

---

# **thompsons\_v Documentation**

*Release*

**Author**

March 07, 2016



<b>1 Overview</b>	<b>3</b>
<b>2 Implementation details</b>	<b>5</b>
<b>3 Contents</b>	<b>7</b>
<b>4 Indices</b>	<b>61</b>
<b>Bibliography</b>	<b>63</b>
<b>Python Module Index</b>	<b>65</b>



Nathan Barker, Andrew Duncan and David Robertson have submitted a paper entitled *The power conjugacy problem in Higman-Thompson groups* which addresses the following problem in the groups named  $G_{n,r}$ .

[AS74] Given two elements  $x$  and  $y$  of a group, are there integers  $a$  and  $b$  and a third element  $z$  for which  $x^a = zy^bz^{-1} \neq 1$ ? If so, provide them.

This package aims to implement the algorithms described in the paper. To do so it provides tools for working in

- the algebra  $V_{n,r}$ , a certain set of *words*, and
- the automorphism group  $G_{n,r} = \text{Aut}(V_{n,r})$ .

This documentation serves three purposes. Firstly, it is meant to serve as a gentle introduction, explaining how to install and use the package. Secondly, it is meant to be a reference to all the various classes and functions provided by `thompson`. Finally, a number of examples are included throughout the documentation, which can be used as a means to test the implementation.

David M. Robertson, March 07, 2016



---

## Overview

---

**Warning:** For the most part, this documentation is automatically generated by Sphinx , so it's not the prettiest thing that's ever been written.

This all began as a series of tools to draw tree pair diagrams in Thompson's group  $V = G_{2,1}$ , hence the name `thompson`. The focus has moved away from trees diagrams towards bijections between bases of words. From there, the package has grown and it now serves as an implementation of the algorithms we describe in the paper. Chief among these are the algorithms which:

- Determine the *type of component which contains a given element*, with respect to some given basis [Lemma 4.28]
- Computes the *quasi-normal basis* for a given automorphism [Lemma 4.28]
- Test to see if two words *share an orbit* [Lemma 4.34]
- Test to see if two automorphism are *conjugate* [Algorithm 5.6] and *power conjugate* [Algorithm 6.13].



---

## Implementation details

---

The implementation is written in Python and runs under Python 3.3 and above. The intent was to provide a proof-of-concept rather than a perfectly optimised implementation. Despite this, we have found the program useful as a calculator for  $G_{n,r}$ , as a means to generate examples, and for experimentally testing conjectures. The source code (both to the program and this documentation) is publicly available from [GitHub](#).

---

### Todo

Make this available under an open-source license.

---



---

## 3.1 Getting Started

### 3.1.1 Installation

1. **Python interpreter.** This program is written in Python and needs a Python interpreter to run; to be specific, **Python 3.3 or above** is required. If you don't have this already installed on your system, your best bet is to download the latest version (currently 3.5) from the Python website. For Linux users, Python may be available from your distribution's repositories; take care to install version 3.x rather than 2.x!
2. **Source code.** If you have git available on your system, the easiest way to obtain the program is to simply clone the repository:

```
git clone https://github.com/DMRRobertson/thompsons_v.git
```

If this is not possible, a ZIPped version can be downloaded from the [project page](#) on GitHub.

3. **Package requirements.** The de-facto package management tool for Python is the `pip` program, which is installed as part of Python 3.4 and higher. Python 3.3 users should consult the [pip documentation](#) for instructions on how to install `pip` directly. Some Linux distributions may also provide `pip` as a package (make sure it's for Python 3—it may be listed as `pip3`).

Once `pip` is installed, navigate to the project directory and run

```
pip install -r requirements.txt
```

(possibly you may need to use `pip3`). This should do all the hard work for you.

**Direct installation:** If you would prefer not to install `pip`, you may install the prerequisites directly. There is only one major Python package required to run the program: [NetworkX](#), a library for working with graphs in Python. This may be available in Linux repositories, e.g. [Ubuntu](#).

This documentation is built using [Sphinx](#). If you wish to build it yourself, or run the test suite, you must install Sphinx also. Again, consider checking Linux repositories e.g. [in Ubuntu](#).

### 3.1.2 First steps

Navigate to the root folder (`thompsons_v`) in a terminal and start up Python using the `python` command. Once the python prompt (`>>>`) is available, try to `import thompson`:

```
$ python
Python 3.3.3 (v3.3.3:c3896275c0f6, Nov 18 2013, 21:19:30) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> import thompson
>>>
```

If this fails, check that you're using Python 3.3 or above. If python started up Python 2.x, try using the command `python3` instead.

We can now use the python [REPL](#) as a sort of calculator for these groups.

```
>>> from thompson import *
>>> sig = (2, 1) #signature of the algebra we work in
>>> domain = Generators.standard_basis(sig).expand(0).expand(0)
>>> range = Generators.standard_basis(sig).expand(0).expand(1)
>>> print(domain)
[x1 a1 a1, x1 a1 a2, x1 a2]
>>> print(range)
[x1 a1, x1 a2 a1, x1 a2 a2]
>>> phi = Automorphism(domain, range)
>>> print(phi)
InfiniteAut: V(2, 1) -> V(2, 1) specified by 3 generators (after expansion and reduction).
x1 a1 a1 -> x1 a1
x1 a1 a2 -> x1 a2 a1
x1 a2 -> x1 a2 a2
```

```
>>> phi.image('x a2 a1 a2 a2 a1')
Word('x1 a2 a2 a1 a2 a2 a1', (2, 1))
>>> phi.order
inf
>>> phi.print_characteristics()
(-1, a1)
(1, a2)
>>> phi.dump_QNB()
x1 a1 Left semi-infinite component with characteristic (-1, a1)
x1 a2 Right semi-infinite component with characteristic (1, a2)
```

```
>>> print(phi ** 2)
InfiniteAut: V(2, 1) -> V(2, 1) specified by 4 generators (after expansion and reduction).
x1 a1 a1 a1 -> x1 a1
x1 a1 a1 a2 -> x1 a2 a1
x1 a1 a2 -> x1 a2 a2 a1
x1 a2 -> x1 a2 a2 a2
>>> print(~phi) #inverse
InfiniteAut: V(2, 1) -> V(2, 1) specified by 3 generators (after expansion and reduction).
x1 a1 -> x1 a1 a1
x1 a2 a1 -> x1 a1 a2
x1 a2 a2 -> x1 a2
>>> print(~phi * phi) # == identity
PeriodicAut: V(2, 1) -> V(2, 1) specified by 1 generators (after expansion and reduction).
x1 -> x1
```

### 3.1.3 Locating examples

A *number of examples* are included in this package, some of which are used in [\[BDR\]](#). To access them, use the `load_example()` function:

```
>>> from thompson import *
>>> phi = load_example('nathan_pond_example')
>>> print(phi)
InfiniteAut: V(2, 1) -> V(2, 1) specified by 7 generators (after expansion and reduction).
```

```
x1 a1 a1 a1 a1 -> x1 a1 a1
x1 a1 a1 a1 a2 -> x1 a1 a2 a1 a1
x1 a1 a1 a2 a1 -> x1 a2 a2
x1 a1 a1 a2 a2 -> x1 a2 a1
x1 a1 a2 -> x1 a1 a2 a1 a2
x1 a2 a1 -> x1 a1 a2 a2 a2
x1 a2 a2 -> x1 a1 a2 a2 a1
```

**Note:** Previously, one would access this example by using `from thompson.examples import nathan_pond_example`. I changed this, because it meant that every example was loaded (and the quasinormal bases etc. computed) whenever `thompson.examples` was imported.

To see the list of available examples, consult the [documentation for the examples module](#). You might also like to consult the [demo notebook](#) on GitHub.

### 3.1.4 Running the test suite

Make sure Sphinx is installed (see the installation section). Then navigate to the `docs` folder and run `make doctest`. This should look through the source code for short tests and alert you if any of them fail. You can also run `make html` to build this documentation. Reset with `make clean`.

## 3.2 Number theory

At the bottom level of the package hierarchy are various helper functions which implement standard number-theoretic algorithms.

`thompson.number_theory.gcd(a, b)`

Calculate the Greatest Common Divisor of  $a$  and  $b$ .

Unless  $b=0$ , the result will have the same sign as  $b$  (so that when  $b$  is divided by it, the result comes out positive).

```
>>> gcd(12, 8)
4
>>> gcd(0, 50)
50
>>> gcd(7, 101)
1
```

`thompson.number_theory.extended_gcd(a, b)`

From [this exposition of the extended gcd algorithm](#). Computes  $d = \gcd(a, b)$  and returns a triple  $(d, x, y)$  where  $d = ax + by$ .

```
>>> extended_gcd(14, 33)
(1, -7, 3)
>>> extended_gcd(12, 32)
(4, 3, -1)
```

`thompson.number_theory.lcm(*args)`

Computes the least common multiple of the given *args*. If a single argument is provided, the least common multiple of its elements is computed.

```

>>> n = randint(1, 1000000)
>>> lcm(n) == n
True
>>> lcm(2, 13)
26
>>> lcm(*range(1, 10)) #1, 2, ..., 9
2520
>>> lcm(range(1, 10)) #Don't have to unpack the arguments
2520

```

`thompson.number_theory.solve_linear_diophantine(a, b, c)`  
Solves the equation  $ax + by = c$  for integers  $x, y \in \mathbb{Z}$ .

**Return type**

None if no solution exists; otherwise a triple (*base*, *inc*, *lcm*) where

- *base* is a single solution  $(x_0, y_0)$ ;
- *inc* is a pair  $(\delta x, \delta y)$  such that if  $(x, y)$  is a solution, so is  $(x, y) \pm (\delta x, \delta y)$ ; and
- *lcm* is the value of  $\text{lcm}(a, b)$ .

`thompson.number_theory.solve_linear_congruence(a, b, n)`  
Solves the congruence  $ax \equiv b \pmod{n}$ .

**Return type** a *SolutionSet* representing all such solutions  $x$ .

```

>>> x = solve_linear_congruence(6, 12, 18)
>>> print(x)
{..., -1, 2, 5, 8, 11, 14, ...}
>>> y = solve_linear_congruence(5, 7, 11)
>>> print(y)
{..., -3, 8, 19, 30, 41, 52, ...}
>>> print(x & y)
{..., -25, 8, 41, 74, 107, 140, ...}

```

`thompson.number_theory.divisors(n, include_one=True)`  
An iterator that yields the positive divisors  $d \mid n$ .

**Parameters** `include_one` (*bool*) – set to `False` to exclude  $d = 1$ .

```

>>> list(divisors(12))
[1, 2, 3, 4, 6, 12]
>>> list(divisors(125, include_one=False))
[5, 25, 125]

```

`thompson.number_theory.prod(iterable)`  
Handy function for computing the product of an iterable collection of numbers. From [Stack Overflow](#).

```

>>> prod(range(1, 5))
24

```

### 3.3 Words and standard forms

By a *word*, we mean a *string* written using the symbols

$$X \cup \Omega = \{x_1, \dots, x_r\} \cup \{\alpha_1, \dots, \alpha_n, \lambda\}.$$

We treat the  $x_i$  are constants, the  $\alpha_i$  are unary operators and  $\lambda$  as an  $n$ -ary operation. We refer to the parameters  $n$  and  $r$  as the *arity* and *alphabet size* respectively.

### 3.3.1 Notation

If we collect together all such words, we obtain an *algebra* (in the sense of [universal algebra](#)) of words. We consider three different algebras; using the notation of [\[Cohn81\]](#):

1.  $W(\Omega; X)$ , the set of all finite strings written using letters in  $X \cup \Omega$ . In other words/jargon, this is the [free monoid](#) on  $X \cup \Omega$ . Cohn calls these strings  $\Omega$ -rows.
2.  $W_\Omega(X)$ , the subset of  $W(\Omega; X)$  whose strings represent a *valid* series of operations. *Valid* means that each the operations  $\alpha_i$  and  $\lambda$  always receive the correct number of arguments. Cohn calls these strings  $\Omega$ -words.
3.  $V_{n,r}(X) = W_\Omega(X)/q$ . This is equivalent to the set of words in  $W_\Omega(X)$  which are in Higman's *standard form*. Roughly speaking, a word is in standard form if is valid (type 2) and it cannot be reduced to a shorter word using the following two rules.
  - (a)  $w_1 \dots w_n \lambda \alpha_i = w_i$ , where each  $w_i$  is in standard form.
  - (b)  $w \alpha_1 \dots w \alpha_n \lambda = w$ , where  $w$  is in standard form.

Sometimes we need to refer to a string which consists only of  $\alpha$ -s. Write  $A$  for the set  $\{\alpha_1, \dots, \alpha_n\}$ . We define  $A^*$  to be the set of finite strings over  $A$ . (Again, this is the [free monoid](#) on  $A$ .)

Finally, let  $S$  be a set of words. We define  $X\langle A \rangle$  to be the set of concatenations  $s\Gamma$ , where  $s \in S$  and  $\Gamma \in A^*$ . It is sometimes helpful to think of  $S\langle A \rangle$  as the set of words *below*  $S$ .

#### See also:

Sections 2 and 2.3 and Remark 3.3 of the paper.

### 3.3.2 Signatures

It may happen that we need to work in different algebras  $V_{n,r}, V_{m,s}$  at the same time. To keep track of the algebras that different words belong to, we associate a *Signature* to each word  $w \in V_{n,r}$ .<sup>1</sup> Signatures are implemented as a glorified tuple  $(n, r)$ . This means they are [immutable](#).

**class** thompson.word.**Signature**

**static** **\_\_new\_\_** (*arity, alphabet\_size*)

Signatures store an *arity*  $n \geq 2$  and an *alphabet\_size*  $r \geq 1$ .

