万兴PDF

looping_start and looping_end are currently only used for loop morphisms, those which have the same domain and codomain. These two attributes should store a valid Xy-pic direction and specify, correspondingly, the direction the arrow gets out into and the direction the arrow gets back from:

```
>>> astr = ArrowStringDescription(
... unit="mm", curving=None, curving_amount=None,
... looping_start="u", looping_end="l", horizontal_direction="",
... vertical_direction="", label_position="_", label="f")
>>> print(str(astr))
\ar@(u,l)[]_{f}
```

label_displacement controls how far the arrow label is from the ends of the arrow. For example, to position the arrow label near the arrow head, use ">":

```
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> astr.label_displacement = ">"
>>> print(str(astr))
\ar@/^12mm/[dr]_>{f}
```

Finally, arrow_style is used to specify the arrow style. To get a dashed arrow, for example, use "{->}" as arrow style:

```
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> astr.arrow_style = "{-->}"
>>> print(str(astr))
\ar@/^12mm/@{-->}[dr]_{f}
```

万兴PDF

Notes

Instances of *ArrowStringDescription* (page 2756) will be constructed by *XypicDiagramDrawer* (page 2758) and provided for further use in formatters. The user is not expected to construct instances of *ArrowStringDescription* (page 2756) themselves.

To be able to properly utilise this class, the reader is encouraged to checkout the Xy-pic user guide, available at [Xypic].

See also:

XypicDiagramDrawer (page 2758)

References

[Xypic]

class sympy.categories.diagram_drawing.**XypicDiagramDrawer**

Given a *Diagram* (page 2749) and the corresponding *DiagramGrid* (page 2752), produces the Xy-pic representation of the diagram.

The most important method in this class is draw. Consider the following triangle diagram:

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```

To draw this diagram, its objects need to be laid out with a *DiagramGrid* (page 2752):

```
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:

```
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C &
}
```

For further details see the docstring of this method.

To control the appearance of the arrows, formatters are used. The dictionary arrow_formatters maps morphisms to formatter functions. A formatter is accepts an *ArrowStringDescription* (page 2756) and is allowed to modify any of the arrow properties exposed thereby. For example, to have all morphisms with the property unique appear as dashed arrows, and to have their names prepended with ∃!, the following should be done:

```
>>> def formatter(astr):
...   astr.label = r"\exists !" + astr.label
...   astr.arrow_style = "{-->}"
>>> drawer.arrow_formatters["unique"] = formatter
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar@{-->}[d]_{\exists !g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C &
}
```

To modify the appearance of all arrows in the diagram, set default_arrow_formatter. For example, to place all morphism labels a little bit farther from the arrow head so that they look more centred, do as follows:

万兴PDF

```
>>> def default_formatter(astr):
...   astr.label_displacement = "(0.45)"
>>> drawer.default_arrow_formatter = default_formatter
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar@{-->}[d]_(0.45){\exists !g\circ f} \ar[r]^(0.45){f} & B \ar[ld]^(0.
↪45){g} \\
C &
}
```

In some diagrams some morphisms are drawn as curved arrows. Consider the following diagram:

```
>>> D = Object("D")
>>> E = Object("E")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])
>>> grid = DiagramGrid(diagram)
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} & B \ar[d]^{g} & D \ar[l]^{k} \ar@/_3mm/[ll]_{h} \\
& C &
}
```

To control how far the morphisms are curved by default, one can use the unit and default_curving_amount attributes:

万兴PDF

```
>>> drawer.unit = "cm"
>>> drawer.default_curving_amount = 1
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} & B \ar[d]^{g} & D \ar[l]^{k} \ar@/_1cm/[ll]_{h} \\
& C &
}
```

In some diagrams, there are multiple curved morphisms between the same two objects. To control by how much the curving changes between two such successive morphisms, use default_curving_step:

```
>>> drawer.default_curving_step = 1
>>> h1 = NamedMorphism(A, D, "h1")
>>> diagram = Diagram([f, g, h, k, h1])
>>> grid = DiagramGrid(diagram)
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} \ar@/^1cm/[rr]^{h_{1}} & B \ar[d]^{g} & D \ar[l]^{k} \ar@/_
↪2cm/[ll]_{h} \\
& C &
}
```

The default value of default_curving_step is 4 units.

万兴PDF

See also:

draw (page 2760), *ArrowStringDescription* (page 2756)

draw(*diagram, grid, masked=None, diagram_format=''*)

Returns the Xy-pic representation of diagram laid out in grid.

Consider the following simple triangle diagram.

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```

To draw this diagram, its objects need to be laid out with a *DiagramGrid* (page 2752):

```
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:

```
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C &
}
```

万兴PDF

The argument masked can be used to skip morphisms in the presentation of the diagram:

```
>>> print(drawer.draw(diagram, grid, masked=[g * f]))
\xymatrix{
A \ar[r]^{f} & B \ar[ld]^{g} \\
C &
}
```

Finally, the diagram_format argument can be used to specify the format string of the diagram. For example, to increase the spacing by 1 cm, proceeding as follows:

```
>>> print(drawer.draw(diagram, grid, diagram_format="@+1cm"))
\xymatrix@+1cm{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C &
}
```

sympy.categories.diagram_drawing.**xypic_draw_diagram**(*diagram, masked=None, diagram_format='', groups=None, **hints*)

Provides a shortcut combining *DiagramGrid* (page 2752) and *XypicDiagramDrawer* (page 2758). Returns an Xy-pic presentation of diagram. The argument masked is a list of morphisms which will be not be drawn. The argument diagram_format is the

万兴PDF

format string inserted after "xymatrix". `groups` should be a set of logical groups. The `hints` will be passed directly to the constructor of *DiagramGrid* (page 2752).

For more information about the arguments, see the docstrings of *DiagramGrid* (page 2752) and `XypicDiagramDrawer.draw`.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import xypic_draw_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
>>> print(xypic_draw_diagram(diagram))
\xymatrix{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C &
}
```

See also:

XypicDiagramDrawer (page 2758), *DiagramGrid* (page 2752)

万兴PDF

`sympy.categories.diagram_drawing.`**`preview_diagram`**(*diagram*, *masked=None*, *diagram_format=''*, *groups=None*, *output='png'*, *viewer=None*, *euler=True*, ***hints*)

Combines the functionality of `xypic_draw_diagram` and `sympy.printing.preview`. The arguments `masked`, `diagram_format`, `groups`, and `hints` are passed to `xypic_draw_diagram`, while `output`, `viewer, and ``euler` are passed to `preview`.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import preview_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> preview_diagram(d)
```

See also:

XypicDiagramDrawer (page 2758)

万兴PDF

Cryptography

Warning: This module is intended for educational purposes only. Do not use the functions in this module for real cryptographic applications. If you wish to encrypt real data, we recommend using something like the cryptography module.

Encryption is the process of hiding a message and a cipher is a means of doing so. Included in this module are both block and stream ciphers:

- Shift cipher
- Affine cipher
- substitution ciphers
- Vigenere's cipher
- Hill's cipher
- Bifid ciphers
- RSA
- Kid RSA
- linear-feedback shift registers (for stream ciphers)
- ElGamal encryption

万兴PDF

In a *substitution cipher* "units" (not necessarily single characters) of plaintext are replaced with ciphertext according to a regular system.

A *transposition cipher* is a method of encryption by which the positions held by "units" of plaintext are replaced by a permutation of the plaintext. That is, the order of the units is changed using a bijective function on the position of the characters to perform the encryption.

A *monoalphabetic cipher* uses fixed substitution over the entire message, whereas a *polyalphabetic cipher* uses a number of substitutions at different times in the message.

