring of order `q` with noise distribution `D`.

INPUT:

- `N` - index of cyclotomic polynomial (integer > 0, must be power of 2)
- `q` - modulus typically > N (integer > 0)
- `D` - an error distribution such as an instance of `DiscreteGaussianDistributionPolynomialSampler` or *UniformSampler*
- `poly` - a polynomial of degree `phi(N)`. If `None` the cyclotomic polynomial used (default: `None`).
- `secret_dist` - distribution of the secret. See documentation of *LWE* for details (default='uniform')

- m - number of allowed samples or None if no such limit exists (default: None)

EXAMPLES:

```
sage: from sage.crypto.lwe import RingLWE
sage: from sage.stats.distributions.discrete_gaussian_polynomial import␣
↪DiscreteGaussianDistributionPolynomialSampler
sage: D = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], n=euler_
↪phi(20), sigma=3.0)
sage: RingLWE(N=20, q=next_prime(800), D=D);
RingLWE(20, 809, Discrete Gaussian sampler for polynomials of degree < 8 with␣
↪σ=3.000000 in each component, x^8 - x^6 + x^4 - x^2 + 1, 'uniform', None)
```

__call__()

EXAMPLES:

```
sage: from sage.crypto.lwe import␣
↪DiscreteGaussianDistributionPolynomialSampler, RingLWE
sage: N = 16
sage: n = euler_phi(N)
sage: D = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], n, 5)
sage: ringlwe = RingLWE(N, 257, D, secret_dist='uniform')
sage: ringlwe()
((228, 149, 226, 198, 38, 222, 222, 127), (178, 132, 72, 147, 77, 159, 187,␣
↪250))
```

class sage.crypto.lwe.**RingLWEConverter**(*ringlwe*)

Bases: sage.structure.sage_object.SageObject

Wrapper callable to convert Ring-LWE oracles into LWE oracles by disregarding the additional structure.

__init__(*ringlwe*)

INPUT:

- ringlwe - an instance of a *RingLWE*

EXAMPLES:

```
sage: from sage.crypto.lwe import␣
↪DiscreteGaussianDistributionPolynomialSampler, RingLWE, RingLWEConverter
sage: D = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], euler_
↪phi(16), 5)
sage: lwe = RingLWEConverter(RingLWE(16, 257, D, secret_dist='uniform'))
sage: set_random_seed(1337)
sage: lwe()
((130, 32, 216, 3, 125, 58, 197, 171), 189)
```

__call__()

EXAMPLES:

```
sage: from sage.crypto.lwe import␣
↪DiscreteGaussianDistributionPolynomialSampler, RingLWE, RingLWEConverter
sage: D = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], euler_
↪phi(16), 5)
sage: lwe = RingLWEConverter(RingLWE(16, 257, D, secret_dist='uniform'))
sage: set_random_seed(1337)
sage: lwe()
((130, 32, 216, 3, 125, 58, 197, 171), 189)
```

class sage.crypto.lwe.**RingLindnerPeikert**(*N*, *delta=0.01*, *m=None*)
Bases: *sage.crypto.lwe.RingLWE*

Ring-LWE oracle with parameters as in [LP2011].

__init__(*N*, *delta=0.01*, *m=None*)
Construct a Ring-LWE oracle in dimension `n=phi(N)` where the modulus `q` and the `stddev` of the noise is chosen as in [LP2011].

INPUT:

- `N` - index of cyclotomic polynomial (integer > 0, must be power of 2)
- `delta` - error probability per symbol (default: 0.01)
- `m` - number of allowed samples or `None` in which case `3*n` is used (default: `None`)

EXAMPLES:

```
sage: from sage.crypto.lwe import RingLindnerPeikert
sage: RingLindnerPeikert(N=16)
RingLWE(16, 1031, Discrete Gaussian sampler for polynomials of degree < 8␣
↪with σ=2.803372 in each component, x^8 + 1, 'noise', 24)
```

class sage.crypto.lwe.**UniformNoiseLWE**(*n*, *instance='key'*, *m=None*)
Bases: *sage.crypto.lwe.LWE*

LWE oracle with uniform secret with parameters as in [CGW2013].

__init__(*n*, *instance='key'*, *m=None*)
Construct LWE instance parameterised by security parameter `n` where all other parameters are chosen as in [CGW2013].

INPUT:

- `n` - security parameter (integer >= 89)
- `instance` - one of
  - "key" - the LWE-instance that hides the secret key is generated
  - "encrypt" - the LWE-instance that hides the message is generated (default: `key`)
- `m` - number of allowed samples or `None` in which case `m` is chosen as in [CGW2013]. (default: `None`)

EXAMPLES:

```
sage: from sage.crypto.lwe import UniformNoiseLWE
sage: UniformNoiseLWE(89)
LWE(89, 154262477, UniformSampler(0, 351), 'noise', 131)

sage: UniformNoiseLWE(89, instance='encrypt')
LWE(131, 154262477, UniformSampler(0, 497), 'noise', 181)
```

class sage.crypto.lwe.**UniformPolynomialSampler**(*P*, *n*, *lower_bound*, *upper_bound*)
Bases: sage.structure.sage_object.SageObject

Uniform sampler for polynomials.

EXAMPLES:

```
sage: from sage.crypto.lwe import UniformPolynomialSampler
sage: UniformPolynomialSampler(ZZ['x'], 8, -2, 2)()
-2*x^7 + x^6 - 2*x^5 - x^3 - 2*x^2 - 2
```

__init__(*P*, *n*, *lower_bound*, *upper_bound*)

Construct a sampler for univariate polynomials of degree `n-1` where coefficients are drawn uniformly at random between `lower_bound` and `upper_bound` (both endpoints inclusive).