**\_\_contains\_\_** (*other*)

We can use the `in` operator to see if a *Word* lies in the algebra that this signature describes.

```
>>> s = Signature(2, 1)
>>> Word('x1 a1 a2', s) in s
True
>>> Word('x1 a1 a2', (2, 2)) in s
False
```

**is\_isomorphic\_to** (*other*)

We can test the (signatures of) two algebras  $V_{n,r}$  and  $V_{m,s}$  to see if they are isomorphic. This happens precisely when  $n = m$  and  $r \equiv s \pmod{n-1}$ .

```
>>> Signature(2, 1).is_isomorphic_to(Signature(2, 4))
True
>>> Signature(2, 1).is_isomorphic_to(Signature(3, 1))
False
>>> Signature(3, 2).is_isomorphic_to(Signature(3, 1))
```

<sup>1</sup> This made it easier to catch bugs before they happened when I was passing words around as arguments to functions.

```
False
>>> Signature(3, 3).is_isomorphic_to(Signature(3, 1))
True
```

**See also:**

Corollary 3.14 of the paper.

### 3.3.3 Operations on words

We represent a word as a `tuple` of integers, where:

- $x_i$  is represented by  $i$ ,
- $\alpha_i$  is represented by  $-i$ , and
- $\lambda$  is represented by 0.

We can write words of all types in this format, but we're only interested in the standard forms (*type 3*). To this end, the `validate()` detects those which are of *type 2*, and `standardise()` reduces type 2 words to type 3.

`thompson.word.format(word)`

Turns a sequence of integers representing a *word* into a nicely formatted string. Can be thought of as an inverse to `from_string()`. The *word* is not processed or reduced in any way.

```
>>> format([2, -1, 2, -2, 0])
'x2 a1 x2 a2 L'
>>> format([])
'<the empty word>'
```

`thompson.word.from_string(str)`

Converts a string representing a word to the internal format—a tuple of integers. Anything which does not denote a basis letter (e.g. 'x', 'x2', 'x45') a descendant operator (e.g. 'a1', 'a3', 'a27') or a contraction ('L' for  $\lambda$ ) is ignored.

Note that 'x' is interpreted as shorthand for 'x1'.

```
>>> x = from_string('x2 a1 x2 a2 L')
>>> print(x); print(format(x))
(2, -1, 2, -2, 0)
x2 a1 x2 a2 L
>>> w = random_simple_word()
>>> from_string(str(w)) == w
True
```

**Return type** `tuple` of integers

---

**Todo**

More convenient ways of inputting words, e.g. `x a1^3` instead of `x a1 a1 a1`.

---

`thompson.word.validate(letters, signature)`

Checks that the given *letters* describe a valid word belonging to the algebra with the given *signature* = (*arity*, *alphabet\_size*). If no errors are raised when running this function, then we know that:

- The first letter is an  $x_i$ .
- Every  $\lambda$  has *arity* arguments to its left.

- If the word contains a  $\lambda$ , every subword is an argument of some  $\lambda$ .
- No  $\alpha_i$  appears with  $i > \text{arity}$ .
- No  $x_i$  occurs with  $i > \text{alphabet\_size}$ .

Calling this function does not modify *letters*. The argument *letters* need not be in standard form.

```
>>> validate(from_string('a1 x a2 L'), (2, 1))
Traceback (most recent call last):
...
ValueError: The first letter (a1) should be an x.
>>> validate(from_string('x1 a2 x12 a1 a2 L'), (2, 4))
Traceback (most recent call last):
...
IndexError: Letter x12 at index 2 is not in the alphabet (maximum is x4).
>>> validate(from_string('x1 a1 a2 a3 a4 a5 x2 a2 L'), (3, 2))
Traceback (most recent call last):
...
IndexError: Letter a4 at index 4 is not in the alphabet (maximum is a3).
>>> validate(from_string('x1 a2 L'), (4, 1))
Traceback (most recent call last):
...
ValueError: Letter L at index 2 is invalid. Check that lambda has 4 arguments.
>>> validate(from_string('x1 x1 L x1 L x1'), (2, 1))
Traceback (most recent call last):
...
ValueError: Word is invalid: valency is 2 (should be 1).
```

### Raises

- **ValueError** – if the first letter is not an  $x_i$ .
- **IndexError** – if this word contains an  $x_i$  with  $i$  outside of the range 1 ... *alphabet\_size*
- **IndexError** – if this word contains an  $\alpha_i$  outside of the range 1 ... *arity*
- **IndexError** – if this word is empty (i.e. consists of 0 letters).
- **ValueError** – if this word fails the valency test.

### See also:

Proposition 2.12 of the paper for the *valency test*.

`thompson.word.standardise` (*letters, signature, tail=()*)

Accepts a valid word as a `tuple` of *letters* and reduces it to standard form. The result—a new `tuple` of letters—is returned. The *signature* must be given so we know how many arguments to pass to each  $\lambda$ . The *tail* argument is used internally in recursive calls to this function and should be omitted.

In the examples below, the *Word* class standardises its input before creating a *Word*.

```
>>> #Extraction only
>>> print(Word("x1 a2 a1 x a2 a1 L a1 a2", (2, 1)))
x1 a1 a2 a1 a2
>>> #Contraction only
>>> print(Word("x2 a2 a2 a1 x2 a2 a2 a2 L", (2, 2)))
x2 a2 a2
>>> #Contraction only, arity 3
>>> print(Word("x1 a1 a1 x1 a1 a2 x1 a1 a3 L", (3, 2)))
x1 a1
>>> #Both extraction and contraction
```

```

>>> print (Word("x a1 a2 a1 a1 x a1 a2 a1 x a2 a1 L a1 a2 a1 x a1 a2 a1 a2 a2 L L", (2, 1)))
x1 a1 a2 a1
>>> #Lambda concatenation
>>> print (Word("x a1 a2 a1 x a1 a2 a1 x a2 a1 L a1 a2 a1 x a1 a2 a1 a2 a2 L L", (2, 1)))
x1 a1 a2 a1 x1 a1 a2 a1 a2 L
>>> #A mix of contraction and extraction
>>> print (Word("x a2 x a1 L x a1 L a1 a2 a1 a2", (2, 1)))
x1 a1 a1 a2
>>> #Can't contract different x-es
>>> print (Word('x1 a1 x2 a2 L', (2, 2)))
x1 a1 x2 a2 L
>>> #Something meaty, arity 3
>>> print (Word('x1 a1 x1 a2 x1 a3 L a2 a1 a3 x1 a2 a1 x2 a1 x1 a1 a1 x1 a1 a2 x2 L x1 a2 L a2 a1
x1 a1 a1 a1 a3 a2

```

**Raises**

- **TypeError** – if *letters* is not a tuple of integers.
- **IndexError** – if *letters* describes an *invalid* word.

**Return type** tuple of integers.

**See also:**

Remark 3.3 of the paper.

`thompson.word.lambda_arguments` (*word*, *signature=None*)

This function takes a *valid* word which ends in a  $\lambda$ , and returns the arguments of the rightmost  $\lambda$  as a list of *Words*.

If *word* is given as a *Word* instance, then *signature* may be omitted. Otherwise the *signature* should be provided.

```

>>> w = Word('x a1 a2 x a2 a2 L x a1 a1 L', (2, 1))
>>> subwords = lambda_arguments(w)
>>> for subword in subwords: print (subword)
x1 a1 a2 x1 a2 a2 L
x1 a1 a1
>>> w = 'x x x x a1 x L x a1 x a2 x L L x a1 a2 x L x a1 a2 x a2 a1 x L L'
>>> subwords = lambda_arguments(Word(w, (3, 1)))
>>> for subword in subwords: print (subword)
x1
x1 x1 x1 a1 x1 L x1 a1 x1 a2 x1 L L x1 a1 a2 x1 L
x1 a1 a2 x1 a2 a1 x1 L

```

**Raises**

- **IndexError** – if *word* is an empty list of letters.
- **ValueError** – if the last letter in *word* is not a lambda.
- **TypeError** – if no arity is provided and *word* has no arity attribute.

**Return type** list of *Words*.

`thompson.word.are_contractible` (*words*)

Let *words* be a list of words, either as sequences of integers or as full *Word* objects. This function tests to see if *words* is a list of the form  $(w\alpha_1, \dots, w\alpha_n)$ , where  $n == \text{len}(\text{words})$ .

**Returns** *w* (as a tuple of integers) if the test passes; the empty tuple if the test fails.

**Warning:** If there is a prefix  $w$ , this function does **not** check to see if

- $w$  is a valid word, or
- $n$  is the same as the arity of the context we're working in.

```
>>> prefix = are_contractible(
...     [Word("x a1 a2 a3 a1", (3, 1)), Word("x a1 a2 a3 a2", (3, 1)), Word("x a1 a2 a3 a3", (3,
...     1))], Word("x a1 a2 a3 a3", (3, 1)))
>>> format(prefix)
'x1 a1 a2 a3'
```

`thompson.word.free_monoid_on_alphas` (*arity*)

An infinite iterator which enumerates the elements of  $A^*$  in `shortlex` order.

```
>>> for i, gamma in enumerate(free_monoid_on_alphas(4)):
...     if i >= 10: break
...     print(format(gamma))
<the empty word>
a1
a2
a3
a4
a1 a1
a1 a2
a1 a3
a1 a4
a2 a1
```

`thompson.word.root` (*sequence*)

Given a sequence  $\Gamma \in A^*$ , this function computes the root  $\sqrt{\Gamma} \in A^*$  and the root power  $r \in \mathbb{N}$ . These are objects such that  $(\sqrt{\Gamma})^r = \Gamma$ , and  $r$  is maximal with this property.

**Returns** the pair  $(\sqrt{\Gamma}, r)$

```
>>> root('abababab')
('ab', 4)
>>> root([1, 2, 3, 1, 2, 3])
([1, 2, 3], 2)
```

**See also:**

The discussion following Corollary 6.7 in the paper.

### 3.3.4 The Word class

It's important to know when a sequence of letters denotes a word in standard form. The `Word` class addresses this problem by standardising its input upon creation. We can think of a `Word` object as a fixed list of letters with the guarantee that it is in standard form. Words are also given a *Signature* at creation time, so that we know which algebra the word comes from.

We also need to know that once a `Word` has been created, it cannot be modified to something not in standard form. We achieve this simply by making it impossible to modify a `Word`. Words are implemented as (subclasses of) `tuple`, which are `immutable`. One useful side effect of this is that `Words` can be used as dictionary keys.<sup>2</sup>

```
>>> w = Word('x1 a1', (2, 1))
>>> x = {}; x[w] = 'stored value'
>>> x[w]
'stored value'
```

<sup>2</sup> This made it so much easier to cache the images of *homomorphisms*, because I could just use a vanilla Python dictionary.

```
>>> #Can also use the underlying tuple as a key
>>> x[1, -1]
'stored value'
>>> #the reason this works
>>> hash(w) == hash((1, -1))
True
```

```
class thompson.word.Word
```

### Variables

- **signature** – the signature of the algebra this word belongs to.
- **lambda\_length** – the number of times the contraction symbol  $\lambda$  occurs in this word after it has been written in standard form.

**static** `__new__` (*letters, signature, preprocess=True*)

Creates a new `Word` consisting of the given *letters* belonging the algebra with the given *signature*. The *letters* may be given as a list of integers or as a string (which is passed to the `from_string()` function). The signature can be given as a tuple of a fully-fledged *Signature*.

```
>>> Word("x2 a1 a2 a3 x1 a1 a2 a1 x2 L a2", (3, 2))
Word('x1 a1 a2 a1', (3, 2))
```

By default, the argument `preprocess` is `True`. This means that words are *validated* and *reduced to standard form* before they are stored as a tuple. These steps can be disabled by using `preprocess=False`. This option is used internally when we *know* we have letters which are already valid and in standard form.

**Raises** `ValueError` – See the errors raised by `validate()`.

`__lt__` (*other*)

Let us assign a total order to the alphabet  $X \cup \Omega$  by declaring:

$$\alpha_1 < \dots < \alpha_n < x_1 < \dots < x_r < \lambda$$

We can use this to order *simple words* by using dictionary order.

```
>>> Word('x1', (2, 2)) < Word('x1', (2, 2))
False
>>> Word('x1', (2, 2)) < Word('x2', (2, 2))
True
>>> Word('x1 a2', (2, 2)) < Word('x1 a1', (2, 2))
False
>>> Word('x1 a1 a2 a3 a4', (4, 1)) < Word('x1 a1 a2 a3 a3', (4, 1))
False
```

We extend this to non-simple words *in standard form* in the following way. Let  $\lambda(u)$  denote the lambda-length of  $u$ .

- 1.If  $\lambda(u) < \lambda(v)$ , then  $u < v$ .

```
>>> Word('x2 x1 L', (2, 2)) < Word('x1 x2 x1 L L', (2, 2))
True
```

- 2.If  $u = v$ , then neither is strictly less than the other.

```
>>> Word('x1 x2 L', (2, 2)) < Word('x1 x2 L', (2, 2))
False
```

3. Otherwise  $u \neq v$  are different words with the same  $\lambda$ -length. Break both words into the  $n$  arguments of the outmost lambda, so that  $u = u_1 \dots u_n \lambda$  and  $v = v_1 \dots v_n \lambda$ , where each subword is in standard form.
4. Let  $i$  be the first index for which  $u_i \neq v_i$ . Test if  $u_i < v_i$  by applying this definition recursively. If this is the case, then  $u < v$ ; otherwise,  $u > v$ .

```
>>> #True, because x2 < x2 a2
>>> Word('x1 x2 L', (2, 2)) < Word('x1 x2 a2 L', (2, 2))
True
>>> #True, because words of lambda-length 1 are less than those of lambda-length 2.
>>> Word('x1 x2 L x3 x4 L L', (2, 4)) < Word('x1 x2 L x3 L x4 L', (2, 4))
True
```

The other three comparison operators (`<=`, `>`, `>=`) are also implemented.

```
>>> Word('x1 x2 L', (2, 2)) <= Word('x1 x2 L', (2, 2))
True
>>> Word('x1 a1', (2, 2)) <= Word('x1 a1', (2, 2)) <= Word('x1 a1 a2', (2, 2))
True
>>> Word('x1', (2, 2)) > Word('x1', (2, 2))
False
>>> Word('x1', (2, 2)) >= Word('x1', (2, 2))
True
>>> Word('x1 a2', (2, 2)) > Word('x1', (2, 2))
True
>>> Word('x1 a2', (2, 2)) >= Word('x1', (2, 2))
True
```

---

### Todo

I think this is a total order—try to prove this. I based the idea on [Zaks] (section 2, definition 1) which describes a total order on  $k$ -ary trees. On another note, isn't this similar to shortlex order?