`sympy.crypto.crypto.`**`AZ`**`(`*s=None*`)`

Return the letters of `s` in uppercase. In case more than one string is passed, each of them will be processed and a list of upper case strings will be returned.

Examples

```
>>> from sympy.crypto.crypto import AZ
>>> AZ('Hello, world!')
'HELLOWORLD'
>>> AZ('Hello, world!'.split())
['HELLO', 'WORLD']
```

See also:

check_and_join (page 2763)

万兴PDF

sympy.crypto.crypto.**padded_key**(*key*, *symbols*)

Return a string of the distinct characters of symbols with those of key appearing first. A ValueError is raised if a) there are duplicate characters in symbols or b) there are characters in key that are not in symbols.

Examples

```
>>> from sympy.crypto.crypto import padded_key
>>> padded_key('PUPPY', 'OPQRSTUVWXY')
'PUYOQRSTVWX'
>>> padded_key('RSA', 'ARTIST')
Traceback (most recent call last):
...
ValueError: duplicate characters in symbols: T
```

sympy.crypto.crypto.**check_and_join**(*phrase*, *symbols=None*, *filter=None*)

Joins characters of phrase and if symbols is given, raises an error if any character in phrase is not in symbols.

Parameters

phrase

String or list of strings to be returned as a string.

symbols

Iterable of characters allowed in phrase.

If symbols is None, no checking is performed.

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import check_and_join
>>> check_and_join('a phrase')
'a phrase'
>>> check_and_join('a phrase'.upper().split())
'APHRASE'
>>> check_and_join('a phrase!'.upper().split(), 'ARE', filter=True)
'ARAE'
>>> check_and_join('a phrase!'.upper().split(), 'ARE')
Traceback (most recent call last):
...
ValueError: characters in phrase but not symbols: "!HPS"
```

sympy.crypto.crypto.**cycle_list**(*k*, *n*)

Returns the elements of the list range(n) shifted to the left by k (so the list starts with k (mod n)).

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import cycle_list
>>> cycle_list(3, 10)
[3, 4, 5, 6, 7, 8, 9, 0, 1, 2]
```

`sympy.crypto.crypto.`**`encipher_shift`**(*msg, key, symbols=None*)

Performs shift cipher encryption on plaintext msg, and returns the ciphertext.

Parameters

key : int

The secret key.

msg : str

Plaintext of upper-case letters.

Returns

str

Ciphertext of upper-case letters.

Examples

```
>>> from sympy.crypto.crypto import encipher_shift, decipher_shift
>>> msg = "GONAVYBEATARMY"
>>> ct = encipher_shift(msg, 1); ct
'HPOBWZCFBUBSNZ'
```

万兴PDF

To decipher the shifted text, change the sign of the key:

```
>>> encipher_shift(ct, -1)
'GONAVYBEATARMY'
```

There is also a convenience function that does this with the original key:

```
>>> decipher_shift(ct, 1)
'GONAVYBEATARMY'
```

Notes

ALGORITHM:

STEPS:

0. Number the letters of the alphabet from 0, …, N
1. Compute from the string `msg` a list `L1` of corresponding integers.
2. Compute from the list `L1` a new list `L2`, given by adding `(k mod 26)` to each element in `L1`.
3. Compute from the list `L2` a string `ct` of corresponding letters.

万兴PDF

The shift cipher is also called the Caesar cipher, after Julius Caesar, who, according to Suetonius, used it with a shift of three to protect messages of military significance. Caesar's nephew Augustus reportedly used a similar cipher, but with a right shift of 1.

See also:

decipher_shift (page 2765)

References

[R136], [R137]

sympy.crypto.crypto.**decipher_shift**(*msg, key, symbols=None*)

Return the text by shifting the characters of `msg` to the left by the amount given by `key`.

Examples

```
>>> from sympy.crypto.crypto import encipher_shift, decipher_shift
>>> msg = "GONAVYBEATARMY"
>>> ct = encipher_shift(msg, 1); ct
'HPOBWZCFBUBSNZ'
```

To decipher the shifted text, change the sign of the key:

```
>>> encipher_shift(ct, -1)
'GONAVYBEATARMY'
```

Or use this function with the original key:

```
>>> decipher_shift(ct, 1)
'GONAVYBEATARMY'
```

万兴PDF

sympy.crypto.crypto.**encipher_rot13**(*msg, symbols=None*)

Performs the ROT13 encryption on a given plaintext `msg`.

Explanation

ROT13 is a substitution cipher which substitutes each letter in the plaintext message for the letter furthest away from it in the English alphabet.

Equivalently, it is just a Caeser (shift) cipher with a shift key of 13 (midway point of the alphabet).

See also:

decipher_rot13 (page 2766), *encipher_shift* (page 2764)

万兴PDF

References

[R138]

sympy.crypto.crypto.**decipher_rot13**(*msg*, *symbols=None*)

Performs the ROT13 decryption on a given plaintext `msg`.

Explanation

`decipher_rot13` is equivalent to `encipher_rot13` as both `decipher_shift` with a key of 13 and `encipher_shift` key with a key of 13 will return the same results. Nonetheless, `decipher_rot13` has nonetheless been explicitly defined here for consistency.

Examples

```
>>> from sympy.crypto.crypto import encipher_rot13, decipher_rot13
>>> msg = 'GONAVYBEATARMY'
>>> ciphertext = encipher_rot13(msg);ciphertext
'TBANILORNGNEZL'
>>> decipher_rot13(ciphertext)
'GONAVYBEATARMY'
>>> encipher_rot13(msg) == decipher_rot13(msg)
True
>>> msg == decipher_rot13(ciphertext)
True
```

万兴PDF

sympy.crypto.crypto.**encipher_affine**(*msg*, *key*, *symbols=None*, *_inverse=False*)

Performs the affine cipher encryption on plaintext `msg`, and returns the ciphertext.

Parameters

msg : str

Characters that appear in `symbols`.

a, b : int, int

A pair integers, with `gcd(a, N) = 1` (the secret key).

symbols

String of characters (default = uppercase letters).

When no symbols are given, `msg` is converted to upper case letters and all other characters are ignored.

Returns

ct

String of characters (the ciphertext message)

万兴PDF

Explanation

Encryption is based on the map $x \rightarrow ax+b$ (mod N) where N is the number of characters in the alphabet. Decryption is based on the map $x \rightarrow cx + d$ (mod N), where $c = a^{-1}$ (mod N) and $d = -a^{-1}b$ (mod N). In particular, for the map to be invertible, we need gcd$(a, N) = 1$ and an error will be raised if this is not true.

Notes

ALGORITHM:

STEPS:

0. Number the letters of the alphabet from 0, …, N
1. Compute from the string `msg` a list `L1` of corresponding integers.
2. Compute from the list `L1` a new list `L2`, given by replacing `x` by `a*x + b (mod N)`, for each element `x` in `L1`.
3. Compute from the list `L2` a string `ct` of corresponding letters.

This is a straightforward generalization of the shift cipher with the added complexity of requiring 2 characters to be deciphered in order to recover the key.

See also:

decipher_affine (page 2767)

References

[R139]

万兴PDF

`sympy.crypto.crypto.`**`decipher_affine`**`(`*msg, key, symbols=None*`)`

Return the deciphered text that was made from the mapping, $x \rightarrow ax + b$ (mod N), where N is the number of characters in the alphabet. Deciphering is done by reciphering with a new key: $x \rightarrow cx + d$ (mod N), where $c = a^{-1}$ (mod N) and $d = -a^{-1}b$ (mod N).

