The Sage-Words Library

Creating finite words

A ***word*** w is a sequence of elements from an ***alphabet*** A (a finite set).

Collection of all words over an alphabet.

To create the collection of all words over an alphabet, use the **Words** command.

```
Words([0,1,2])
```

Words over Ordered Alphabet [0, 1, 2]

```
A = Words("ab")
A
```

Words over Ordered Alphabet ['a', 'b']

To create a word in this set, pass data that describes the word.

```
A("abbabaab")
```

word: abbabaab

```
A(["a","b","b","a","b","a","a","b"])
```

word: abbabaab

```
W = Words([0,1,2], length=3)
W
```

Finite Words over Ordered Alphabet [0, 1, 2] of length 3

```
W.list()
```

[word: 000,
word: 001,
word: 002,
word: 010,
word: 011,
word: 012,
word: 020,
word: 021,
word: 022,
word: 100,
word: 101,
word: 102,
word: 110,
word: 111,
word: 112,
word: 120,
word: 121,
word: 122,
word: 200,
word: 201,
word: 202,
word: 210,
word: 211,
word: 212,
word: 220,
word: 221,
word: 222]

Finite words from strings and lists.

You can also use the **Word** command to construct words. This builds an alphabet from the letters occurring in the word.

```
Word("abbabaab")
```

word: abbabaab

```
w = Word([0,1,1,0,1,0,0,1])
w
```

word: 01101001

```
w.alphabet()
```

Ordered Alphabet [0, 1]

Finite words from words.

Words can be concatenated.

```
abc = Word("abc")
ba = Word("ba")
```

```
abc * ba
```

word: abcba

```
ba + abc
```

word: baabc

```
ba**3
```

word: bababa

Finite words from infinite words.

If you have an infinite word, then you can slice it to get a finite word.

```
u = ba**Infinity
u
```

Infinite word over ['a', 'b']

```
u[10:50]
```

word: babababababababababababababababababababa

Constructing infinite words.

Infinite words from functions.

An **infinite word** can be described by a function f that takes values in the alphabet: $f(0)f(1)f(2)f(3)\ldots$.

```
WordOptions(letter_separator='')
Word(['f(0)','f(1)','f(2)','f(3)',"..."])
```

word: f(0)f(1)f(2)f(3)...

```
def f(n):
   return n%3
```

```
map(f, range(40))
```

$[0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0]$

```
u = Word(f, alphabet=[0,1,2])
u
```

Infinite word over [0, 1, 2]

```
u[:13]
```

word: 0120120120120

```
def t(n):
   return add(Integer(n).digits(base=2)) % 2
```

```
tm = Word(t, alphabet = [0, 1])
tm
```

Infinite word over [0, 1]

```
tm[:37]
```

word: 0110100110010110100101100110100110010

```
Word(lambda n : add(Integer(n).digits(base=2)) % 2, alphabet = [0, 1])
```

Infinite word over [0, 1]

Infinite words from iterators.

Infinite words can be constructed using an iterative process. Start with two words a and ab.

```
W = Words("ab")
```

```
f0 = W("a")
f0
```

word: a

```
f1 = W("ab")
f1
```

word: ab

Concatenate them:

```
f2 = f1 * f0
f2
```

word: aba

Next concatenate the previous two words.

```
f3 = f2 * f1
f3
```

word: abaab

Next concatenate the previous two words.

```
f4 = f3 * f2
f4
```

word: abaababa

Next concatenate the previous two words.

```
f5 = f4 * f3
f5
```

word: abaababaabaab

```
f6 = f5 * f4
f6
```

word: abaababaabaababaababa

And so on.... This is called the **Fibonacci Word**.

```
def fibword():
   f0 = "a"
   f1 = "ab"
   yield W(f0)
   while True:
       yield W(f1)
       f0, f1 = f1, f1+f0
```

```
f = fibword()
for i in range(7):
   print f.next()
```

```
word: a
word: ab
word: aba
word: abaab
word: abaababa
word: abaababaabaab
word: abaababaabaababaababa
```

```
def fibword_letter_iterator():
   r"""
   Iterates through the letters of the Fibonacci word.
   """
   n = 0
   for w in fibword():
       for x in w[n:]:
           n += 1
           yield x
```

```
F = Word(fibword_letter_iterator(), alphabet="ab")
F
```

Infinite word over ['a', 'b']

```
F[:37]
```

word: abaababaabaababaababaabaababaabaababa

Infinite words from morphisms.

Let $mu : A \to Words(A)$.

```
mu = WordMorphism('a->ab,b->ba'); mu
```

WordMorphism: a->ab, b->ba

```
mu('a')
```

word: ab

```
mu(_)
```

word: abba

```
mu(_)
```

word: abbabaab

```
mu(_)
```

word: abbabaabbaababba

```
mu(_)
```

word: abbabaabbaababbabaababbaabbabaab

```
tm = mu('a',Infinity)
tm
```

Fixed point beginning with 'a' of the morphism WordMorphism: a->ab, b->ba

```
tm[:37]
```

word: abbabaabbaababbabaababbaabbabaabbaaba

Pre-defined words.

```
words.FibonacciWord()
```

Fibonacci word over [0, 1], defined recursively

```
words.FibonacciWord("ab")
```

Fibonacci word over ['a', 'b'], defined recursively

```
words.ThueMorseWord("ab")
```

Thue-Morse word on the alphabet ['a', 'b']

```
words.FixedPointOfMorphism(mu,'a')
```

Fixed point beginning with 'a' of the morphism WordMorphism: a->ab, b->ba

```
words.ChristoffelWord(7,3,"xy")
```

word: xyyxyyxyyy

```
words.RandomWord(18,5)
```

word: 003133210413204143

```
Tribonacci = words.StandardEpisturmianWord(Word('abc'))
Tribonacci
```

Standard episturmian word over ['a', 'b', 'c']

```
Tribonacci[:40]
```

word: abacabaabacababacabaabacabacabaabacababa

Create your own word class.