---

### `is_simple()`

Let us call a *Word simple* if it has a lambda length of zero once it has been reduced to standard form.

```
>>> Word("x1 a1 a2", (2, 1)).is_simple()
True
>>> Word("x1 a1 a2 x2 a2 L", (2, 2)).is_simple()
False
>>> #Simplify a contraction
>>> Word("x1 a1 a1 x1 a1 a2 L", (2, 1)).is_simple()
True
>>> #Lambda-alpha extraction
>>> Word("x1 x2 L a2", (2, 2)).is_simple()
True
```

### `is_above(word)`

*Tests* to see if the current word is an initial segment of *word*. Returns True if the test passes and False if the test fails.

```
>>> w = Word('x1 a1', (2, 2))
>>> w.is_above(Word('x1 a1 a1 a2', (2, 2)))
True
>>> w.is_above(Word('x1', (2, 2)))
False
>>> w.is_above(w)
```

```

True
>>> v = Word('x1 a1 x1 a2 L', (2, 2))
>>> w.is_above(v)
False
>>> v.is_above(w)
True

```

**test\_above** (*word*)

Tests to see if the current word  $c$  is an initial segment of the given *word*  $w$ . In symbols, we're testing if  $w = c\Gamma$ , where  $\Gamma \in \langle A \rangle$  is some string of alphas only.

The test returns  $\Gamma$  (as a tuple of integers) if the test passes; note that  $\Gamma$  could be the empty word 1 (if  $c = w$ ). If the test fails, returns None.

```

>>> c = Word('x1 a1', (2, 2))
>>> c.test_above(Word('x1 a1 a1 a2', (2, 2)))
(-1, -2)
>>> c.test_above(Word('x1', (2, 2))) is None
True
>>> c.test_above(c)
()
>>> w = Word('x1 a2 x1 a1 L', (2, 2))
>>> print(c.test_above(w))
None
>>> w.test_above(c)
(-2,)
>>> v = Word('x1 a2 x1 a1 L x1 a2 a2 L x1 a2 x1 a1 L L', (2, 2))
>>> print(c.test_above(v))
None
>>> #There are two possible \Gamma values here; only one is returned.
>>> v.test_above(c)
(-1, -1, -2)

```

---

**Note:** There may be many different values of  $\Gamma$  for which  $w = c\Gamma$ ; this method simply returns the first such  $\Gamma$  that it finds.

---

**max\_depth\_to** (*basis*)

Let  $w$  be the current word and let  $X$  be a *basis*. Choose a (possibly empty) string of alphas  $\Gamma$  of length  $s \geq 0$  at random. What is the smallest value of  $s$  for which we can guarantee that  $w\Gamma$  is below  $X$ , i.e. in  $X\langle A \rangle$ ?

```

>>> from thompson.generators import Generators
>>> basis = Generators.standard_basis((2, 1)).expand(0).expand(0).expand(0)
>>> basis
Generators((2, 1), ['x1 a1 a1 a1', 'x1 a1 a1 a2', 'x1 a1 a2', 'x1 a2'])
>>> Word('x', (2, 1)).max_depth_to(basis)
3
>>> Word('x a1', (2, 1)).max_depth_to(basis)
2
>>> Word('x a2 x a1 L x1 a1 L', (2, 1)).max_depth_to(basis)
4
>>> Word('x a2', (2, 1)).max_depth_to(basis)
0

```

**Warning:** If *basis* doesn't actually generate the algebra we're working in, this method could potentially loop forever.

**as\_interval()**

Returns a pair (*start*, *end*) of Fractions which describe the interval  $I \subseteq [0, 1]$  that this word corresponds to.

```
>>> def print_interval(w, s):
...     start, end = Word(w, s).as_interval()
...     print('{{}, {}'.format(start, end))
...
>>> print_interval('x a1 a2 a1 x L', (2, 1))
Traceback (most recent call last):
...
ValueError: The non-simple word x1 a1 a2 a1 x1 L does not correspond to a interval.
>>> print_interval('x1', (2, 1))
[0, 1]
>>> print_interval('x1', (3, 1))
[0, 1]
>>> print_interval('x1', (4, 2))
[0, 1/2]
>>> print_interval('x1 a1', (2, 1))
[0, 1/2]
>>> print_interval('x1 a1', (3, 1))
[0, 1/3]
>>> print_interval('x1 a1', (4, 2))
[0, 1/8]
>>> print_interval('x1 a2 a1 a2', (2, 1))
[5/8, 3/4]
>>> print_interval('x1 a2 a1 a2', (3, 1))
[10/27, 11/27]
>>> print_interval('x1 a2 a1 a2', (4, 2))
[17/128, 9/64]
```

```
>>> from thompson.examples import random_simple_word
>>> s, e = random_simple_word().as_interval(); 0 <= s < e <= 1
True
```

Raises **ValueError** – if the word *is not simple*.

**alpha** (*index*)

Let *w* stand for the current word. This method creates and returns a new word  $w\alpha_{\text{index}}$ .

```
>>> Word("x a1 a2", (3, 2)).alpha(3)
Word('x1 a1 a2 a3', (3, 2))
```

Raises **IndexError** – if *index* is not in the range 1... *arity*.

**extend** (*tail*)

Concatenates the current word with the series of letters *tail* to form a new word. The argument *tail* can be given as either a string or a tuple of integers.

```
>>> Word('x1 a2 a1', (2, 1)).extend('a2 a2')
Word('x1 a2 a1 a2 a2', (2, 1))
>>> Word('x1 a2 a1', (2, 1)).extend('x1 a2 a2')
Traceback (most recent call last):
...
ValueError: Word is invalid: valency is 2 (should be 1).
>>> Word('x1 a2 a1', (2, 1)).extend('x1 a2 a2 L')
Word('x1 a2', (2, 1))
```

**expand()**

Returns an iterator that yields the *arity* descendants of this word.

```
>>> w = Word("x a1", (5, 1))
>>> for child in w.expand():
...     print(child)
x1 a1 a1
x1 a1 a2
x1 a1 a3
x1 a1 a4
x1 a1 a5
>>> w = Word("x a1 a1 x a2 a1 L", (2, 1))
>>> for child in w.expand():
...     print(child)
x1 a1 a1
x1 a2 a1
```

**Return type** iterator which yields *Words*.

**split** (*head\_width*)

Splits the current word *w* into a pair of tuples *head*, *tail* where  $\text{len}(\text{head}) == \text{head\_width}$ . The segments of the word are returned as tuples of integers, i.e. not fully fledged *Words*.

**Raises `IndexError`** – if *head\_width* is outside of the range  $0 \leq \text{head\_width} \leq \text{len}(w)$ .

```
>>> Word('x1 a1 a2 a3 a1 a2', (3, 1)).split(4)
((1, -1, -2, -3), (-1, -2))
>>> Word('x1 a1 a2 a3 a1 a2', (3, 1)).split(10)
Traceback (most recent call last):
...
IndexError: The head width 10 is not in the range 0 to 6.
```

**rsplit** (*tail\_width*)

The same as *split()*, but this time *tail\_width* counts from the right hand side. This means that  $\text{len}(\text{tail}) == \text{tail\_width}$ .

**Raises `IndexError`** – if *tail\_width* is outside of the range  $0 \leq \text{tail\_width} \leq \text{len}(w)$ .

```
>>> Word('x1 a1 a2 a3 a1 a2', (3, 1)).rsplit(4)
((1, -1), (-2, -3, -1, -2))
>>> Word('x1 a1 a2 a3 a1 a2', (3, 1)).rsplit(10)
Traceback (most recent call last):
...
IndexError: The tail width 10 is not in the range 0 to 6.
```

**subwords()**

todo docstring

### 3.3.5 Next steps

Now that we can work with words in  $V_{n,r}$ , we need to be able to work with *collections* of words. The *generators* module will let us treat a list of words as a generating set and examine the subalgebra that the collection generates.

## 3.4 Generating sets and bases

Let  $S$  be a set of words in  $V_{n,r}$ . We can obtain new words by applying the operations  $\alpha_i$  and  $\lambda$  to elements of  $S$ . Next, we can apply the operations to the new words we've received. This process continues indefinitely, producing a set of words  $T$  whose elements are sourced from the original set  $S$ . We call  $T$  the *subalgebra generated by  $S$*  and say that  $S$  *generates  $T$* .

Of particular interest are *bases*, which we can think of as generating sets which do not contain any redundant elements. For example, it would be pointless to include each of  $x$ ,  $x\alpha_1$  and  $x\alpha_2$  in  $S$ , as the first can be obtained from the last two and vice versa.

### 3.4.1 The Generators class

**class** thompson.generators.**Generators** (*signature*, *generators=None*)

An ordered subset of  $V_{n,r}$ , together with methods which treat such sets as generating sets and bases. Internally, this is a subclass of `list`, so it is possible to reorder and otherwise modify collections of `Generators`.

**Variables** `signature` – The *Signature* of the algebra this set belongs to.

`__init__` (*signature*, *generators=None*)

When creating a generating set, you must specify the algebra  $V_{n,r}$  it is a subset of. A list of generators can optionally be provided. Each generator is passed to `append()`.

`append` ( $w$ )

Adds  $w$  to this generating set. If  $w$  is a string, a `Word` object is created and assigned the same *arity* and *alphabet\_size* as the generating set.

**Raises**

- **TypeError** – if  $w$  is neither a string nor a `Word`.
- **IndexError** – if  $w$  has a different arity to the generating set.
- **IndexError** – if  $w$  has a larger *alphabet\_size* the generating set.
- **ValueError** – if  $w$  is already contained in this generating set.

`copy` ()

We override `list.copy()` so that we don't have to recreate `Generators` instances by hand.

```
>>> olga_f = load_example('olga_f')
>>> X = olga_f.quasinormal_basis.copy()
>>> X is olga_f.quasinormal_basis
False
>>> type(X).__name__
'Generators'
```

**filter** (*func*)

Creates a copy of the current generating set whose elements  $x$  are those for which  $func(x)$  is `True`. The original generating set is unmodified, and the order of elements is inherited by the filtered copy.

```
>>> X = load_example('olga_f').quasinormal_basis
>>> print(X)
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2 a1, x1 a2 a2 a2]
>>> def test(x):
...     return len(x) % 2 == 0
...
>>> print(X.filter(test))
[x1 a2 a2 a1, x1 a2 a2 a2]
```

**test\_free()**

Tests to see if the current generating set  $X$  is a free generating set. To do so, the words contained in  $X$  must all be *simple*.

If the test fails, returns the first pair of indices  $(i, j)$  found for which  $X_i$  is above  $X_j$ . If the test passes, returns  $(-1, -1)$ .

```
>>> g = Generators((2, 3), ["x1 a1", "x1 a2 a1", "x1 a2 a1 a1", "x1 a2 a2"])
>>> g.test_free()
(1, 2)
>>> print(g[1], g[2], sep='\n')
x1 a2 a1
x1 a2 a1 a1
>>> g = Generators((2, 1), ['x1 a2 a2', 'x1 a2 a1 x1 a2 a2 a2 a1 L x1 a1 L'])
>>> g.test_free()
Traceback (most recent call last):
...
ValueError: Cannot test for freeness unless all words are simple.
```

Raises **ValueError** – if any word in the basis is not simple.

**See also:**

Lemma 3.16 of the paper.

**is\_free()**

Returns True if this is a free generating set; otherwise False.

```
>>> g = Generators((2, 3), ['x1', 'x2']);
>>> g.is_free()
True
>>> g.append('x3'); g.is_free()
True
>>> g.append('x2 a1'); g.is_free()
False
```

**preorder\_traversal()**

Yields words as if traversing the vertices of a tree in pre-order.

**simple\_words\_above()**

An iterator that yields the simple words which can be obtained by contracting this basis  $n \geq 0$  times. (So the ‘above’ condition is not strict.)

```
>>> g = Generators((3, 1), ["x a1 a1 a3", "x a1 a1 a1", "x a1 a2", "x a1 a1 a2", "x a1 a3"])
>>> for word in g.simple_words_above():
...     print(format(word))
x1 a1 a1 a1
x1 a1 a1 a2
x1 a1 a1 a3
x1 a1 a2
x1 a1 a3
x1 a1 a1
x1 a1
```

**test\_generates\_algebra()**

Tests to see if this set generates all of  $V_{n,r}$ . The generating set is sorted, and then contracted as much as possible. Then we check to see which elements of the *standard basis*  $x = \{x_1, \dots, x_r\}$  are not included in the contraction.

The test fails if there is at least one such element; in this case, the list of all such elements is returned. Otherwise the test passes, and an empty list is returned.

```
>>> #A basis for V_2,1
>>> Y = Generators((2, 1), ["x a1", "x a2 a1", "x a2 a2 a1 a1", "x a2 a2 a1 a2", "x a2 a2 a2 a1 a1"])
>>> Y.test_generates_algebra()
[]
>>> #The same words do not generate V_2,3
>>> Y = Generators((2, 3), ["x a1", "x a2 a1", "x a2 a2 a1 a1", "x a2 a2 a1 a2", "x a2 a2 a2 a1 a1"])
>>> missing = Y.test_generates_algebra(); missing
[Word('x2', (2, 3)), Word('x3', (2, 3))]
>>> for x in missing: print(x)
x2
x3
```

**Return type** a list of *Words*.

#### **generates\_algebra()**

Returns True if this set generates all of  $V_{n,r}$ . Otherwise returns False.

```
>>> from random import randint
>>> arity = randint(2, 5); alphabet_size = randint(2, 10)
>>> Y = Generators.standard_basis((arity, alphabet_size))
>>> Y.generates_algebra()
True
>>> random_basis().generates_algebra()
True
```

#### **test\_above(word, return\_index=False)**

Searches for a pair (gen, tail) where gen belongs to the current basis and gen + tail = word. If no such pair exists, returns None.