Examples

```
>>> from sympy.crypto.crypto import encipher_affine, decipher_affine
>>> msg = "GO NAVY BEAT ARMY"
>>> key = (3, 1)
>>> encipher_affine(msg, key)
'TROBMVENBGBALV'
>>> decipher_affine(_, key)
'GONAVYBEATARMY'
```

See also:

encipher_affine (page 2766)

`sympy.crypto.crypto.`**`encipher_atbash`**`(`*msg, symbols=None*`)`

Enciphers a given `msg` into its Atbash ciphertext and returns it.

万兴PDF

Explanation

Atbash is a substitution cipher originally used to encrypt the Hebrew alphabet. Atbash works on the principle of mapping each alphabet to its reverse / counterpart (i.e. a would map to z, b to y etc.)

Atbash is functionally equivalent to the affine cipher with `a = 25` and `b = 25`

See also:

decipher_atbash (page 2768)

`sympy.crypto.crypto.`**`decipher_atbash`**`(`*msg, symbols=None*`)`

Deciphers a given `msg` using Atbash cipher and returns it.

Explanation

`decipher_atbash` is functionally equivalent to `encipher_atbash`. However, it has still been added as a separate function to maintain consistency.

Examples

```
>>> from sympy.crypto.crypto import encipher_atbash, decipher_atbash
>>> msg = 'GONAVYBEATARMY'
>>> encipher_atbash(msg)
'TLMZEBYVZGZINB'
>>> decipher_atbash(msg)
'TLMZEBYVZGZINB'
>>> encipher_atbash(msg) == decipher_atbash(msg)
True
>>> msg == encipher_atbash(encipher_atbash(msg))
True
```

万兴PDF

See also:

encipher_atbash (page 2767)

References

[R140]

`sympy.crypto.crypto.`**`encipher_substitution`**`(`*msg, old, new=None*`)`

Returns the ciphertext obtained by replacing each character that appears in `old` with the corresponding character in `new`. If `old` is a mapping, then `new` is ignored and the replacements defined by `old` are used.

万兴PDF

Explanation

This is a more general than the affine cipher in that the key can only be recovered by determining the mapping for each symbol. Though in practice, once a few symbols are recognized the mappings for other characters can be quickly guessed.

Examples

```
>>> from sympy.crypto.crypto import encipher_substitution, AZ
>>> old = 'OEYAG'
>>> new = '034^6'
>>> msg = AZ("go navy! beat army!")
>>> ct = encipher_substitution(msg, old, new); ct
'60N^V4B3^T^RM4'
```

To decrypt a substitution, reverse the last two arguments:

```
>>> encipher_substitution(ct, new, old)
'GONAVYBEATARMY'
```

In the special case where `old` and `new` are a permutation of order 2 (representing a transposition of characters) their order is immaterial:

```
>>> old = 'NAVY'
>>> new = 'ANYV'
>>> encipher = lambda x: encipher_substitution(x, old, new)
>>> encipher('NAVY')
'ANYV'
>>> encipher(_)
'NAVY'
```

万兴PDF

The substitution cipher, in general, is a method whereby "units" (not necessarily single characters) of plaintext are replaced with ciphertext according to a regular system.

```
>>> ords = dict(zip('abc', ['\\%i' % ord(i) for i in 'abc']))
>>> print(encipher_substitution('abc', ords))
\97\98\99
```

References

[R141]

`sympy.crypto.crypto.`**`encipher_vigenere`**`(`*msg, key, symbols=None*`)`

Performs the Vigenere cipher encryption on plaintext `msg`, and returns the ciphertext.

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import encipher_vigenere, AZ
>>> key = "encrypt"
>>> msg = "meet me on monday"
>>> encipher_vigenere(msg, key)
'QRGKKTHRZQEBPR'
```

Section 1 of the Kryptos sculpture at the CIA headquarters uses this cipher and also changes the order of the alphabet [R143]. Here is the first line of that section of the sculpture:

```
>>> from sympy.crypto.crypto import decipher_vigenere, padded_key
>>> alp = padded_key('KRYPTOS', AZ())
>>> key = 'PALIMPSEST'
>>> msg = 'EMUFPHZLRFAXYUSDJKZLDKRNSHGNFIVJ'
>>> decipher_vigenere(msg, key, alp)
'BETWEENSUBTLESHADINGANDTHEABSENC'
```

Explanation

The Vigenere cipher is named after Blaise de Vigenere, a sixteenth century diplomat and cryptographer, by a historical accident. Vigenere actually invented a different and more complicated cipher. The so-called *Vigenere cipher* was actually invented by Giovan Batista Belaso in 1553.

This cipher was used in the 1800's, for example, during the American Civil War. The Confederacy used a brass cipher disk to implement the Vigenere cipher (now on display in the NSA Museum in Fort Meade) [R142].

The Vigenere cipher is a generalization of the shift cipher. Whereas the shift cipher shifts each letter by the same amount (that amount being the key of the shift cipher) the Vigenere cipher shifts a letter by an amount determined by the key (which is a word or phrase known only to the sender and receiver).

For example, if the key was a single letter, such as "C", then the so-called Vigenere cipher is actually a shift cipher with a shift of 2 (since "C" is the 2nd letter of the alphabet, if you start counting at 0). If the key was a word with two letters, such as "CA", then the so-called Vigenere cipher will shift letters in even positions by 2 and letters in odd positions are left alone (shifted by 0, since "A" is the 0th letter, if you start counting at 0).

ALGORITHM:

INPUT:

`msg`: string of characters that appear in `symbols` (the plaintext)

`key`: a string of characters that appear in `symbols` (the secret key)

`symbols`: a string of letters defining the alphabet

OUTPUT:

`ct`: string of characters (the ciphertext message)

STEPS:

0. Number the letters of the alphabet from 0, …, N

万兴PDF

1. Compute from the string `key` a list `L1` of corresponding integers. Let `n1 = len(L1)`.
2. Compute from the string `msg` a list `L2` of corresponding integers. Let `n2 = len(L2)`.
3. Break `L2` up sequentially into sublists of size `n1`; the last sublist may be smaller than `n1`
4. For each of these sublists `L` of `L2`, compute a new list `C` given by `C[i] = L[i] + L1[i] (mod N)` to the `i`-th element in the sublist, for each `i`.
5. Assemble these lists `C` by concatenation into a new list of length `n2`.
6. Compute from the new list a string `ct` of corresponding letters.

Once it is known that the key is, say, n characters long, frequency analysis can be applied to every n-th letter of the ciphertext to determine the plaintext. This method is called *Kasiski examination* (although it was first discovered by Babbage). If they key is as long as the message and is comprised of randomly selected characters – a one-time pad – the message is theoretically unbreakable.

The cipher Vigenere actually discovered is an "auto-key" cipher described as follows.

ALGORITHM:

INPUT:

`key`: a string of letters (the secret key)

`msg`: string of letters (the plaintext message)

OUTPUT:

`ct`: string of upper-case letters (the ciphertext message)

万兴PDF

STEPS:

0. Number the letters of the alphabet from 0, …, N
1. Compute from the string `msg` a list `L2` of corresponding integers. Let `n2 = len(L2)`.
2. Let `n1` be the length of the key. Append to the string `key` the first `n2 - n1` characters of the plaintext message. Compute from this string (also of length `n2`) a list `L1` of integers corresponding to the letter numbers in the first step.
3. Compute a new list `C` given by `C[i] = L1[i] + L2[i] (mod N)`.
4. Compute from the new list a string `ct` of letters corresponding to the new integers.

To decipher the auto-key ciphertext, the key is used to decipher the first `n1` characters and then those characters become the key to decipher the next `n1` characters, etc…:

```
>>> m = AZ('go navy, beat army! yes you can'); m
'GONAVYBEATARMYYESYOUCAN'
>>> key = AZ('gold bug'); n1 = len(key); n2 = len(m)
>>> auto_key = key + m[:n2 - n1]; auto_key
'GOLDBUGGONAVYBEATARMYYE'
>>> ct = encipher_vigenere(m, auto_key); ct
```

(continues on next page)

万兴PDF

(continued from previous page)

```
'MCYDWSHKOGAMKZCELYFGAYR'
>>> n1 = len(key)
>>> pt = []
>>> while ct:
...     part, ct = ct[:n1], ct[n1:]
...     pt.append(decipher_vigenere(part, key))
...     key = pt[-1]
...
>>> ''.join(pt) == m
True
```

References

[R142], [R143], [R144]

`sympy.crypto.crypto.`**`decipher_vigenere`**(*msg*, *key*, *symbols=None*)

Decode using the Vigenere cipher.

Examples

```
>>> from sympy.crypto.crypto import decipher_vigenere
>>> key = "encrypt"
>>> ct = "QRGK kt HRZQE BPR"
>>> decipher_vigenere(ct, key)
'MEETMEONMONDAY'
```

万兴PDF

`sympy.crypto.crypto.`**`encipher_hill`**(*msg*, *key*, *symbols=None*, *pad='Q'*)

Return the Hill cipher encryption of `msg`.

Parameters

msg

Plaintext message of n upper-case letters.

key

A $k \times k$ invertible matrix K, all of whose entries are in Z_{26} (or whatever number of symbols are being used).

pad

Character (default "Q") to use to make length of text be a multiple of k.

Returns

ct

Ciphertext of upper-case letters.

万兴PDF

Explanation

The Hill cipher [R145], invented by Lester S. Hill in the 1920's [R146], was the first polygraphic cipher in which it was practical (though barely) to operate on more than three symbols at once. The following discussion assumes an elementary knowledge of matrices.

First, each letter is first encoded as a number starting with 0. Suppose your message msg consists of n capital letters, with no spaces. This may be regarded an n-tuple M of elements of Z_{26} (if the letters are those of the English alphabet). A key in the Hill cipher is a kxk matrix K, all of whose entries are in Z_{26}, such that the matrix K is invertible (i.e., the linear transformation $K : Z_N^k \rightarrow Z_N^k$ is one-to-one).

Notes

ALGORITHM:

STEPS:

0. Number the letters of the alphabet from 0, …, N
1. Compute from the string `msg` a list `L` of corresponding integers. Let `n = len(L)`.
2. Break the list `L` up into `t = ceiling(n/k)` sublists `L_1`, …, `L_t` of size `k` (with the last list "padded" to ensure its size is `k`).
3. Compute new list `C_1`, …, `C_t` given by `C[i] = K*L_i` (arithmetic is done mod N), for each `i`.
4. Concatenate these into a list `C = C_1 + ... + C_t`.
5. Compute from `C` a string `ct` of corresponding letters. This has length `k*t`.

See also:

decipher_hill (page 2773)

References

[R145], [R146]

`sympy.crypto.crypto.`**`decipher_hill`**`(`*msg, key, symbols=None*`)`

Deciphering is the same as enciphering but using the inverse of the key matrix.

Examples

```
>>> from sympy.crypto.crypto import encipher_hill, decipher_hill
>>> from sympy import Matrix
```

```
>>> key = Matrix([[1, 2], [3, 5]])
>>> encipher_hill("meet me on monday", key)
'UEQDUEODOCTCWQ'
```

(continues on next page)

万兴PDF

万兴PDF

(continued from previous page)

```
>>> decipher_hill(_, key)
'MEETMEONMONDAY'
```

When the length of the plaintext (stripped of invalid characters) is not a multiple of the key dimension, extra characters will appear at the end of the enciphered and deciphered text. In order to decipher the text, those characters must be included in the text to be deciphered. In the following, the key has a dimension of 4 but the text is 2 short of being a multiple of 4 so two characters will be added.

```
>>> key = Matrix([[1, 1, 1, 2], [0, 1, 1, 0],
...               [2, 2, 3, 4], [1, 1, 0, 1]])
>>> msg = "ST"
>>> encipher_hill(msg, key)
'HJEB'
>>> decipher_hill(_, key)
'STQQ'
>>> encipher_hill(msg, key, pad="Z")
'ISPK'
>>> decipher_hill(_, key)
'STZZ'
```

If the last two characters of the ciphertext were ignored in either case, the wrong plaintext would be recovered:

```
>>> decipher_hill("HD", key)
'ORMV'
>>> decipher_hill("IS", key)
'UIKY'
```

万兴PDF

See also:

encipher_hill (page 2772)

sympy.crypto.crypto.**encipher_bifid**(*msg*, *key*, *symbols=None*)

Performs the Bifid cipher encryption on plaintext `msg`, and returns the ciphertext.

This is the version of the Bifid cipher that uses an $n \times n$ Polybius square.

Parameters

msg

Plaintext string.

key

Short string for key.

Duplicate characters are ignored and then it is padded with the characters in `symbols` that were not in the short key.

symbols

$n \times n$ characters defining the alphabet.

(default is string.printable)

Returns

ciphertext

万兴PDF

Ciphertext using Bifid5 cipher without spaces.

See also:

decipher_bifid (page 2775), *encipher_bifid5* (page 2776), *encipher_bifid6* (page 2778)

References

[R147]

`sympy.crypto.crypto.`**`decipher_bifid`**(*msg*, *key*, *symbols=None*)

Performs the Bifid cipher decryption on ciphertext `msg`, and returns the plaintext.

This is the version of the Bifid cipher that uses the $n \times n$ Polybius square.

Parameters

msg

Ciphertext string.

key

Short string for key.

Duplicate characters are ignored and then it is padded with the characters in symbols that were not in the short key.

symbols

$n \times n$ characters defining the alphabet.

(default=string.printable, a 10×10 matrix)

Returns

deciphered

Deciphered text.

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import (
...     encipher_bifid, decipher_bifid, AZ)
```

Do an encryption using the bifid5 alphabet:

```
>>> alp = AZ().replace('J', '')
>>> ct = AZ("meet me on monday!")
>>> key = AZ("gold bug")
>>> encipher_bifid(ct, key, alp)
'IEILHHFSTSFQYE'
```

When entering the text or ciphertext, spaces are ignored so it can be formatted as desired. Re-entering the ciphertext from the preceding, putting 4 characters per line and padding with an extra J, does not cause problems for the deciphering:

```
>>> decipher_bifid('''
... IEILH
... HFSTS
... FQYEJ''', key, alp)
'MEETMEONMONDAY'
```

When no alphabet is given, all 100 printable characters will be used:

```
>>> key = ''
>>> encipher_bifid('hello world!', key)
'bmtwmg-bIo*w'
>>> decipher_bifid(_, key)
'hello world!'
```

If the key is changed, a different encryption is obtained:

```
>>> key = 'gold bug'
>>> encipher_bifid('hello world!', 'gold_bug')
'hg2sfuei7t}w'
```

And if the key used to decrypt the message is not exact, the original text will not be perfectly obtained:

```
>>> decipher_bifid(_, 'gold pug')
'heldo~wor6d!'
```

万兴PDF

sympy.crypto.crypto.**bifid5_square**(*key=None*)

5x5 Polybius square.

Produce the Polybius square for the 5×5 Bifid cipher.