Lyndon Words

A word w is a **Lyndon word** if it appears first in dictionary order among its cyclic rearrangements. (The cyclic rearrangements of a word are called its conjugates.)

```
w = Word("abbaab")
w
```

word: abbaab

```
w.conjugates()
```

set([word: aababb, word: baabab, word: babbaa, word: abbaab, word: bbaaba, word: abab

```
min(w.conjugates())
```

word: aababb

```
class LyndonWord(sage.combinat.words.word.FiniteWord_over_OrderedAlphabet):
   def __init__(self, lw, alphabet=(0,1)):
       # initialize
       super(LyndonWord, self).__init__(Words(alphabet), lw)

       # type checking
       if not self.is_lyndon():
           raise TypeError, "not a Lyndon word"
```

```
LyndonWord([0,0,1,0,1,1])
```

word: 001011

```
LyndonWord("abb", alphabet="ab")
```

word: abb

```
LyndonWord("abbaab", alphabet="ab")
```

```
Traceback (click to the left for traceback)
...
TypeError: not a Lyndon word
```

```
w = Word("abbaab")
```

```
w.is_lyndon()
```

False

```
Word("abb").is_lyndon()
```

True

```
Word("aab").is_lyndon()
```

True

```
print w.lyndon_factorization()
```

(abb.aab)

Interrogating words

```
w = words.ThueMorseWord("ab")[:8]
w
```

word: abbabaab

```
w.is_palindrome()
```

False

```
w.is_lyndon()
```

False

```
print w.lyndon_factorization()
```

(abb.ab.aab)

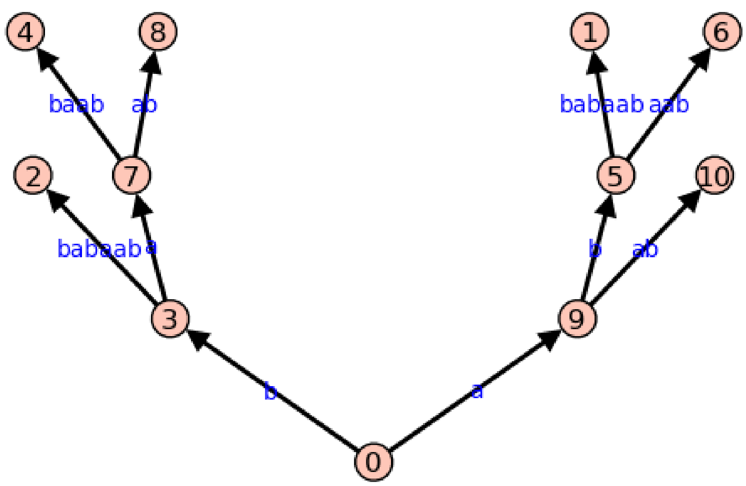
```
print w.crochemore_factorization()
```

(a.b.b.ab.a.ab)

```
st = w.suffix_tree()
st
```

Implicit Suffix Tree of the word: abbabaab

```
st.show(word_labels=True)
```



```
w.number_of_factors()
```

28

```
Word("abba").factor_set()
```

{word: , word: b, word: bb, word: abb, word: ab, word: ba, word: bba, word: abba, word: a}

Currently available commands

```
for s in dir(w):
   if not s.startswith("_"):
       print s
```

```
BWT
alphabet
apply_morphism
apply_permutation_to_letters
apply_permutation_to_positions
border
category
charge
coerce
colored_vector
commutes_with
complete_return_words
conjugate
conjugate_position
conjugates
count
critical_exponent
crochemore_factorization
db
defect
deg_inv_lex_less
deg_lex_less
deg_rev_lex_less
degree
delta
delta_derivate
delta_derivate_left
delta_derivate_right
delta_inv
dump
dumps
evaluation
evaluation_dict
evaluation_partition
evaluation_sparse
exponent
factor_iterator
factor_occurrences_in
factor_set
first_pos_in
freq
good_suffix_table
implicit_suffix_tree
```

```
inv_lex_less
inversions
is_balanced
is_cadence
is_conjugate_with
is_cube
is_cube_free
is_empty
is_factor_of
is_full
is_lyndon
is_overlap
is_palindrome
is_prefix_of
is_primitive
is_proper_prefix_of
is_proper_suffix_of
is_quasiperiodic
is_smooth_prefix
is_square
is_square_free
is_subword_of
is_suffix_of
is_symmetric
iterated_palindromic_closure
lacunas
last_position_table
length_border
lengths_lps
lengths_unioccurrent_lps
lex_greater
lex_less
longest_common_prefix
longest_common_suffix
lps
lyndon_factorization
minimal_period
nb_factor_occurrences_in
nb_subword_occurrences_in
number_of_factors
order
overlap_partition
palindromes
palindromic_closure
palindromic_lacunas_study
parent
parikh_vector
phi
phi_inv
prefix_function_table
primitive
primitive_length
```

```
quasiperiods
rename
reset_name
return_words
return_words_derivate
rev_lex_less
reversal
save
shifted_shuffle
shuffle
standard_factorization
standard_factorization_of_lyndon_factorization
standard_permutation
string_rep
suffix_tree
suffix_trie
swap
swap_decrease
swap_increase
version
```