If *return\_index* is False, returns the pair (gen, tail). Otherwise, returns the pair (i, tail) where *i* is the index of *gen* in the current basis.

```
>>> basis = Generators.standard_basis((2, 1)).expand(0).expand(0).expand(0)
>>> basis
Generators((2, 1), ['x1 a1 a1 a1', 'x1 a1 a1 a2', 'x1 a1 a2', 'x1 a2'])
>>> gen, tail = basis.test_above(Word('x1 a2 a2 a1', (2, 1)))
>>> print(gen, '|', format(tail))
x1 a2 | a2 a1
>>> i, tail = basis.test_above(Word('x1 a2 a2 a1', (2, 1)), return_index=True)
>>> print(basis[i])
x1 a2
```

#### **is\_above(word)**

Returns True if any generator *is\_above()* the given *word*.

```
>>> g = Generators((2, 2), ['x1 a1', 'x1 a2', 'x2'])
>>> g.is_above(Word('x1 a1 a1 a2', (2, 2)))
True
>>> g.is_above(Word('x1', (2, 2)))
False
>>> g.is_above(Word('x2', (2, 2)))
True
>>> g.is_above(Word('x1 a1 x1 a2 L', (2, 2)))
False
```

#### **is\_basis()**

Returns True if this is a *free generating set* which generates all of  $V_{n,r}$ . Otherwise returns False.

```
>>> g = Generators((2, 2), ["x1 a1", "x1 a2"])
>>> g.is_free()
True
>>> g.generates_algebra()
False
>>> g.is_basis()
False
>>> g.append('x2')
>>> g.is_basis()
True
>>> random_basis().is_basis()
True
```

#### **descendants\_above** (*floor*)

Suppose we have a basis *floor* below the current basis *ceiling*. There are a finite number of elements below *ceiling* which are not below *floor*. This method enumerates them. In symbols, we are enumerating the set  $X\langle A \rangle \setminus F\langle A \rangle$ , where  $X$  is the current basis and  $F$  is the *floor*.

```
>>> phi = load_example('example_5_15')
>>> X = phi.quasinormal_basis
>>> Y = X.minimal_expansion_for(phi)
>>> Z = phi.image_of_set(Y)
>>> terminal = X.descendants_above(Y)
>>> initial = X.descendants_above(Z)
>>> print(initial, terminal, sep='\n')
[x1 a1, x1 a1 a1]
[x1 a2 a2, x1 a2 a2 a1]
```

#### **classmethod standard\_basis** (*signature*)

Creates the standard basis  $x = \{x_1, \dots, x_r\}$  for  $V_{n,r}$ , where  $n$  is the arity and  $r$  is the *alphabet\_size* of the *signature*.

```
>>> Generators.standard_basis((2, 4))
Generators((2, 4), ['x1', 'x2', 'x3', 'x4'])
```

#### **minimal\_expansion\_for** (\**automorphisms*)

Suppose we are given a finite sequence of *automorphisms* of  $V_{n,r}$  and that the current generating set is a basis  $X$  for  $V_{n,r}$ . This method returns an expansion  $Y$  of  $X$  such that each automorphism maps  $Y$  into  $X\langle A \rangle$ .

```
>>> std_basis = Generators.standard_basis((2, 1))
>>> reduced_domain = Generators((2, 1), ["x a1 a1", "x a1 a2 a1", "x a1 a2 a2", "x a2 a1", "x a2 a2"])
>>> std_basis.minimal_expansion_for(load_example('cyclic_order_six')) == reduced_domain
True
```

```
>>> phi = load_example('example_5_3')
>>> basis = phi.quasinormal_basis
>>> print(basis.minimal_expansion_for(phi))
[x1 a1 a1 a1 a1, x1 a1 a1 a1 a2, x1 a1 a1 a2, x1 a1 a2 a1, x1 a1 a2 a2, x1 a2 a1 a1, x1 a2 a2 a1]
```

#### **Raises**

- **ValueError** – if no automorphisms are passed to this method.
- **ValueError** – if the automorphisms or basis do not all belong to the same group  $G_{n,r}$ .
- **ValueError** – if the given basis does not generate  $V_{n,r}$ .

**See also:**

Lemma 4.3 of the paper proves that this expansion exists, is of minimal size, and is unique with this that property.

**expand** (*index*)

Replaces the word  $w$  at index *index* in the current generating set with its children,  $w\alpha_1, \dots, w\alpha_n$ , where  $n$  is the arity of the generating set. As with ordinary Python lists, negative values of *index* are supported.

```
>>> g = Generators.standard_basis((3, 1)); g
Generators((3, 1), ['x1'])
>>> g.expand(0)
Generators((3, 1), ['x1 a1', 'x1 a2', 'x1 a3'])
>>> g.expand(-2) #expand at the second entry from the right, i.e. at 'x1 a2'
Generators((3, 1), ['x1 a1', 'x1 a2 a1', 'x1 a2 a2', 'x1 a2 a3', 'x1 a3'])
```

**Raises** `IndexError` – if there is no generator at index *index*.

**Returns** the current generating set, after modification.

**classmethod sort\_mapping\_pair** (*domain, range*)

Makes copies of the given lists of words *domain* and *range*. The copy of *domain* is sorted according to the `order` defined on words. The same re-ordering is applied to the *range*, so that the mapping from *domain* to *range* is preserved.

**Return type** A pair of class:*Generator* instances.

**expand\_to\_size** (*size*)

Expands the current generating set until it has the given *size*. The expansions begin from the end of the generating set and work leftwards, wrapping around if we reach the start. (This is to try and avoid creating long words where possible.)

```
>>> basis = Generators.standard_basis((3, 1)); print(basis)
[x1]
>>> basis.expand_to_size(11); print(basis)
[x1 a1 a1, x1 a1 a2, x1 a1 a3, x1 a2 a1, x1 a2 a2, x1 a2 a3, x1 a3 a1, x1 a3 a2, x1 a3 a3 a1]
>>> basis = Generators.standard_basis((2, 1)); print(basis)
[x1]
>>> basis.expand_to_size(2); print(basis)
[x1 a1, x1 a2]
>>> basis = Generators.standard_basis((2, 3)).expand(2).expand(0); print(basis)
[x1 a1, x1 a2, x2, x3 a1, x3 a2]
>>> basis.expand_to_size(12); print(basis)
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2, x2 a1, x2 a2, x3 a1 a1, x3 a1 a2, x3 a2 a1 a1, x3 a2 a1 a2, x3 a2 a1 a3]
```

**Returns** the current generating set

**Raises** `ValueError` – if an expansion to the given size is not possible.

**cycled** (*positions=1*)

Produces a copy of the current generating set, cyclicly shifted by a number of *positions* to the right.

```
>>> phi = load_example('example_5_15')
>>> print(phi.domain)
[x1 a1, x1 a2 a1, x1 a2 a2 a1 a1, x1 a2 a2 a1 a2, x1 a2 a2 a2]
>>> print(phi.domain.cycled(2))
[x1 a2 a2 a1 a1, x1 a2 a2 a1 a2, x1 a2 a2 a2, x1 a1, x1 a2 a1]
```

```
>>> X = random_basis()
>>> X == X.cycled(0) == X.cycled(len(X))
True
```

`thompson.generators.rewrite_set` (*set*, *basis*, *new\_basis*)

Maps *set* into the algebra generated by *new\_basis* according to the bijection between *basis* and *new\_basis*. See also `rewrite_word()`.

`thompson.generators.rewrite_word` (*word*, *basis*, *new\_basis*)

Suppose that we have a *word* which is below a given *basis*. Suppose we have a bijection between *basis* and some image *new\_basis*. Then we can rewrite *word* as a descendant of *new\_basis* in a manner which is compatible with this bijection.

#### Raises

- **ValueError** – if *basis* is not above *word*.
- **ValueError** – if *basis* and *new\_basis* have different sizes.

## 3.4.2 Next steps

Suppose we have a map  $f: V_{n,r} \rightarrow V_{n',r'}$ . If this is just a set map then we would have to specify the image of every word in  $V_{n,r}$ . However, if  $f$  preserves structure then it is sufficient to specify the image of a generating set—or even better, a basis—for  $V_{n,r}$ . Such maps  $f$  are called *homomorphisms*. The next module describes homomorphisms in terms of the image of a given basis.

## 3.5 Homomorphisms

The algebra of *words* has its own structure, and just like groups, rings etc. there exist homomorphisms which preserve this structure. In our specific case, a homomorphism  $\phi: V_{n,r} \rightarrow V_{n,s}$  is a function satisfying

$$w\alpha_i\phi = w\phi\alpha_i \quad \text{and} \quad w_1 \dots w_n \lambda \phi = a w_1 \phi \dots w_n \phi \lambda.$$

In other words, a homomorphism is a function which ‘commutes’ with the algebra operations  $\alpha_i$  and  $\lambda$ .

### 3.5.1 The Homomorphism Class

**class** `thompson.homomorphism.Homomorphism` (*domain*, *range*, *reduce=True*)

Let  $f: D \rightarrow R$  be some map embedding a basis  $D$  for  $V_{n,r}$  into another algebra  $V_{n,s}$  of the same *Signature*.

This map uniquely extends to a homomorphism of algebras  $\psi: V_{n,r} \rightarrow V_{n,s}$ .

#### Variables

- **domain** – a *basis* of preimages
- **range** – a *basis* of images.

The next few attributes are used internally when constructing free factors. We need to have a way to map back to the parent automorphisms.

#### Variables

- **domain\_relabeller** – the homomorphism which maps relabelled words back into the original algebra that this automorphism came from.
- **range\_relabeller** – the same.

`__init__` (*domain*, *range*, *reduce=True*)

Creates a homomorphism with the specified *domain* and *range*. Sanity checks are performed so that the arguments do genuinely define a basis.

By default, any redundancy in the mapping is reduced. For example, the rules  $x_1 a_1 a_1 \rightarrow x_1 a_2 a_1$  and  $x_1 a_1 a_2 \rightarrow x_1 a_2 a_2$  reduce to the single rule  $x_1 a_1 \rightarrow x_1 a_2$ . The *reduce* argument can be used to disable this. This option is intended for internal use only.<sup>1</sup>

#### Raises

- **TypeError** – if the bases are of different sizes.
- **TypeError** – if the algebras described by *domain* and *range* have different arities.
- **ValueError** – if *domain* is *not a basis*.

**classmethod** `from_file` (*filename*)

Reads in a file specifying a homomorphism and returns the homomorphism being described. Here is an example of the format:

```
5
(2,1)          ->    (2,1)
x1 a1 a1 a1    ->    x1 a1 a1
x1 a1 a1 a2    ->    x1 a1 a2 a1
x1 a1 a2       ->    x1 a1 a2 a2
x1 a2 a1       ->    x1 a2 a2
x1 a2 a2       ->    x1 a2 a1
```

This is an example intended to illustrate the format for reading and writing automorphisms t

There are three components

- number *k* of generators in domain and range
- signatures of domain and range
- list of *k* rules domain  $\rightarrow$  range

Any lines after this are ignored, and can be treated as comments. Comments are read in and added to the `__doc__` attribute of the homomorphism that gets created.

**classmethod** `from_string` (*string*)

An alternative `from_file()` for working with examples that you might want to copy and paste somewhere. *string* should be a Python string which describes an automorphism in the same format as in `from_file()`.

**save\_to\_file** (*filename=None*, *comment=None*)

Takes a homomorphism and saves it to the file with the given *filename*. The homomorphism is stored in a format which is compatible with `from_file()`. Optionally, a *comment* may be appended to the end of the homomorphism file.

```
>>> before = random_automorphism()
>>> before.save_to_file('test_saving_homomorphism.aut')
>>> after = Automorphism.from_file('test_saving_homomorphism.aut')
>>> before == after, before is after
(True, False)
```

`__eq__` (*other*)

We can test for equality of homomorphisms by using Python's equality operator `==`.

<sup>1</sup> Sometimes we have to expand *free factors* so that the orbit sizes match up.

```

>>> phi = random_automorphism()
>>> phi == phi
True
>>> phi * ~phi == Homomorphism.identity(phi.signature)
True

```

**\_\_mul\_\_** (*other*)

If the current automorphism is  $\psi$  and the *other* is  $\phi$ , multiplication forms the composition  $\psi\phi$ , which maps  $x \mapsto x\psi \mapsto (x\psi)\phi$ .

**Raises `TypeError`** – if the homomorphisms cannot be composed in the given order; i.e. if `self.range.signature != other.domain.signature`.

**Return type** an *MixedAut* if possible; otherwise a *Homomorphism*.

```

>>> phi = load_example('alphabet_size_two')
>>> print(phi * phi)
MixedAut: V(3, 2) -> V(3, 2) specified by 8 generators (after expansion and reduction).
x1 a1 -> x1 a1
x1 a2 -> x1 a2 a3 a3
x1 a3 a1 -> x1 a3
x1 a3 a2 -> x1 a2 a2
x1 a3 a3 -> x1 a2 a1
x2 a1 -> x2
x2 a2 -> x1 a2 a3 a2
x2 a3 -> x1 a2 a3 a1

```

**classmethod `identity`** (*signature*)

Creates a new homo/automorphism which is the identity map on the algebra with the given *signature*.

```

>>> print(Homomorphism.identity((3, 2)))
PeriodicAut: V(3, 2) -> V(3, 2) specified by 2 generators (after expansion and reduction).
x1 -> x1
x2 -> x2

```

**`is_identity`** ()

Returns True if this automorphism is the identity map on the algebra with the given *signature*. Otherwise returns False.

```

>>> aut = Homomorphism.identity(random_signature())
>>> aut.is_identity()
True
>>> load_example('example_5_15').is_identity()
False

```

**`image`** (*key*, *sig\_in=None*, *sig\_out=None*, *cache=None*)

Computes the image of a *key* under the given homomorphism. The result is cached for further usage later.