Examples

```
>>> from sympy.crypto.crypto import bifid5_square
>>> bifid5_square("gold bug")
Matrix([
[G, O, L, D, B],
[U, A, C, E, F],
[H, I, K, M, N],
[P, Q, R, S, T],
[V, W, X, Y, Z]])
```

sympy.crypto.crypto.**encipher_bifid5**(*msg, key*)

Performs the Bifid cipher encryption on plaintext msg, and returns the ciphertext.

Parameters

msg : str

Plaintext string.

Converted to upper case and filtered of anything but all letters except J.

key

万兴PDF

Short string for key; non-alphabetic letters, J and duplicated characters are ignored and then, if the length is less than 25 characters, it is padded with other letters of the alphabet (in alphabetical order).

Returns

ct

Ciphertext (all caps, no spaces).

Explanation

This is the version of the Bifid cipher that uses the 5×5 Polybius square. The letter “J” is ignored so it must be replaced with something else (traditionally an “I”) before encryption.

ALGORITHM: (5x5 case)

STEPS:

0. Create the 5×5 Polybius square `S` associated to `key` as follows:

   a) moving from left-to-right, top-to-bottom, place the letters of the key into a 5×5 matrix,

   b) if the key has less than 25 letters, add the letters of the alphabet not in the key until the 5×5 square is filled.

1. Create a list `P` of pairs of numbers which are the coordinates in the Polybius square of the letters in `msg`.
2. Let `L1` be the list of all first coordinates of `P` (length of `L1 = n`), let `L2` be the list of all second coordinates of `P` (so the length of `L2` is also `n`).
3. Let `L` be the concatenation of `L1` and `L2` (length `L = 2*n`), except that consecutive numbers are paired `(L[2*i], L[2*i + 1])`. You can regard `L` as a list of pairs of length `n`.
4. Let `C` be the list of all letters which are of the form `S[i, j]`, for all `(i, j)` in `L`. As a string, this is the ciphertext of `msg`.

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import (
...     encipher_bifid5, decipher_bifid5)
```

“J” will be omitted unless it is replaced with something else:

```
>>> round_trip = lambda m, k: \
...     decipher_bifid5(encipher_bifid5(m, k), k)
>>> key = 'a'
>>> msg = "JOSIE"
>>> round_trip(msg, key)
'OSIE'
>>> round_trip(msg.replace("J", "I"), key)
'IOSIE'
>>> j = "QIQ"
```

(continues on next page)

万兴PDF

(continued from previous page)

```
>>> round_trip(msg.replace("J", j), key).replace(j, "J")
'JOSIE'
```

Notes

The Bifid cipher was invented around 1901 by Felix Delastelle. It is a *fractional substitution* cipher, where letters are replaced by pairs of symbols from a smaller alphabet. The cipher uses a 5×5 square filled with some ordering of the alphabet, except that "J" is replaced with "I" (this is a so-called Polybius square; there is a 6×6 analog if you add back in "J" and also append onto the usual 26 letter alphabet, the digits 0, 1, ..., 9). According to Helen Gaines' book *Cryptanalysis*, this type of cipher was used in the field by the German Army during World War I.

See also:

decipher_bifid5 (page 2778), *encipher_bifid* (page 2774)

sympy.crypto.crypto.**decipher_bifid5**(*msg*, *key*)

Return the Bifid cipher decryption of msg.

Parameters

msg

Ciphertext string.

key

Short string for key; duplicated characters are ignored and if the length is less then 25 characters, it will be padded with other letters from the alphabet omitting "J". Non-alphabetic characters are ignored.

Returns

plaintext

Plaintext from Bifid5 cipher (all caps, no spaces).

Explanation

This is the version of the Bifid cipher that uses the 5×5 Polybius square; the letter "J" is ignored unless a key of length 25 is used.

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import encipher_bifid5, decipher_bifid5
>>> key = "gold bug"
>>> encipher_bifid5('meet me on friday', key)
'IEILEHFSTSFXEE'
>>> encipher_bifid5('meet me on monday', key)
'IEILHHFSTSFQYE'
>>> decipher_bifid5(_, key)
'MEETMEONMONDAY'
```

万兴PDF

`sympy.crypto.crypto.`**`encipher_bifid6`**(*msg*, *key*)

Performs the Bifid cipher encryption on plaintext `msg`, and returns the ciphertext.

This is the version of the Bifid cipher that uses the 6×6 Polybius square.

Parameters

msg

Plaintext string (digits okay).

key

Short string for key (digits okay).

If `key` is less than 36 characters long, the square will be filled with letters A through Z and digits 0 through 9.

Returns

ciphertext

Ciphertext from Bifid cipher (all caps, no spaces).

See also:

`decipher_bifid6` (page 2779), *`encipher_bifid`* (page 2774)

`sympy.crypto.crypto.`**`decipher_bifid6`**(*msg*, *key*)

Performs the Bifid cipher decryption on ciphertext `msg`, and returns the plaintext.

This is the version of the Bifid cipher that uses the 6×6 Polybius square.

Parameters

msg

Ciphertext string (digits okay); converted to upper case

key

Short string for key (digits okay).

If `key` is less than 36 characters long, the square will be filled with letters A through Z and digits 0 through 9. All letters are converted to uppercase.

Returns

plaintext

Plaintext from Bifid cipher (all caps, no spaces).

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import encipher_bifid6, decipher_bifid6
>>> key = "gold bug"
>>> encipher_bifid6('meet me on monday at 8am', key)
'KFKLJJHF5MMMKTFRGPL'
>>> decipher_bifid6(_, key)
'MEETMEONMONDAYAT8AM'
```

万兴PDF

sympy.crypto.crypto.**bifid6_square**(*key=None*)

6x6 Polybius square.

Produces the Polybius square for the 6×6 Bifid cipher. Assumes alphabet of symbols is "A", …, "Z", "0", …, "9".

Examples