INPUT:

- `P` - a univariate polynomial ring over the Integers
- `n` - number of coefficients to be sampled
- `lower_bound` - integer
- `upper_bound` - integer

EXAMPLES:

```
sage: from sage.crypto.lwe import UniformPolynomialSampler
sage: UniformPolynomialSampler(ZZ['x'], 10, -10, 10)
UniformPolynomialSampler(10, -10, 10)
```

__call__()

Return a new sample.

EXAMPLES:

```
sage: from sage.crypto.lwe import UniformPolynomialSampler
sage: sampler = UniformPolynomialSampler(ZZ['x'], 8, -12, 12)
sage: sampler()
-10*x^7 + 5*x^6 - 8*x^5 + x^4 - 4*x^3 - 11*x^2 - 10
```

class sage.crypto.lwe.**UniformSampler**(*lower_bound*, *upper_bound*)

Bases: `sage.structure.sage_object.SageObject`

Uniform sampling in a range of integers.

EXAMPLES:

```
sage: from sage.crypto.lwe import UniformSampler
sage: sampler = UniformSampler(-2, 2); sampler
UniformSampler(-2, 2)
sage: sampler()
-2
```

__init__(*lower_bound*, *upper_bound*)

Construct a uniform sampler with bounds `lower_bound` and `upper_bound` (both endpoints inclusive).

INPUT:

- `lower_bound` - integer
- `upper_bound` - integer

EXAMPLES:

```
sage: from sage.crypto.lwe import UniformSampler
sage: UniformSampler(-2, 2)
UniformSampler(-2, 2)
```

__call__()

Return a new sample.

EXAMPLES:

```
sage: from sage.crypto.lwe import UniformSampler
sage: sampler = UniformSampler(-12, 12)
sage: sampler()
-10
```

sage.crypto.lwe.**balance_sample**(*s*, *q=None*)

Given `(a,c) = s` return a tuple `(a',c')` where `a'` is an integer vector with entries between -q//2 and q//2 and `c` is also within these bounds.

If `q` is given `(a,c) = s` may live in the integers. If `q` is not given, then `(a,c)` are assumed to live in $\mathbf{Z}/q\mathbf{Z}$.

INPUT:

- `s` - sample of the form (a,c) where a is a vector and c is a scalar
- `q` - modulus (default: `None`)

EXAMPLES:

```
sage: from sage.crypto.lwe import balance_sample, samples, Regev
sage: list(map(balance_sample, samples(10, 5, Regev)))
[((-9, -4, -4, 4, -4), 4), ((-8, 11, 12, -11, -11), -7),
...
((-11, 12, 0, -6, -3), 7), ((-7, 14, 8, 11, -8), -12)]


sage: from sage.crypto.lwe import balance_sample,␣
↪DiscreteGaussianDistributionPolynomialSampler, RingLWE, samples
sage: D = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 5)
sage: rlwe = RingLWE(20, 257, D)
sage: list(map(balance_sample, samples(10, 8, rlwe)))
[((-7, -37, -64, 107, -91, -24, 120, 54), (74, 83, 18, 55, -53, 43, 4, 10)),
...
((-63, 34, 82, -112, 49, 89, -72, -41), (117, 43, 13, -37, 102, 55, -97, 56))]
```

Note: This function is useful to convert between Sage's standard representation of elements in $\mathbf{Z}/q\mathbf{Z}$ as integers between 0 and q-1 and the usual representation of such elements in lattice cryptography as integers between -q//2 and q//2.

---

sage.crypto.lwe.**samples**(*m*, *n*, *lwe*, *seed=None*, *balanced=False*, ***kwds*)

Return `m` LWE samples.

INPUT:

- `m` - the number of samples (integer > 0)
- `n` - the security parameter (integer > 0)
- `lwe` - either
  - a subclass of *LWE* such as *Regev* or *LindnerPeikert*
  - an instance of *LWE* or any subclass
  - the name of any such class (e.g., "Regev", "LindnerPeikert")
- `seed` - seed to be used for generation or `None` if no specific seed shall be set (default: `None`)
- `balanced` - use function *balance_sample()* to return balanced representations of finite field elements (default: `False`)

- `**kwds` - passed through to LWE constructor

EXAMPLES:

```
sage: from sage.crypto.lwe import samples, Regev
sage: samples(2, 20, Regev, seed=1337)
[((199, 388, 337, 53, 200, 284, 336, 215, 75, 14, 274, 234, 97, 255, 246, 153,␣
↪268, 218, 396, 351), 15),
 ((365, 227, 333, 165, 76, 328, 288, 206, 286, 42, 175, 155, 190, 275, 114, 280,␣
↪45, 218, 304, 386), 143)]

sage: from sage.crypto.lwe import samples, Regev
sage: samples(2, 20, Regev, balanced=True, seed=1337)
[((199, -13, -64, 53, 200, -117, -65, -186, 75, 14, -127, -167, 97, -146, -155,␣
↪153, -133, -183, -5, -50), 15),
 ((-36, -174, -68, 165, 76, -73, -113, -195, -115, 42, 175, 155, 190, -126, 114, -
↪121, 45, -183, -97, -15), 143)]

sage: from sage.crypto.lwe import samples
sage: samples(2, 20, 'LindnerPeikert')
[((506, 1205, 398, 0, 337, 106, 836, 75, 1242, 642, 840, 262, 1823, 1798, 1831,␣
↪1658, 1084, 915, 1994, 163), 1447),
 ((463, 250, 1226, 1906, 330, 933, 1014, 1061, 1322, 2035, 1849, 285, 1993, 1975,␣
↪864, 1341, 41, 1955, 1818, 1357), 312)]
```

CHAPTER

NINETEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W