The *key* must be given as one of:

- a string to be passed to `from_string()`;
- a sequence of integers (see the `word` module); or
- a `Word` instance.

---

**Note:** All other arguments are intended for internal use only. <sup>2</sup>

---

<sup>2</sup> Exposing more arguments means I can write this function in a more general manner. Doing so makes it easy to compute *inverse images* under an *MixedAut*.

The input need not be in standard form. This method

1. Converts *key* to standard form if necessary.
2. Checks if the image of the standard form of *key* has been cached, and returns the image if so.
3. Computes the image of *key* under the homomorphism, then caches and returns the result.

```
>>> #An element of the domain---just a lookup
>>> phi = load_example('example_5_15')
>>> print(phi.image('x1 a1'))
x1 a1 a1 a1
>>> #A word below a the domain words
>>> print(phi.image('x1 a1 a2 a2'))
x1 a1 a1 a1 a2 a2
>>> #Above domain words---have to expand.
>>> print(phi.image('x1'))
x1 a1 a1 a1 x1 a1 a1 a2 x1 a2 a2 x1 a1 a2 L x1 a2 a1 L L L
>>> #Let's try some words not in standard form
>>> print(phi.image('x1 a1 a1 x1 a1 a2 L'))
x1 a1 a1 a1
>>> print(phi.image('x1 a1 a1 x1 a1 a2 L a2 a1'))
x1 a1 a1 a1 a2 a1
>>> print(phi.image('x1 a1 a1 x1 a2 a2 L'))
x1 a1 a1 a1 a1 x1 a2 a2 x1 a1 a2 L x1 a2 a1 L L
```

**Return type** a *Word* instance (which are always in standard form).

**image\_of\_set** (*set*, *sig\_in=None*, *sig\_out=None*, *cache=None*)

Computes the image of a list of *Generators* under the current homomorphism. The order of words in the list is preserved.

**Return type** another list of *Generators*.

```
>>> basis = Generators.standard_basis((2,1))
>>> basis.expand_to_size(8); print(basis)
[x1 a1 a1 a1, x1 a1 a1 a2, x1 a1 a2 a1, x1 a1 a2 a2, x1 a2 a1 a1, x1 a2 a1 a2, x1 a2 a2 a1,
>>> print(load_example('example_5_3').image_of_set(basis))
[x1 a1 a1 a1 x1 a1 a1 a2 a1 L, x1 a1 a1 a2 a2, x1 a1 a2 a2, x1 a1 a2 a1, x1 a2 a1 a1 a1, x1
```

**\_\_str\_\_** ()

Printing an automorphism gives its arity, *alphabet\_size*, and lists the images of its domain elements.

```
>>> print(load_example('cyclic_order_six'))
PeriodicAut: V(2, 1) -> V(2, 1) specified by 5 generators (after expansion and reduction).
x1 a1 a1 -> x1 a1 a1
x1 a1 a2 a1 -> x1 a1 a2 a2 a2
x1 a1 a2 a2 -> x1 a2
x1 a2 a1 -> x1 a1 a2 a2 a1
x1 a2 a2 -> x1 a1 a2 a1
```

**add\_relabellers** (*domain\_relabeller*, *range\_relabeller*)

**Raises**

- **LookupError** – if a relabeller is provided and the relabeller’s signature does not match that of *domain* or *range*
- **TypeError** – if a relabeller is provided and the relabeller’s signature does not match that of *domain* or *range* (as appropriate)

**relabel()**

If this automorphism was derived from a parent automorphism, this converts back to the parent algebra after doing computations in the derived algebra.

In the following example `test_conjugate_to()` takes a pure periodic automorphism and extracts factors. A conjugator  $\rho$  is produced by *the overridden version of this method*. Finally  $\rho$  is relabelled back to the parent algebra.

**Raises `AttributeError`** – if the factor has not been assigned any relabellers.

```
>>> psi, phi = load_example_pair('example_5_12')
>>> rho = psi.test_conjugate_to(phi)
>>> print(rho)
PeriodicAut: V(2, 1) -> V(2, 1) specified by 6 generators (after expansion and reduction).
x1 a1 a1 a1 a1 -> x1 a1 a2
x1 a1 a1 a1 a2 -> x1 a2 a2
x1 a1 a1 a2    -> x1 a1 a1 a1
x1 a1 a2      -> x1 a2 a1 a1
x1 a2 a1      -> x1 a1 a1 a2
x1 a2 a2      -> x1 a2 a1 a2
>>> rho * phi == psi * rho
True
```

### 3.5.2 Next steps

It is not possible to repeatedly apply a homomorphism  $\psi$  to a word unless  $\psi$  is actually an automorphism. The next class extends `Homomorphism` to represent an `Automorphism`.

## 3.6 Automorphisms

As usual, a bijective homomorphism is called an *isomorphism*, and an isomorphism from an algebra to itself is called an *automorphism*. A collection of automorphisms of an object  $O$  forms a group  $\text{Aut}(O)$  under composition. The group of automorphisms  $\text{Aut}(V_{n,r})$  is known as  $G_{n,r}$ .

We consider three different classes of automorphism; this module implements functionality common to each class.

### 3.6.1 The Automorphisms class

**class** thompson.automorphism.**Automorphism**(*domain, range, reduce=True*)

Represents an automorphism as a bijection between *bases*.

Generic attributes:

#### Variables

- **signature** – The *Signature* shared by the generating sets domain and range.
- **quasinormal\_basis** – See `compute_quasinormal_basis()`.
- **pond\_banks** – A list of tuples  $(\ell, k, r)$  such that  $(\ell, r)$  are banks of a pond with  $\ell\phi^k = r$ .

For instance:

```
>>> olga_f = load_example('olga_f')
>>> olga_g = load_example('olga_g')
>>> print(len(olga_f.pond_banks), sep=', ') #No ponds
0
```

```
>>> print(len(olga_g.pond_banks)) #One pond
1
>>> print(*olga_g.pond_banks[0], sep=', ')
x1 a2 a1 a2 a1 a1, 2, x1 a1 a2 a2
```

Periodic attributes:

#### Variables

- **periodic\_orbits** – a mapping  $d \mapsto L_d$ , where  $L_d$  is the list of size  $d$  orbits in the quasinormal basis.
- **multiplicity** – a mapping  $d \mapsto m_\phi(d, X_\phi)$ , which is the number of periodic orbits of size  $d$  in the *quasinormal basis* for  $\phi$ . See Definition 5.8 in the paper.
- **cycle\_type** – the set  $\{d \in \mathbb{N} : \exists \text{ an orbit of length } d.\}$
- **order** – The *lcm()* of the automorphism’s cycle type. This is the group-theoretic order of the *periodic factor* of  $\phi$ . If the cycle type is empty, the order is  $\infty$ .

```
>>> def display_orbits(orbitals_by_size):
...     for key in sorted(orbitals_by_size):
...         print('Orbits of length', key)
...         for list in orbitals_by_size[key]:
...             print('...', *list, sep=' ->', end=' -> ...\n')
>>> display_orbits(load_example('example_5_9').periodic_orbits)
Orbits of length 2
... -> x1 a2 a1 a1 -> x1 a2 a1 a2 -> ...
... -> x1 a2 a2 a1 -> x1 a2 a2 a2 -> ...
Orbits of length 3
... -> x1 a1 a1 a1 -> x1 a1 a1 a2 -> x1 a1 a2 -> ...
```

---

**Note:** *mixed automorphisms* will have a **finite** order, despite being infinite-order group elements.

---

Infinite attributes:

**Variables characteristics** – the set of characteristics  $(m, \Gamma)$  of this automorphism.

**\_\_init\_\_** (*domain*, *range*, *reduce=True*)

Creates an automorphism mapping the given *domain* basis to the *range* basis in the given order.

$$\text{domain}_i \mapsto \text{range}_i$$

The *quasi-normal basis*  $X$  and the various attributes are all calculated at creation time.

#### Raises

- **TypeError** – if the bases have different arities or alphabet sizes.
- **TypeError** – if either basis *isn't actually a basis*.

**See also:**

*The superclass method.*

**image** (*key*, *inverse=False*)

If *inverse* is *True*, the inverse of the current automorphism is used to map *key* instead. Otherwise this method delegates to *Homomorphism.image*.

Examples of finding inverse images:

```

>>> phi = load_example('example_5_15')
>>> print(phi.image('x1 a2 a2', inverse=True))
x1 a2 a2 a1 a1
>>> print(phi.image('x1 a1 a1 a2 a2 a1', inverse=True))
x1 a2 a1 a2 a1
>>> print(phi.image('x a2', inverse=True))
x1 a2 a2 a2 x1 a2 a2 a1 a1 L
>>> print(phi.image('x a2 a2 x a1 a2 L', inverse=True))
x1 a2 a2 a1

```

**image\_of\_set** (*set*, *inverse=False*)

If *inverse* is True, the inverse of the current automorphism is used to map *set* instead. Otherwise this method delegates to *Homomorphism.image\_of\_set*.

Examples of finding inverse images:

```

>>> basis = Generators.standard_basis((2,1))
>>> basis.expand_to_size(8);
>>> print(load_example('example_5_3').image_of_set(basis, inverse=True))
[x1 a1 a1 a1 a1, x1 a1 a1 a1 a2 x1 a1 a1 a2 L, x1 a1 a2 a2, x1 a1 a2 a1, x1 a2 a1, x1 a2 a2

```

**repeated\_image** (*key*, *power*)

If  $\psi$  is the current automorphism, returns  $\text{key}\psi^{\text{power}}$ .

**Return type** a *Word* instance.

```

>>> phi = load_example('example_5_15')
>>> print(phi.repeated_image('x1 a1', 10))
x1 a1 a1
>>> print(phi.repeated_image('x1 a1 a1 a1 a1 a1 a1 a1', -3))
x1 a1
>>> phi = load_example('arity_four')
>>> print(phi.repeated_image('x1 a4 a4 a2', 4))
x1 a3 a3 a2
>>> print(phi.repeated_image('x1 a3 a3 a2', -4))
x1 a4 a4 a2

```

**Todo**

This could be made more efficient for large values of *power* by using knowledge of the component containing *key*.

**\_\_pow\_\_** (*power*)

```

>>> psi = random_automorphism()
>>> ~psi ** 2 == ~psi * ~psi == psi ** -2
True

```

**\_\_invert\_\_** ()

We overload python's unary negation operator  $\sim$  as shorthand for inversion. (In Python,  $\sim$  is normally used for bitwise negation.) We can also call a method explicitly: `phi.inverse()` is exactly the same as `~phi`.

```

>>> phi = random_automorphism()
>>> phi * ~phi == ~phi * phi
True
>>> (phi * ~phi).is_identity()

```

```
True
>>> (~phi).quasinormal_basis == phi.quasinormal_basis
True
```

**compute\_quasinormal\_basis()**

We say that  $\phi$  is *in semi-normal form* with respect to the basis  $X$  if no element of  $X$  lies in an incomplete  $X$ -component of a  $\phi$  orbit. See the *orbits* module for more details.

There is a minimal such basis,  $X_\phi$  say, and we say that  $\phi$  is *in quasi-normal form* with respect to  $X_\phi$ . This method determines and returns the basis  $X_\phi$  where  $\phi$  denotes the current automorphism. The result is cached so that further calls to this method perform no additional computation.

**Note:** This method is called automatically at creation time and does **not** need to be called by the user.

```
>>> for name in ['example_4_5', 'alphabet_size_two', 'example_5_12_phi', 'example_6_2', 'example_6_3']
...     print(load_example(name).quasinormal_basis)
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2]
[x1 a1, x1 a2, x1 a3, x2]
[x1 a1, x1 a2]
[x1 a1 a1, x1 a1 a2, x1 a2]
[x1 a1, x1 a2]
```

**Return type** a *Generators* instance.

**See also:**

Quasi-normal forms are introduced in Section 4.2 of the paper. In particular, this method implements Lemma 4.28. Higman first described the idea of quasi-normal forms in Section 9 of [Hig74].

**seminormal\_form\_start\_point()**

Returns the minimal expansion  $X$  of  $x$  such that every element of  $X$  belongs to either *self.domain* or *self.range*. Put differently, this is the minimal expansion of  $x$  which does not contain any elements which are above  $Y \cup Z$ . This basis that this method produces is the smallest possible which *might* be semi-normal.

```
>>> for name in ['example_4_5', 'example_4_11', 'example_4_12', 'example_5_15', 'example_6_2', 'example_6_3']
...     print(load_example(name).seminormal_form_start_point())
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2]
[x1 a1, x1 a2]
[x1 a1, x1 a2]
[x1 a1, x1 a2 a1, x1 a2 a2]
[x1 a1 a1, x1 a1 a2 a1, x1 a1 a2 a2, x1 a2]
```

**See also:**

Remark 4.10 of the paper.

**orbit\_type**( $y$ , *basis=None*)

Returns the orbit type of  $y$  with respect to the given *basis*. If *basis* is omitted, the *quasi-normal basis* is used by default. Also returns a dictionary of computed images, the list (11) from the paper.

```
>>> phi = load_example('example_4_5')
>>> print_component_types(phi, phi.domain)
x1 a1 a1 a1: Left semi-infinite component with characteristic (-1, a1)
x1 a1 a1 a2: Bi-infinite component
x1 a1 a2: Right semi-infinite component with characteristic (1, a2)
x1 a2 a1: Periodic component of order 2
x1 a2 a2: Periodic component of order 2
with respect to the basis [x1 a1 a1 a1, x1 a1 a1 a2, x1 a1 a2, x1 a2 a1, x1 a2 a2]
```

```
>>> print_component_types(phi, basis=phi.domain, words=['x', 'x a1', 'x a2'])
x1: Incomplete component
x1 a1: Incomplete component
x1 a2: Incomplete component
with respect to the basis [x1 a1 a1 a1, x1 a1 a1 a2, x1 a1 a2, x1 a2 a1, x1 a2 a2]
```

Components can be calculated with respect to any *basis*, not just the *quasi-normal basis*.

```
>>> print(olga_f.quasinormal_basis)
[x1 a1 a2, x1 a1 a2, x1 a2 a1, x1 a2 a2 a1, x1 a2 a2 a2]
>>> basis = olga_f.quasinormal_basis.copy().expand(-1); print(basis)
[x1 a1 a1, x1 a1 a2, x1 a2 a1, x1 a2 a2 a1, x1 a2 a2 a2 a1, x1 a2 a2 a2 a2]
>>> from pprint import pprint
>>> print(olga_f.orbit_type('x a2 a2 a2')[0])
Bi-infinite component
>>> print(olga_f.orbit_type('x a2 a2 a2', basis)[0])
Incomplete component
```

#### See also:

Lemmas 4.18, 4.28 of the paper.