```
>>> from sympy.crypto.crypto import bifid6_square
>>> key = "gold bug"
>>> bifid6_square(key)
Matrix([
[G, O, L, D, B, U],
[A, C, E, F, H, I],
[J, K, M, N, P, Q],
[R, S, T, V, W, X],
[Y, Z, 0, 1, 2, 3],
[4, 5, 6, 7, 8, 9]])
```

sympy.crypto.crypto.**rsa_public_key**(**args*, ***kwargs*)

Return the RSA *public key* pair, (n, e)

Parameters

args : naturals

If specified as p, q, e where p and q are distinct primes and e is a desired public exponent of the RSA, $n = pq$ and e will be verified against the totient $\phi(n)$ (Euler totient) or $\lambda(n)$ (Carmichael totient) to be gcd$(e, \phi(n)) = 1$ or gcd$(e, \lambda(n)) = 1$.

万兴PDF

If specified as $p_1, p_2, \ldots, p_n, e$ where $p_1, p_2, \ldots, p_n$ are specified as primes, and e is specified as a desired public exponent of the RSA, it will be able to form a multi-prime RSA, which is a more generalized form of the popular 2-prime RSA.

It can also be possible to form a single-prime RSA by specifying the argument as p, e, which can be considered a trivial case of a multi-prime RSA.

Furthermore, it can be possible to form a multi-power RSA by specifying two or more pairs of the primes to be same. However, unlike the two-distinct prime RSA or multi-prime RSA, not every numbers in the complete residue system $(\mathbb{Z}_n)$ will be decryptable since the mapping $\mathbb{Z}_n \to \mathbb{Z}_n$ will not be bijective. (Only except for the trivial case when $e = 1$ or more generally,

$$e \in \{1 + k\lambda(n) \mid k \in \mathbb{Z} \wedge k \geq 0\}$$

when RSA reduces to the identity.) However, the RSA can still be decryptable for the numbers in the reduced residue system $(\mathbb{Z}_n^\times)$, since the mapping $\mathbb{Z}_n^\times \to \mathbb{Z}_n^\times$ can still be bijective.

If you pass a non-prime integer to the arguments $p_1, p_2, \ldots, p_n$, the particular number will be prime-factored and it will become either a multi-prime RSA or a multi-power RSA in its canonical form, depending on whether the product equals its radical or not. $p_1 p_2 \ldots p_n =$ rad$(p_1 p_2 \ldots p_n)$

万兴PDF

totient : bool, optional

If `'Euler'`, it uses Euler's totient $\phi(n)$ which is *sympy.ntheory.factor_.totient()* (page 1501) in SymPy.

If `'Carmichael'`, it uses Carmichael's totient $\lambda(n)$ which is *sympy.ntheory.factor_.reduced_totient()* (page 1501) in SymPy.

Unlike private key generation, this is a trivial keyword for public key generation because $\gcd(e, \phi(n)) = 1 \iff \gcd(e, \lambda(n)) = 1$.

index : nonnegative integer, optional

Returns an arbitrary solution of a RSA public key at the index specified at $0, 1, 2, \ldots$. This parameter needs to be specified along with `totient='Carmichael'`.

Similarly to the non-uniquenss of a RSA private key as described in the `index` parameter documentation in *rsa_private_key()* (page 2782), RSA public key is also not unique and there is an infinite number of RSA public exponents which can behave in the same manner.

From any given RSA public exponent e, there are can be an another RSA public exponent $e + k\lambda(n)$ where k is an integer, λ is a Carmichael's totient function.

However, considering only the positive cases, there can be a principal solution of a RSA public exponent e_0 in $0 < e_0 < \lambda(n)$, and all the other solutions can be canonicalzed in a form of $e_0 + k\lambda(n)$.

`index` specifies the k notation to yield any possible value an RSA public key can have.

An example of computing any arbitrary RSA public key:

```
>>> from sympy.crypto.crypto import rsa_public_key
>>> rsa_public_key(61, 53, 17, totient='Carmichael',␣
↪index=0)
(3233, 17)
>>> rsa_public_key(61, 53, 17, totient='Carmichael',␣
↪index=1)
(3233, 797)
>>> rsa_public_key(61, 53, 17, totient='Carmichael',␣
↪index=2)
(3233, 1577)
```

multipower : bool, optional

Any pair of non-distinct primes found in the RSA specification will restrict the domain of the cryptosystem, as noted in the explanation of the parameter `args`.

SymPy RSA key generator may give a warning before dispatching it as a multi-power RSA, however, you can disable the warning if you pass `True` to this keyword.

Returns

(n, e) : int, int

万兴PDF

n is a product of any arbitrary number of primes given as the argument.

e is relatively prime (coprime) to the Euler totient $\phi(n)$.

False

Returned if less than two arguments are given, or e is not relatively prime to the modulus.

Examples

```
>>> from sympy.crypto.crypto import rsa_public_key
```

A public key of a two-prime RSA:

```
>>> p, q, e = 3, 5, 7
>>> rsa_public_key(p, q, e)
(15, 7)
>>> rsa_public_key(p, q, 30)
False
```

A public key of a multiprime RSA:

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> e = 7
>>> args = primes + [e]
>>> rsa_public_key(*args)
(30030, 7)
```

万兴PDF

Notes

Although the RSA can be generalized over any modulus n, using two large primes had became the most popular specification because a product of two large primes is usually the hardest to factor relatively to the digits of n can have.

However, it may need further understanding of the time complexities of each prime-factoring algorithms to verify the claim.

See also:

rsa_private_key (page 2782), *encipher_rsa* (page 2784), *decipher_rsa* (page 2785)

References

[R148], [R149], [R150], [R151]

sympy.crypto.crypto.**rsa_private_key**(**args*, ***kwargs*)

Return the RSA *private key* pair, (n, d)

Parameters

args : naturals

The keyword is identical to the args in *rsa_public_key()* (page 2780).

万兴PDF

totient : bool, optional

If `'Euler'`, it uses Euler's totient convention $\phi(n)$ which is *sympy.ntheory.factor_.totient()* (page 1501) in SymPy.

If `'Carmichael'`, it uses Carmichael's totient convention $\lambda(n)$ which is *sympy.ntheory.factor_.reduced_totient()* (page 1501) in SymPy.

There can be some output differences for private key generation as examples below.

Example using Euler's totient:

```
>>> from sympy.crypto.crypto import rsa_private_key
>>> rsa_private_key(61, 53, 17, totient='Euler')
(3233, 2753)
```

Example using Carmichael's totient:

```
>>> from sympy.crypto.crypto import rsa_private_key
>>> rsa_private_key(61, 53, 17, totient='Carmichael')
(3233, 413)
```

index : nonnegative integer, optional

Returns an arbitrary solution of a RSA private key at the index specified at $0, 1, 2, \ldots$. This parameter needs to be specified along with `totient='Carmichael'`.

RSA private exponent is a non-unique solution of $ed \mod \lambda(n) = 1$ and it is possible in any form of $d + k\lambda(n)$, where d is an another already-computed private exponent, and λ is a Carmichael's totient function, and k is any integer.

However, considering only the positive cases, there can be a principal solution of a RSA private exponent d_0 in $0 < d_0 < \lambda(n)$, and all the other solutions can be canonicalzed in a form of $d_0 + k\lambda(n)$.

`index` specifies the k notation to yield any possible value an RSA private key can have.

An example of computing any arbitrary RSA private key:

```
>>> from sympy.crypto.crypto import rsa_private_key
>>> rsa_private_key(61, 53, 17, totient='Carmichael', 
↪index=0)
(3233, 413)
>>> rsa_private_key(61, 53, 17, totient='Carmichael', 
↪index=1)
(3233, 1193)
>>> rsa_private_key(61, 53, 17, totient='Carmichael', 
↪index=2)
(3233, 1973)
```

multipower : bool, optional

The keyword is identical to the `multipower` in *rsa_public_key()* (page 2780).

万兴PDF

Returns

(n, d) : int, int

n is a product of any arbitrary number of primes given as the argument.

d is the inverse of e (mod $\phi(n)$) where e is the exponent given, and ϕ is a Euler totient.

False

Returned if less than two arguments are given, or e is not relatively prime to the totient of the modulus.

Examples

```
>>> from sympy.crypto.crypto import rsa_private_key
```

A private key of a two-prime RSA:

```
>>> p, q, e = 3, 5, 7
>>> rsa_private_key(p, q, e)
(15, 7)
>>> rsa_private_key(p, q, 30)
False
```

A private key of a multiprime RSA:

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> e = 7
>>> args = primes + [e]
>>> rsa_private_key(*args)
(30030, 823)
```

万兴PDF

See also:

rsa_public_key (page 2780), *encipher_rsa* (page 2784), *decipher_rsa* (page 2785)

References

[R152], [R153], [R154], [R155]

sympy.crypto.crypto.**encipher_rsa**(*i*, *key*, *factors=None*)

Encrypt the plaintext with RSA.

Parameters

i : integer

The plaintext to be encrypted for.

key : (n, e) where n, e are integers

n is the modulus of the key and e is the exponent of the key. The encryption is computed by i^e mod n.

The key can either be a public key or a private key, however, the message encrypted by a public key can only be decrypted by a private key, and vice versa, as RSA is an asymmetric cryptography system.

万兴PDF

factors : list of coprime integers

This is identical to the keyword `factors` in *decipher_rsa()* (page 2785).

Notes

Some specifications may make the RSA not cryptographically meaningful.

For example, 0, 1 will remain always same after taking any number of exponentiation, thus, should be avoided.

Furthermore, if $i^e < n$, i may easily be figured out by taking e th root.

And also, specifying the exponent as 1 or in more generalized form as $1 + k\lambda(n)$ where k is an nonnegative integer, λ is a carmichael totient, the RSA becomes an identity mapping.

Examples