---

#### Todo

This should be renamed to `component_type`.

---

#### `test_semi_infinite` (*y*, *basis*, *backward=False*)

Computes the orbit type of *y* under the current automorphism  $\psi$  with respect to *basis* in the given direction. Let  $y\psi^m$  be the most recently computed image. The process stops when either:

1.  $y\psi^m$  is not below the *basis*, for some  $m \geq 0$ .
  - infinite: False
  - start: 0
  - end:  $m-1$
  - images:  $y, y\psi, \dots, y\psi^{m-1}$ .
2. For some  $0 \leq l < m$ ,  $y\psi^l$  is above (or equal to)  $y\psi^m$ .
  - infinite: True
  - start:  $l$
  - end:  $m$
  - images:  $y, y\psi, \dots, y\psi^m$ .

---

**Note:** The word  $y\psi^m$  is not strictly in the core part of the orbit of Lemma 4.24. We return this as part of *images* so that we can compute the characteristic multiplier in `orbit_type()`.

---

**Returns** the tuple (*infinite*, *start*, *end*, *images*).

#### See also:

Lemma 4.28 of the paper.

**semi\_infinite\_end\_points** (*exclude\_characteristics=False*)

Returns the list of terminal *Words* in left semi-infinite components and the list of initial words in right semi-infinite components. This is all computed with respect the current automorphism's *quasinormal basis*. These are the sets  $X\langle A \rangle \setminus Y\langle A \rangle$  and  $X\langle A \rangle \setminus Z\langle A \rangle$ .

**Parameters** `exclude_characteristics` – if True, only the endpoints of non-characteristic semi-infinite orbits will be returned.

```
>>> for id in ['4_5', '4_11', '4_12', '5_15', '6_2']:
...     aut = load_example('example_' + id)
...     print(*aut.semi_infinite_end_points())
[x1 a1 a1] [x1 a1 a2]
[x1 a1] [x1 a2]
[] []
[x1 a2 a2, x1 a2 a2 a1] [x1 a1, x1 a1 a1]
[x1 a1 a1] [x1 a2]
>>> phi = load_example('nathan_pond_example')
>>> print(*phi.semi_infinite_end_points())
[x1 a1 a1, x1 a1 a1 a1, x1 a1 a1 a2] [x1 a1 a2, x1 a1 a2 a1, x1 a1 a2 a2]
>>> print(*phi.semi_infinite_end_points(exclude_characteristics=True))
[x1 a1 a1 a2] [x1 a1 a2 a2]
```

**Return type** A pair of *Generators*.

**See also:**

The discussion before Lemma 4.6.

**dump\_QNB** ()

A convenience method for printing the quasinormal basis  $X$ . The  $X$ -components of the elements  $x \in X$  are displayed.

```
>>> load_example('example_5_3').dump_QNB()
x1 a1 a1 a1 Left semi-infinite component with characteristic (-1, a1)
x1 a1 a1 a2 Right semi-infinite component with characteristic (1, a2)
x1 a1 a2 a1 Periodic component of order 2
x1 a1 a2 a2 Periodic component of order 2
x1 a2 a1 Right semi-infinite component with characteristic (1, a1)
x1 a2 a2 Left semi-infinite component with characteristic (-1, a2)
```

**print\_characteristics** ()

For convenience, a method that prints out all of the characteristics of type B components wrt the quasinormal basis.

```
>>> for id in ['4_1', '5_15', '6_2', '6_8_phi']:
...     name = 'example_' + id
...     print(name)
...     load_example(name).print_characteristics()
example_4_1
(-1, a1)
(1, a2)
example_5_15
(-1, a1 a1)
(1, a1 a1)
example_6_2
(-2, a1)
(1, a2)
example_6_8_phi
(-1, a1 a1 a1)
```

```
(1, a2 a2 a2)
>>> (load_example('example_6_2')**2).print_characteristics()
(-1, a1)
(1, a2 a2)
>>> #Lemma 5.16
>>> psi, phi = random_conjugate_pair()
>>> psi.characteristics == phi.characteristics
True
```

**See also:**

Defintion 5.14.

**share\_orbit** (*u*, *v*)

Determines if *u* and *v* are in the same orbit of the current automorphism  $\psi$ . Specifically, does there exist an integer  $m$  such that  $u\psi^m = v$ ?

```
>>> phi = load_example('alphabet_size_two')
>>> u = Word('x1 a2 a3 a1 a2', (3, 2))
>>> v1 = Word('x1 a1 a2 a2 a3 a1', (3, 2))
>>> v2 = Word('x2 a3 a2', (3, 2))
>>> print(phi.share_orbit(u, v1))
{}
>>> print(phi.share_orbit(u, v2))
{-2}
>>> print(phi.share_orbit(u, u))
{0}
```

**Returns** The (possibly empty) *SolutionSet* of all integers  $m$  for which  $u\psi^m = v$ . Note that if  $u = v$  this method returns  $\mathbb{Z}$ .

**See also:**

The implementation is due to lemma 4.34 of the paper.

**is\_conjugate\_to** (*other*)

A shortcut for `self.test_conjugate_to(other)` is not None.

**preserves\_order** ()

Returns True if this automorphism is an element of  $F_{n,r}$ , Higman's analogue of Thompson's group  $F$ . Otherwise returns False.

```
>>> random_automorphism(group='F').preserves_order()
True
>>> phi = random_automorphism(group='T')
>>> #phi preserves order iff it is in F.
>>> (sorted(phi.range) == phi.range) == phi.preserves_order()
True
>>> load_example('nathan_pond_example').preserves_order()
False
```

**cycles\_order** ()

Returns True if this automorphism is an element of  $T_{n,r}$ , Higman's analogue of Thompson's group  $T$ . Otherwise returns False.

```
>>> random_automorphism(group='F').cycles_order()
True
>>> random_automorphism(group='T').cycles_order()
True
```

```
>>> load_example('nathan_pond_example').cycles_order() # in  $V \setminus T$ 
False
```

**test\_revealing** (*domain*='wrt QNB')

Determines if the given automorphism  $\phi$  is revealed by the tree pair  $(D, \phi(D))$ , where  $D$  is the given *domain*.

The *domain* may be implicitly specified by the string 'minimal' or 'wrt QNB'. In the first case, *domain* is taken to be the minimal *domain* required to specify the automorphism. In the second case, *domain* is taken to be the minimal expansion of the quasinormal basis.

**Returns** *None* if the pair is revealing for  $\phi$ . Otherwise, returns (as a *Word*) the root of a component of either  $D \setminus \phi(D)$  or  $\phi(D) \setminus D$  which does not contain an attractor/repeller.

```
>>> load_example('olga_f').test_revealing() is None
True
>>> print(load_example('semi_inf_c').test_revealing())
x1 a2 a1
>>> print(load_example('non_revealing').test_revealing())
x1 a1 a1
>>> f = load_example('cyclic_order_six')
>>> print(f.test_revealing('minimal'))
x1 a2
>>> f.test_revealing('wrt QNB') is None
True
```

**Caution:** This is an experimental feature based on [SD10].

#### Todo

This method is badly named; something like `test_revealed_by(basis)` would be better.

**is\_revealing** (*domain*='wrt QNB')

Calls `test_revealing()`, but only returns True or False.

**Returns** True if the pair is revealing, otherwise False.

```
>>> load_example('olga_f').is_revealing()
True
>>> load_example('semi_inf_c').is_revealing()
False
>>> load_example('non_revealing').is_revealing()
False
>>> f = load_example('cyclic_order_six')
>>> f.is_revealing('minimal')
False
>>> f.is_revealing('wrt QNB')
True
```

**Caution:** This is an experimental feature based on [SD10].

#### Todo

This method is badly named; something like `is_revealed_by(basis)` would be better.

**area\_to\_identity** (*scaled=False*)

Let  $\phi \in F_{n,1}$ , viewed as a bijection of the interval. What is the (unsigned) area between  $\phi$  and the identity?

**Parameters** **scaled** (*bool*) – If true, the area is normalised by ignoring subintervals where the function is equal to the identity.

**Warning:** This is an experimental feature based on a suggestion by Henry Bradford.

```
>>> for n in range(4):
...     print(standard_generator(n).area_to_identity())
...
5/32
5/128
5/512
5/2048
>>> x0 = standard_generator(0)
>>> x1 = standard_generator(1)
>>> g = ~x0 * x1 * x0
>>> print(g.area_to_identity())
3/32
>>> f = load_example('non_dyadic_fixed_point')
>>> print(f.area_to_identity())
43/768

>>> for n in range(4):
...     print(standard_generator(n).area_to_identity(scaled=True))
5/32
5/32
5/32
5/32
```

**centralise\_period** (*period, trees, rearranger*)

Constructs a new element  $\psi$  commuting with the given element  $\phi$ . We construct the centralising element by altering the  $\phi$ -orbit structure below the orbits of the given *period*.

There are two parameters: a collection of labelled *trees* and a *rearranger* element; in the notation of [BGG11] these are elements of  $K_{m_i}$  and  $G_{n,r_i}$  respectively. .. todo:: doctests .. caution:: This is an experimental feature based on [BGG11].

### 3.6.2 Next steps

Because an Automorphism can be *repeatedly applied*, we may consider the orbit  $\{w\psi^n \mid n \in \mathbb{Z}\}$  of any word  $w$ . (Note that this is the orbit of  $w$  under the cyclic subgroup  $\langle \psi \rangle$ , rather than all of  $G_{n,r}$ .)

The next module gives us tools to classify and work with these orbits. (In truth, the *Automorphism* class uses these tools, so this documentation is really in the wrong order.)

## 3.7 Orbits

Let  $y$  be a word, and let  $X$  be an expansion of the standard basis  $x = \{x_1, \dots, x_n\}$ ; finally let  $\phi$  be an *MixedAut*. We call the set  $\text{orb}_\phi(y) = \{y\phi^i\}_{i \in \mathbb{Z}}$  the  $\phi$ -orbit of  $y$ . Let us refer to the intersection  $\text{orb}_\phi(y) \cap X \langle A \rangle$  as the *X-component* of (the  $\phi$ -orbit of)  $y$ . Higman demonstrated how we could classify these components in [Hig74] (section 9).

This module is responsible for two orbit-related data structures. Firstly, `ComponentType` is set up to describe all possible types of orbits according to Higman’s scheme. Secondly, the `SolutionSet` describes solutions to the equation  $u\phi^i = v$ .

### 3.7.1 Types of orbits and components

These components come in five different types:

1. **Complete infinite components.** The component is the entire orbit  $\{y\phi^i\}_{i \in \mathbb{Z}}$  and each element of the orbit is different.
2. **Complete finite components.** The component is the entire orbit, which eventually it repeats itself. Thus the component only contains a finite number of distinct words.
3. **Right semi-infinite components.** The forward orbit  $\{y\phi^n\}_{n \in \mathbb{N}}$  belongs to the component and does not repeat itself. However, the backward orbit  $\{y\phi^{-n}\}_{n \in \mathbb{N}}$  eventually leaves  $X\langle A \rangle$ ; thus the component only contains a finite amount of the backward orbit.
4. **Left semi-infinite components.** The backward orbit is contained in the component, and does not repeat itself; the forward orbit eventually leaves  $X\langle A \rangle$ .
5. **Incomplete finite components.** A finite part of the orbit

$$y\phi^{-n}, \dots, y\phi^{-1}, y, y\phi, \dots, y\phi^m$$

for which both  $y\phi^{-n-1}$  and  $y\phi^{m+1}$  are not in  $X\langle A \rangle$ .

There are six different ways we can form orbits using these components:

- **Complete infinite (or *doubly infinite*) orbits.** The entire orbit is a complete infinite component (type 1).
- **Complete finite (or *periodic*) orbits.** The entire orbit is a component finite component (type 2).
- **Incomplete finite (or *bad*) orbits.** The orbit does not contain any complete or semi-infinite components. Thus the only components which may appear are of type 5. Note that these orbits need not have any components at all!
- **Incomplete infinite orbits.** The orbit contains at least one semi-infinite component (types 3, 4). The orbit may also contain incomplete finite components (type 5). We give names to the different cases:
  - **Right semi-infinite orbits.** One component of type 3, none of type 4.
  - **Left semi-infinite orbits.** One component of type 4, none of type 3.
  - **Pond orbits.** One component of type 3 and one of type 4. These components must be separated by a finite list of words in  $V_{n,r} \setminus X\langle A \rangle$ ; we think of this list as being a *pond*.

If that wasn’t confusing enough, we have a another way to classify those orbits which are not incomplete finite.

1. Type A components are those with characteristic  $(m, \epsilon)$ , where  $\epsilon$  is the empty word. These are precisely the periodic orbits.
2. Type B components are those with characteristic  $(m, \Gamma)$ , where  $\Gamma \neq \epsilon$  is nontrivial.
3. Type C components are those which are not of type A or B—that is, they are the components which do not have a characteristic. If  $x$  belongs to a type C orbit, then  $x\phi^\ell = z\Delta$  for some  $z$  of type B. That is, type C orbits are those ‘below’ type B orbits.

---

**Note:** Type B components must be semi-infinite, **but not all semi-infinite components are of type B**. Complete infinite components are always of type C, but **some semi-infinite components are of type C too**.

---

**class** thompson.orbits.ComponentType

This class is essentially a glorified `enumeration`: its instances store a number from 1 to 5 to represent the type of a component.