```
>>> from sympy.crypto.crypto import encipher_rsa
>>> from sympy.crypto.crypto import rsa_public_key, rsa_private_key
```

Public Key Encryption:

```
>>> p, q, e = 3, 5, 7
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> encipher_rsa(msg, puk)
3
```

万兴PDF

Private Key Encryption:

```
>>> p, q, e = 3, 5, 7
>>> prk = rsa_private_key(p, q, e)
>>> msg = 12
>>> encipher_rsa(msg, prk)
3
```

Encryption using chinese remainder theorem:

```
>>> encipher_rsa(msg, prk, factors=[p, q])
3
```

`sympy.crypto.crypto.`**`decipher_rsa`**(*i*, *key*, *factors=None*)

Decrypt the ciphertext with RSA.

Parameters

i : integer

The ciphertext to be decrypted for.

key : (n, d) where n, d are integers

n is the modulus of the key and d is the exponent of the key. The decryption is computed by $i^d \bmod n$.

万兴PDF

The key can either be a public key or a private key, however, the message encrypted by a public key can only be decrypted by a private key, and vice versa, as RSA is an asymmetric cryptography system.

factors : list of coprime integers

As the modulus n created from RSA key generation is composed of arbitrary prime factors $n = {p_1}^{k_1}{p_2}^{k_2}\dots{p_n}^{k_n}$ where $p_1, p_2, \dots, p_n$ are distinct primes and $k_1, k_2, \dots, k_n$ are positive integers, chinese remainder theorem can be used to compute $i^d \bmod n$ from the fragmented modulo operations like

$$i^d \bmod {p_1}^{k_1}, i^d \bmod {p_2}^{k_2}, \dots, i^d \bmod {p_n}^{k_n}$$

or like

$$i^d \bmod {p_1}^{k_1}{p_2}^{k_2}, i^d \bmod {p_3}^{k_3}, \dots, i^d \bmod {p_n}^{k_n}$$

as long as every moduli does not share any common divisor each other.

The raw primes used in generating the RSA key pair can be a good option.

Note that the speed advantage of using this is only viable for very large cases (Like 2048-bit RSA keys) since the overhead of using pure Python implementation of *sympy.ntheory.modular.crt()* (page 1509) may overcompensate the theoritical speed advantage.

万兴PDF

Notes

See the Notes section in the documentation of *encipher_rsa()* (page 2784)

Examples

```
>>> from sympy.crypto.crypto import decipher_rsa, encipher_rsa
>>> from sympy.crypto.crypto import rsa_public_key, rsa_private_key
```

Public Key Encryption and Decryption:

```
>>> p, q, e = 3, 5, 7
>>> prk = rsa_private_key(p, q, e)
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> new_msg = encipher_rsa(msg, prk)
>>> new_msg
3
>>> decipher_rsa(new_msg, puk)
12
```

Private Key Encryption and Decryption:

万兴PDF

```
>>> p, q, e = 3, 5, 7
>>> prk = rsa_private_key(p, q, e)
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> new_msg = encipher_rsa(msg, puk)
>>> new_msg
3
>>> decipher_rsa(new_msg, prk)
12
```

Decryption using chinese remainder theorem:

```
>>> decipher_rsa(new_msg, prk, factors=[p, q])
12
```

See also:

encipher_rsa (page 2784)

sympy.crypto.crypto.**kid_rsa_public_key**(*a*, *b*, *A*, *B*)

Kid RSA is a version of RSA useful to teach grade school children since it does not involve exponentiation.

Explanation

Alice wants to talk to Bob. Bob generates keys as follows. Key generation:

- Select positive integers a, b, A, B at random.
- Compute $M = ab - 1$, $e = AM + a$, $d = BM + b$, $n = (ed - 1)//M$.
- The *public key* is (n, e). Bob sends these to Alice.
- The *private key* is (n, d), which Bob keeps secret.

万兴PDF

Encryption: If p is the plaintext message then the ciphertext is $c = pe \pmod n$.

Decryption: If c is the ciphertext message then the plaintext is $p = cd \pmod n$.

Examples

```
>>> from sympy.crypto.crypto import kid_rsa_public_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_public_key(a, b, A, B)
(369, 58)
```

sympy.crypto.crypto.**kid_rsa_private_key**(*a*, *b*, *A*, *B*)

Compute $M = ab - 1$, $e = AM + a$, $d = BM + b$, $n = (ed - 1)/M$. The *private key* is d, which Bob keeps secret.

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import kid_rsa_private_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_private_key(a, b, A, B)
(369, 70)
```

sympy.crypto.crypto.**encipher_kid_rsa**(*msg, key*)

Here msg is the plaintext and key is the public key.

Examples

```
>>> from sympy.crypto.crypto import (
...     encipher_kid_rsa, kid_rsa_public_key)
>>> msg = 200
>>> a, b, A, B = 3, 4, 5, 6
>>> key = kid_rsa_public_key(a, b, A, B)
>>> encipher_kid_rsa(msg, key)
161
```

sympy.crypto.crypto.**decipher_kid_rsa**(*msg, key*)

Here msg is the plaintext and key is the private key.

Examples

```
>>> from sympy.crypto.crypto import (
...     kid_rsa_public_key, kid_rsa_private_key,
...     decipher_kid_rsa, encipher_kid_rsa)
>>> a, b, A, B = 3, 4, 5, 6
>>> d = kid_rsa_private_key(a, b, A, B)
>>> msg = 200
>>> pub = kid_rsa_public_key(a, b, A, B)
>>> pri = kid_rsa_private_key(a, b, A, B)
>>> ct = encipher_kid_rsa(msg, pub)
>>> decipher_kid_rsa(ct, pri)
200
```

万兴PDF

sympy.crypto.crypto.**encode_morse**(*msg, sep='|', mapping=None*)

Encodes a plaintext into popular Morse Code with letters separated by sep and words by a double sep.

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import encode_morse
>>> msg = 'ATTACK RIGHT FLANK'
>>> encode_morse(msg)
'.-|-|-|.-|-.-.|-.-||.-.|..|--.|....|-||..-.|.-..|.-|-.|-.-'
```

References

[R156]

`sympy.crypto.crypto.`**`decode_morse`**(*msg*, *sep='|'*, *mapping=None*)

Decodes a Morse Code with letters separated by `sep` (default is '|') and words by $word_sep$ (default is '||) into plaintext.

Examples

```
>>> from sympy.crypto.crypto import decode_morse
>>> mc = '--|---|...-|.||.|.-|...|-'
>>> decode_morse(mc)
'MOVE EAST'
```

万兴PDF

References

[R157]

`sympy.crypto.crypto.`**`lfsr_sequence`**(*key*, *fill*, *n*)

This function creates an LFSR sequence.

Parameters

key : list

A list of finite field elements, $[c_0, c_1, \ldots, c_k]$.

fill : list

The list of the initial terms of the LFSR sequence, $[x_0, x_1, \ldots, x_k]$.

n

Number of terms of the sequence that the function returns.

Returns

L

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

万兴PDF

Notes

S. Golomb [G157] gives a list of three statistical properties a sequence of numbers $a = \{a_n\}_{n=1}^\infty$, $a_n \in \{0,1\}$, should display to be considered "random". Define the autocorrelation of a to be

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

In the case where 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 a is periodic with period P.

- balance:

$$\left|\sum_{n=1}^P (-1)^{a_n}\right| \le 1.$$

- low autocorrelation:

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

  (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.

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import lfsr_sequence
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> lfsr_sequence(key, fill, 10)
[1 mod 2, 1 mod 2, 0 mod 2, 1 mod 2, 0 mod 2,
1 mod 2, 1 mod 2, 0 mod 2, 0 mod 2, 1 mod 2]
```

References

[G157]

sympy.crypto.crypto.**lfsr_autocorrelation**(*L*, *P*, *k*)

This function computes the LFSR autocorrelation function.