**Variables characteristic** – Either the characteristic  $(m, \Gamma)$  of this component, or `None` if this component has no characteristic. Periodic components have characteristic  $(m, \varepsilon)$ , where  $\varepsilon$  is the empty word.

**See also:**

Section 4.1 of the paper.

**classmethod** periodic(*period*)

Describes a complete finite (i.e. periodic, type 2) component. The argument is the period of the component.

**classmethod** semi\_infinite(*characteristic, backward=False*)

Describes a semi-infinite component (types 3, 4). The first argument is the characteristic of the component; use `None` if this component doesn't have a characteristic. The second argument should specify where this component is left or right semi-infinite.

**classmethod** complete\_infinite()

Describes a component which is infinite in both directions (type 1).

**classmethod** incomplete()

Describes a component which is incomplete finite (type 5).

**is\_type\_A**()

Returns true if this component belongs to an orbit of type A (periodic).

**is\_type\_B**()

Returns true if this component belongs to an orbit of type B (has a characteristic)

**is\_type\_C**()

Returns true if this component belongs to an orbit of type C (does not have a characteristic)

**is\_incomplete**()

Returns True if this component is incomplete, otherwise False.

**is\_semi\_infinite**()

Returns True is this component is semi\_infinite (with or without characteristic); otherwise returns False.

### 3.7.2 The SolutionSet structure

Solutions to the equation  $u\phi^m = v$  come in specific instances. If there are no solutions, the solution set is  $\emptyset$ . Otherwise  $u$  and  $v$  share an orbit. If this orbit is periodic, then the solution set is of the form  $m + p\mathbb{Z}$ , where  $p$  is the period of the orbit. Otherwise the solution set is a single integer  $m$ .

Internally we represent this as a pair (*base, increment*) of integers. The first element *base* is a solution  $m$  if it exists; otherwise `None`. The second element *increment* is the increment  $p$  between solutions (which occurs for a periodic orbit only).

**class** thompson.orbits.SolutionSet

Solution sets to the orbit sharing test  $u\phi^k = v$ . These are either

- empty (no such  $k$ ) exists;
- a singleton; or
- a linear sequence  $a + b*n$ .

Create solution sets as follows:

```

>>> print (SolutionSet(5, 2))
{..., 3, 5, 7, 9, 11, 13, ...}
>>> print (SolutionSet.singleton(4))
{4}
>>> print (SolutionSet.empty_set())
{}
>>> print (SolutionSet.the_integers())
{..., -1, 0, 1, 2, 3, 4, ...}

```

**is\_sequence()**

Returns True if this set contains more than one distinct element; otherwise returns False.

```

>>> SolutionSet.empty_set().is_sequence()
False
>>> SolutionSet.singleton(2).is_sequence()
False
>>> SolutionSet(base=4, increment=3).is_sequence()
True
>>> SolutionSet.the_integers().is_sequence()
True

```

**is\_singleton()**

Returns True if this contains precisely one element, otherwise False.

```

>>> SolutionSet.empty_set().is_singleton()
False
>>> SolutionSet.singleton(2).is_singleton()
True
>>> SolutionSet(base=4, increment=3).is_singleton()
False
>>> SolutionSet.the_integers().is_singleton()
False

```

**is\_empty()**

Returns True if this set is empty, otherwise False.

```

>>> SolutionSet.empty_set().is_empty()
True
>>> SolutionSet.singleton(2).is_empty()
False
>>> SolutionSet(base=4, increment=3).is_empty()
False
>>> SolutionSet.the_integers().is_empty()
False

```

**is\_the\_integers()**

Returns True if this set contains every integer; otherwise returns False.

```

>>> SolutionSet.empty_set().is_the_integers()
False
>>> SolutionSet.singleton(2).is_the_integers()
False
>>> SolutionSet(base=4, increment=3).is_the_integers()
False
>>> SolutionSet.the_integers().is_the_integers()
True

```

**\_\_contains\_\_ (other)**

Returns true if this set contains an *other* number.

```

>>> 1024 in SolutionSet.empty_set()
False
>>> 1024 in SolutionSet.singleton(128)
False
>>> 1024 in SolutionSet(0, 256)
True
>>> 1024 in SolutionSet.the_integers()
True

```

### and (*other*)

The & operator (usually used for bitwise and) stands for intersection of sets.

```

>>> phi = SolutionSet.empty_set()
>>> Z = SolutionSet.the_integers()
>>> singleton = SolutionSet.singleton
>>> print(phi & phi)
{}
>>> print(phi & singleton(1))
{}
>>> print(phi & SolutionSet(2, 3))
{}
>>> print(singleton(1) & singleton(1))
{1}
>>> print(singleton(1) & singleton(2))
{}
>>> print(singleton(8) & SolutionSet(4, 2))
{8}
>>> print(SolutionSet(1, 3) & SolutionSet(2, 3))
{}
>>> print(SolutionSet(1, 3) & SolutionSet(1, 2))
{..., -5, 1, 7, 13, 19, 25, ...}
>>> print(SolutionSet(1, 18) & SolutionSet(5, 24))
{}
>>> print(SolutionSet(1, 18) & SolutionSet(13, 24))
{..., -35, 37, 109, 181, 253, 325, ...}
>>> print(SolutionSet(1, 3) & Z)
{..., -2, 1, 4, 7, 10, 13, ...}
>>> print(Z & Z)
{..., -1, 0, 1, 2, 3, 4, ...}

```

## 3.7.3 Helper functions

`thompson.orbits.print_component_types` (*aut*, *basis=None*, *words=None*)

Prints the classification of the components under *aut* of each word in *words* with respect to *basis*. If *basis* is omitted, it is taken to be the smallest possible expansion which could potentially be a semi-normal basis; see `seminormal_form_start_point()`. If *words* is omitted, it is taken to be the same as *basis*.

```

>>> print_component_types(load_example('arity_three_order_inf'))
x1 a1: Left semi-infinite component with characteristic (-1, a1)
x1 a2: Bi-infinite component
x1 a3 a1: Bi-infinite component
x1 a3 a2: Bi-infinite component
x1 a3 a3: Right semi-infinite component with characteristic (1, a1)
with respect to the basis [x1 a1, x1 a2, x1 a3 a1, x1 a3 a2, x1 a3 a3]

```

```

>>> print_component_types(load_example('arity_four'))
x1 a1 a1: Periodic component of order 4

```

```
x1 a1 a2: Periodic component of order 4
x1 a1 a3: Periodic component of order 4
x1 a1 a4: Periodic component of order 4
x1 a2: Bi-infinite component
x1 a3: Right semi-infinite component with characteristic (1, a3)
x1 a4: Left semi-infinite component with characteristic (-1, a4)
with respect to the basis [x1 a1 a1, x1 a1 a2, x1 a1 a3, x1 a1 a4, x1 a2, x1 a3, x1 a4]
```

**See also:**

The `orbit_type()` method.

**class** `thompson.orbits.Characteristic` (*power, multiplier*)

`__getnewargs__()`

Return self as a plain tuple. Used by copy and pickle.

`__getstate__()`

Exclude the OrderedDict from pickling

**static** `__new__` (*\_cls, power, multiplier*)

Create new instance of Characteristic(power, multiplier)

`__repr__()`

Return a nicely formatted representation string

**multiplier**

Alias for field number 1

**power**

Alias for field number 0

### 3.7.4 Next steps

With tools to describe orbits in hand, we can dive straight into the `Automorphism` class. In particular, we need to *know how orbits work to*

- determine *quasinormal bases*
- perform the *orbit sharing test*
- perform the *conjugacy test*

## 3.8 MixedAuts

A general automorphism  $\phi \in G_{n,r}$  may exhibit periodic behaviour, infinite behaviour, or a mix of both. We say that an automorphism is

- *purely periodic* if it has finite order;
- *purely infinite* if it has no periodic *orbits*; and
- *mixed* otherwise.

We can represent a mixed automorphism  $\psi$  as a **free product**  $\psi = \psi_P * \psi_I$  of a purely periodic and purely infinite automorphism. The automorphisms  $\psi_P$  and  $\psi_I$  are called `free_factors()`. We form this decomposition because the conjugacy problem is easier to solve when the automorphisms are both pure infinite or both pure periodic.

This class is responsible for:

- Generating the free factors from a mixed automorphism;
- Combining a periodic and infinite factor into a mixed automorphism; and
- Delegating the conjugacy and power conjugacy tests to these factors.

### 3.8.1 The MixedAut class

`class thompson.mixed.MixedAut` (*domain, range, reduce=True*)

#### `free_factors()`

In principle, an automorphism can be decomposed into free factors using any partition of the *quasi-normal basis*  $X = X_1 \sqcup X_2$ . The decomposition  $X = X_P \sqcup X_I$  is particularly interesting since these sets generate  $\psi$ -invariant subalgebras, which means that a conjugator can be reassembled from conjugators of the factors.

#### See also:

This jargon comes from Theorem 5.1.

This is a convenience method which produces the free factors  $\psi_P$  and  $\psi_I$  from  $\psi$ .

#### `free_factor(generators)`

This method restricts the current automorphism to the subalgebra generated by the given set  $W$  of *generators*. This is then transformed into an automorphism of an isomorphic algebra with minimal alphabet size  $1 \leq s \leq n - 1$ .

$$\begin{aligned}
 &G_{n,r} \\
 &\rightarrow G_{n,|W|} \\
 &\rightarrow G_{n,s} \\
 &\phi \\
 &\mapsto \phi|_{W\langle A \rangle \langle \lambda \rangle} \\
 &\mapsto \phi'
 \end{aligned}$$

**Returns** The transformed automorphism  $\phi \in G_{n,s}$ . Its type is *PeriodicAut* or *InfiniteAut* as appropriate.

**Raises** `ValueError` – if an empty list of *generators* is provided.

```

>>> phi = load_example('alphabet_size_two')
>>> qnb = phi.quasinormal_basis
>>> p, i = phi._partition_basis(qnb)
>>> print(phi.free_factor(p))
PeriodicAut: V(3, 1) -> V(3, 1) specified by 1 generators (after expansion and reduction).
This automorphism was derived from a parent automorphism.
'x' and 'y' represent root words of the parent and current derived algebra, respectively.
x1 a1 ~> y1 => y1 ~> x1 a1
>>> print(phi.free_factor(i))
InfiniteAut: V(3, 1) -> V(3, 1) specified by 5 generators (after expansion and reduction).
This automorphism was derived from a parent automorphism.
'x' and 'y' represent root words of the parent and current derived algebra, respectively.
x1 a2 ~> y1 a1 => y1 a1 a3 ~> x1 a2 a3
x1 a3 ~> y1 a2 => y1 a3 ~> x2
x2 a1 ~> y1 a3 a1 => y1 a2 ~> x1 a3
x2 a2 ~> y1 a3 a2 => y1 a1 a2 ~> x1 a2 a2
x2 a3 ~> y1 a3 a3 => y1 a1 a1 ~> x1 a2 a1

```

**test\_conjugate\_to** (*other*)

Determines if the current automorphism  $\psi$  is conjugate to the *other* automorphism  $\phi$ .

**Returns** if it exists, a conjugating element  $\rho$  such that  $\rho^{-1}\psi\rho = \phi$ . If no such  $\rho$  exists, returns `None`.

**Raises** **ValueError** – if the automorphisms have different arities or alphabet sizes.

```
>>> psi, phi = random_conjugate_pair()
>>> rho = psi.test_conjugate_to(phi)
>>> rho is not None
True
>>> psi * rho == rho * phi
True
>>> psi, phi = load_example_pair('first_pond_example')
>>> rho = psi.test_conjugate_to(phi)
>>> rho is not None
True
>>> psi * rho == rho * phi
True
```

**See also:**

This is an implementation of Algorithm 5.6 in the paper. It depends on Algorithms 5.13 and 5.27; these are the *periodic* and *infinite* tests for conjugacy.

**find\_power\_conjugators** (*other*, *cheat=False*)

Determines if some power of the current automorphism  $\psi$  is conjugate to some power of the *other* automorphism  $\phi$ . This method exhaustively searches for all minimal solutions  $(a, b, \rho)$  such that  $\rho^{-1}\psi^a\rho = \phi^b$ . This method returns a generator which yields such minimal solutions.

**Warning:** If the *power\_conjugacy\_bounds()* are reasonable large (say  $> 30$ ), this method could potentially take a long time!

**Parameters** **cheat** – (internal) for speeding up testing.

**Raises** **ValueError** – if the automorphisms have different arities or alphabet sizes.

**See also:**

This is an implementation of Algorithm 6.13 in the paper. It depends on Algorithms 5.6 (the *conjugacy test*) and 6.11 (the *infinite power conjugate test*.)

**test\_power\_conjugate\_to** (*other*, *cheat=False*)

Determines if some power of the current automorphism  $\psi$  is conjugate to some power of the *other* automorphism  $\phi$ . This method searches for a **single** minimal solution  $(a, b, \rho)$  such that  $\rho^{-1}\psi^a\rho = \phi^b$ . If no such solution exists, returns `None`.

**Warning:** If the *power\_conjugacy\_bounds()* are reasonable large (say  $> 30$ ), this method could potentially take a long time!

**Parameters** **cheat** – (internal) for speeding up testing.

**Raises** **ValueError** – if the automorphisms have different arities or alphabet sizes.

```
>>> psi, phi = load_example_pair('mixed_pconj')
>>> a, b, rho = psi.test_power_conjugate_to(phi)
>>> a, b
(6, 3)
>>> psi**a * rho == rho * phi ** b
True
```

**See also:**

This is an implementation of Algorithm 6.13 in the paper. It depends on Algorithms 5.6 (the *conjugacy test*) and 6.11 (the *infinite power conjugate test*.)

### 3.8.2 Next steps

To complete the details of the *conjugacy test*, we have to be able to test if two pure periodic automorphisms are conjugate and if two pure infinite automorphisms are conjugate. Any given automorphism can be broken down into a pure periodic and pure infinite part. These parts are called *free factors*.