万兴PDF

Parameters

L

A periodic sequence of elements of $GF(2)$. L must have length larger than P.

P

The period of L.

k : int

An integer k ($0 < k < P$).

Returns

autocorrelation

The k-th value of the autocorrelation of the LFSR L.

Examples

```
>>> from sympy.crypto.crypto import (
...     lfsr_sequence, lfsr_autocorrelation)
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_autocorrelation(s, 15, 7)
-1/15
>>> lfsr_autocorrelation(s, 15, 0)
1
```

万兴PDF

sympy.crypto.crypto.**lfsr_connection_polynomial**(*s*)

This function computes the LFSR connection polynomial.

Parameters

s

A sequence of elements of even length, with entries in a finite field.

Returns

C(x)

The connection polynomial of a minimal LFSR yielding s.

This implements the algorithm in section 3 of J. L. Massey's article [M158].

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import (
...     lfsr_sequence, lfsr_connection_polynomial)
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**4 + x + 1
>>> fill = [F(1), F(0), F(0), F(1)]
>>> key = [F(1), F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(1), F(0)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + x**2 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + x + 1
```

万兴PDF

References

[M158]

sympy.crypto.crypto.**elgamal_public_key**(*key*)

Return three number tuple as public key.

Parameters

key : (p, r, e)

Tuple generated by `elgamal_private_key`.

Returns

tuple : (p, r, e)

$e = r * *d$ mod p

d is a random number in private key.

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import elgamal_public_key
>>> elgamal_public_key((1031, 14, 636))
(1031, 14, 212)
```

sympy.crypto.crypto.**elgamal_private_key**(*digit=10*, *seed=None*)

Return three number tuple as private key.

Parameters

digit : int

Minimum number of binary digits for key.

Returns

tuple : (p, r, d)

p = prime number.

r = primitive root.

d = random number.

Explanation

Elgamal encryption is based on the mathmatical problem called the Discrete Logarithm Problem (DLP). For example,

$a^b \equiv c \pmod p$

In general, if a and b are known, ct is easily calculated. If b is unknown, it is hard to use a and ct to get b.

万兴PDF

Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See sympy.core.random._randrange.

Examples

```
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.ntheory import is_primitive_root, isprime
>>> a, b, _ = elgamal_private_key()
>>> isprime(a)
True
>>> is_primitive_root(b, a)
True
```

sympy.crypto.crypto.**encipher_elgamal**(*i*, *key*, *seed=None*)

Encrypt message with public key.

Parameters

msg

int of encoded message.

万兴PDF

key

Public key.

Returns

tuple : (c1, c2)

Encipher into two number.

Explanation

i is a plaintext message expressed as an integer. key is public key (p, r, e). In order to encrypt a message, a random number a in range(2, p) is generated and the encryped message is returned as c_1 and c_2 where:

$c_1 \equiv r^a \pmod p$

$c_2 \equiv me^a \pmod p$

Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See sympy.core.random._randrange.

Examples

```
>>> from sympy.crypto.crypto import encipher_elgamal, elgamal_private_
↪key, elgamal_public_key
>>> pri = elgamal_private_key(5, seed=[3]); pri
(37, 2, 3)
>>> pub = elgamal_public_key(pri); pub
(37, 2, 8)
>>> msg = 36
>>> encipher_elgamal(msg, pub, seed=[3])
(8, 6)
```

万兴PDF

sympy.crypto.crypto.**decipher_elgamal**(*msg*, *key*)

Decrypt message with private key.

$msg = (c_1, c_2)$

$key = (p, r, d)$

According to extended Eucliden theorem, $uc_1^d + pn = 1$

$u \equiv 1/{c_1}^d \pmod p$

$uc_2 \equiv \frac{1}{c_1^d} c_2 \equiv \frac{1}{r^{ad}} c_2 \pmod p$

$\frac{1}{r^{ad}} me^a \equiv \frac{1}{r^{ad}} mr^{da} \equiv m \pmod p$

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import decipher_elgamal
>>> from sympy.crypto.crypto import encipher_elgamal
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.crypto.crypto import elgamal_public_key
```

```
>>> pri = elgamal_private_key(5, seed=[3])
>>> pub = elgamal_public_key(pri); pub
(37, 2, 8)
>>> msg = 17
>>> decipher_elgamal(encipher_elgamal(msg, pub), pri) == msg
True
```

sympy.crypto.crypto.**dh_public_key**(*key*)

Return three number tuple as public key.

This is the tuple that Alice sends to Bob.

Parameters

key : (p, g, a)

A tuple generated by dh_private_key.

Returns

tuple : int, int, int

A tuple of $(p, g, g^a \mod p)$ with p, g and a given as parameters.s

万兴PDF

Examples

```
>>> from sympy.crypto.crypto import dh_private_key, dh_public_key
>>> p, g, a = dh_private_key();
>>> _p, _g, x = dh_public_key((p, g, a))
>>> p == _p and g == _g
True
>>> x == pow(g, a, p)
True
```

sympy.crypto.crypto.**dh_private_key**(*digit=10*, *seed=None*)

Return three integer tuple as private key.

Parameters

digit

Minimum number of binary digits required in key.

Returns

tuple : (p, g, a)

p = prime number.

g = primitive root of p.

a = random number from 2 through p - 1.

万兴PDF

Explanation

Diffie-Hellman key exchange is based on the mathematical problem called the Discrete Logarithm Problem (see ElGamal).

Diffie-Hellman key exchange is divided into the following steps:

- Alice and Bob agree on a base that consist of a prime `p` and a primitive root of `p` called `g`
- Alice choses a number `a` and Bob choses a number `b` where `a` and `b` are random numbers in range $[2, p)$. These are their private keys.
- Alice then publicly sends Bob $g^{a} \pmod p$ while Bob sends Alice $g^{b} \pmod p$
- They both raise the received value to their secretly chosen number (`a` or `b`) and now have both as their shared key $g^{ab} \pmod p$

Notes

For testing purposes, the `seed` parameter may be set to control the output of this routine. See sympy.core.random._randrange.

Examples

```
>>> from sympy.crypto.crypto import dh_private_key
>>> from sympy.ntheory import isprime, is_primitive_root
>>> p, g, _ = dh_private_key()
>>> isprime(p)
True
>>> is_primitive_root(g, p)
True
>>> p, g, _ = dh_private_key(5)
>>> isprime(p)
True
>>> is_primitive_root(g, p)
True
```

万兴PDF

`sympy.crypto.crypto.`**`dh_shared_key`**`(`*key*, *b*`)`

Return an integer that is the shared key.

This is what Bob and Alice can both calculate using the public keys they received from each other and their private keys.

Parameters

key : (p, g, x)

Tuple (p, g, x) generated by `dh_public_key`.

b

Random number in the range of 2 to $p - 1$ (Chosen by second key exchange member (Bob)).

Returns

int