## 3.9 Free Factors and conjugacy

To *test for conjugacy* we need to extract periodic and infinite components of an automorphism and test those components for conjugacy. These next few classes keep track of

- how the components embed into the original automorphism, and
- any extra information that we need to test the components for conjugacy.

### 3.9.1 Periodic Factors

**class** thompson.periodic.**PeriodicAut** (*domain, range, reduce=True*)

A purely periodic automorphism, which may have been extracted from a mixed automorphism.

```
>>> example_5_9 = load_example('example_5_9')
>>> print(example_5_9)
PeriodicAut: V(2, 1) -> V(2, 1) specified by 7 generators (after expansion and reduction).
x1 a1 a1 a1 -> x1 a1 a1 a2
x1 a1 a1 a2 -> x1 a1 a2
x1 a1 a2 -> x1 a1 a1 a1
x1 a2 a1 a1 -> x1 a2 a1 a2
x1 a2 a1 a2 -> x1 a2 a1 a1
x1 a2 a2 a1 -> x1 a2 a2 a2
x1 a2 a2 a2 -> x1 a2 a2 a1
>>> sorted(example_5_9.cycle_type)
[2, 3]
>>> #Two orbits of size 2, one orbit of size 3
>>> from pprint import pprint
>>> pprint(example_5_9.multiplicity)
{2: 2, 3: 1}
>>> example_5_9.order
6
```

```
>>> load_example('cyclic_order_six').order
6
>>> phi = random_periodic_automorphism()
>>> 1 <= phi.order < float('inf')
True
>>> len(phi.cycle_type) != 0
True
>>> len(phi.characteristics)
0
```

**test\_conjugate\_to** (*other*)

We can determine if two purely periodic automorphisms are conjugate by examining their orbits.

```
>>> psi_p, phi_p = load_example('example_5_12_psi'), load_example('example_5_12_phi')
>>> rho_p = psi_p.test_conjugate_to(phi_p)
>>> print(rho_p)
PeriodicAut: V(2, 1) -> V(2, 1) specified by 6 generators (after expansion and reduction).
x1 a1 a1 a1 a1 -> x1 a1 a2
x1 a1 a1 a1 a2 -> x1 a2 a2
x1 a1 a1 a2 -> x1 a1 a1 a1
x1 a1 a2 -> x1 a2 a1 a1
x1 a2 a1 -> x1 a1 a1 a2
x1 a2 a2 -> x1 a2 a1 a2
>>> rho_p * phi_p == psi_p * rho_p
True
>>> psi_p, phi_p = random_conjugate_periodic_pair()
>>> rho_p = psi_p.test_conjugate_to(phi_p)
>>> rho_p * phi_p == psi_p * rho_p
True
```

**See also:**

This implements algorithm 5.13 of the paper—see section 5.3.

**find\_power\_conjugators** (*other*, *identity\_permitted=False*, *cheat=False*)**test\_power\_conjugate\_to** (*other*, *cheat=False*)

Tests two periodic factors to see if they are power conjugate. Yields minimal solutions  $(a, b, \rho)$  such that  $\rho^{-1}\psi^a\rho = \phi^b$ .

**See also:**

Section 6.1 of the paper.

**power\_conjugacy\_bounds** (*other*, *identity\_permitted*)

We simply try all powers of both automorphisms. There are only finitely many, because everything is periodic.

**Returns** `self.power`, `other.power` if the identity is permitted as a solution; otherwise `self.power - 1`, `other.power - 1`.

**Todo**

Maybe this should be 0 to order if the identity is permitted and 1 to order otherwise?

**See also:**

Section 6.1 of the paper.

## 3.9.2 Infinite Factors

**class** thompson.infinite.**InfiniteAut** (*domain*, *range*, *reduce=True*)

A purely infinite automorphism which may have been extracted from a mixed automorphism.

```
>>> print(load_example('example_5_3').free_factors()[1])
InfiniteAut: V(2, 1) -> V(2, 1) specified by 6 generators (after expansion and reduction).
This automorphism was derived from a parent automorphism.
'x' and 'y' represent root words of the parent and current derived algebra, respectively.
x1 a1 a1 a1 a1 ~> y1 a1 a1 a1 => y1 a1 a1 ~> x1 a1 a1 a1
```

```

x1 a1 a1 a1 a2 ~> y1 a1 a1 a2 => y1 a1 a2 a1 ~> x1 a1 a1 a2 a1
x1 a1 a1 a2 ~> y1 a1 a2 => y1 a1 a2 a2 ~> x1 a1 a1 a2 a2
x1 a2 a1 ~> y1 a2 a1 => y1 a2 a1 a1 ~> x1 a2 a1 a1
x1 a2 a2 a1 ~> y1 a2 a2 a1 => y1 a2 a1 a2 ~> x1 a2 a1 a2
x1 a2 a2 a2 ~> y1 a2 a2 a2 => y1 a2 a2 ~> x1 a2 a2
>>> phi = random_infinite_automorphism()
>>> phi.order
inf
>>> len(phi.cycle_type)
0
>>> len(phi.characteristics) != 0
True

```

**test\_conjugate\_to** (*other*)

We can determine if two purely infinite automorphisms are conjugate by breaking down the quasi-normal basis into *equivalence classes*.

```

>>> for psi_i, phi_i in (
...     load_example_pair('example_5_26'),
...     load_example_pair('inf_conj'),
...     random_conjugate_infinite_pair()):
...     rho_i = psi_i.test_conjugate_to(phi_i)
...     print('Conjugate:', rho_i is not None, ' Conjugator works:', rho_i * phi_i == psi_i)
Conjugate: True Conjugator works: True
Conjugate: True Conjugator works: True
Conjugate: True Conjugator works: True

```

```

>>> f, g = (load_example('not_conjugate_' + c) for c in 'fg')
>>> f.is_conjugate_to(g)
False

```

**See also:**

This implements Algorithm 5.27 of the paper—see Section 5.4.

**equivalence\_data** (*type\_b*, *type\_c*)

Let  $X$  be the quasi-normal basis for the current automorphism  $\psi$ . We can define an equivalence relation  $\equiv$  on  $X$  by taking the non-transitive relation

$$x \equiv y \iff \exists k, \Gamma, \Delta : x\Gamma\psi^k = y\Delta$$

and allowing it to generate an equivalence relation. This method returns a pair (*roots*, *graph*) where

- *graph* is a `DiGraph`
  - vertices are type B words in  $X$
  - directed edges  $x \rightarrow y$  correspond to the direct relation  $x \equiv y$
  - the *graph* is a directed forest
- *roots* is a list of type B words in  $X$ 
  - Each *root* is the root of a different tree in the forest *graph*.

This information allows us to (attempt to) compute the images of  $X$  under a conjugator given only the images of *roots*.

**See also:**

Definition 5.17 through to Lemma 5.20.

**potential\_image\_endpoints** (*other, self\_type\_b*)

Let  $x$  be a type B word with respect to the current automorphism. This returns a mapping which takes  $x$  and produces the set of words  $w$  which are endpoints of *other*-orbits which have the same characteristic as  $x$ .

**See also:**

The sets  $\mathcal{R}_i$  of definition 5.23.

**find\_power\_conjugators** (*other, cheat=False*)

Tests two infinite factors to see if they are power conjugate. Yields minimal solutions  $(a, b, ho)$ .

```
>>> example_6_8_psi, example_6_8_phi = load_example_pair('example_6_8')
>>> solns = example_6_8_psi.find_power_conjugators(example_6_8_phi)
>>> for a, b, rho in solns:
...     print(a, b)
...     assert example_6_8_psi ** a * rho == rho * example_6_8_phi ** b
3 1
-3 -1
```

**Warning:** If the `power_conjugacy_bounds()` are reasonable large (say  $> 30$ ), this method could potentially take a long time!

**See also:**

Algorithm 6.11 of the paper.

**test\_power\_conjugate\_to** (*other, cheat=False*)

Tests two infinite factors to see if they are power conjugate. If a solution exists, returns  $(a, b, \rho)$  such that  $\rho^{-1}\psi^a\rho = \phi^b$ . Otherwise returns None.

```
>>> result = example_6_8_psi.test_power_conjugate_to(example_6_8_phi)
>>> result is not None
True
>>> a, b, rho = result
>>> (example_6_8_psi ** a) * rho == rho * (example_6_8_phi ** b)
True
```

**Warning:** If the `power_conjugacy_bounds()` are reasonable large (say  $> 30$ ), this method could potentially take a long time!

**See also:**

Algorithm 6.11 of the paper.

**power\_conjugacy\_bounds** (*other*)

Compute the bounds  $\hat{a}, \hat{b}$  on powers which could solve the power conjugacy problem.

```
>>> example_6_8_psi.power_conjugacy_bounds(example_6_8_phi)
(9, 1)
```

**See also:**

Proposition 6.6 and Corollary 6.7.

**minimal\_partition** ()

Let  $\psi$  be the current automorphism. This method partitions the characteristics  $M_\psi$  into cells  $P_1 \sqcup \dots \sqcup P_L$ , where - The multipliers  $\Gamma$  all have the same `root()`, for all  $(m, \Gamma)$  in each  $P_i$ . -  $L$  is minimal with this property.

```

>>> def print_partition(p):
...     for root in sorted(p.keys(), reverse=True):
...         print(format(root), end = ':')
...         for power, mult, _ in p[root]:
...             print(' ({} , {})'.format(power, format(mult)), end='')
...         print()
>>> print_partition(example_6_8_psi.minimal_partition())
a1: (-1, a1)
a2: (1, a2)
>>> print_partition(example_6_8_phi.minimal_partition())
a1: (-1, a1 a1 a1)
a2: (1, a2 a2 a2)
>>> example_6_9_psi, example_6_9_phi = load_example_pair('example_6_9')
>>> print_partition(example_6_9_phi.minimal_partition())
a1: (-2, a1)
a2: (1, a2)

```

**Returns** a dictionary of sets. The keys of this dictionary are the roots  $\sqrt{\Gamma}$ ; the values are the cells  $P_i$ . An element of a cell looks like  $m, \Gamma, r$  where  $m, \Gamma$  is a characteristic and  $r$  is the root power corresponding to  $\sqrt{\Gamma}$ .

**See also:**

the discussion following Corollary 6.7.

**static compute\_bounds** (*s\_parts*, *o\_parts*)

Computes the bounds  $\hat{a}, \hat{b}$  (in terms of the partitions  $P, Q$  given by *s\_parts* and *o\_parts*) as in eqns (14) and (15) of the paper.

```

>>> def bounds(s, o):
...     P = s.minimal_partition()
...     Q = o.minimal_partition()
...     a_hat = InfiniteAut.compute_bounds(P, Q)
...     b_hat = InfiniteAut.compute_bounds(Q, P)
...     return a_hat, b_hat
>>> bounds(example_6_8_psi, example_6_8_phi)
(9, 1)
>>> bounds(example_6_9_psi, example_6_9_phi)
(1, 2)

```

### 3.9.3 Next steps

With these classes, the conjugacy and power conjugacy tests are implemented. The other important part of this package is the *examples* module.

## 3.10 Examples

This module provides a number of explicit examples for use in doctests. Additionally, functions to generate random automorphisms are provided.

### 3.10.1 Explicit named examples

A list of named examples is loaded from the `.aut` files in the `examples` folder. This includes all the examples given in the paper, as well as others used to test the package. Use the following functions to load one of these examples.

`thompson.examples.available_examples()`

Returns an iterator yielding the names of the examples that are provided with the package. (Note that *the full list is provided* in this documentation.)

```
>>> list(available_examples())[:4]
['alphabet_size_two', 'arity_four', 'arity_three_order_inf', 'bleak_alpha']
```

`thompson.examples.load_example(name)`

Loads the example with the given *name* from disk. A corresponding *Automorphism* instance is created and returned. The results are cached, so call this method as often as you like.

`thompson.examples.load_example_pair(name)`

Loads a pair of examples, *\*name\*\_psi* and *\*name\*\_phi*.

**Return type** a 2-tuple of automorphisms.

`thompson.examples.load_all_examples()`

Loads (and processes) **all** examples provided by the package. Returns a dictionary whose keys are the example names and whose values are the loaded automorphisms.

---

**Note:** To discourage use of this function, it is not available when importing `*` from `thompson.examples`. Instead it must be explicitly imported with `from thompson.examples import load_all_examples`.

---

**Warning:** Some of the examples are slow to process, so calling this could take a long time.

`thompson.examples.standard_generator(n=0)`

Produces the standard generator  $X_n$  of  $F$  as described in [CFP96]. For instance,  $X_0$  is the following:

```
>>> print(standard_generator())
InfiniteAut: V(2, 1) -> V(2, 1) specified by 3 generators (after expansion and reduction).
x1 a1 -> x1 a1 a1
x1 a2 a1 -> x1 a1 a2
x1 a2 a2 -> x1 a2
```

For  $n > 0$  the element  $X_n$  is a *Mixed automorphism*, consisting of a large fixed part and a smaller part which looks like  $X_0$ .

```
>>> from random import randint
>>> n = randint(1, 20)
>>> x_n = standard_generator(n)
>>> type(x_n)
<class 'thompson.mixed.MixedAut'>
```

The  $X_n$  generate  $F$ ; in fact just  $X_0$  and  $X_1$  are sufficient, due to the relation  $X_k^{-1}X_nX_k = X_{n+1}$  for  $k < n$ . See [CFP96] for more details.

```
>>> x_k = standard_generator(randint(0, n-1))
>>> x_k * x_n * ~x_k == standard_generator(n+1) #operation is the other way round in Python
True
```

---

**Todo**

Add the named generators A, B, C of Thompson's T and V. Analogues for general  $G_{n,r}$ ?

---

## List of named examples

Note that some automorphisms have more than one name—they will appear once in this list for every alias.

**alphabet\_size\_two** A toy example to test that the program works in  $V_{3,2}$ .

**arity\_four** Another example for sanity checking. This is an automorphism of  $V_{4,1}$ .

**arity\_three\_order\_inf** A purely *infinite* automorphism of  $V_{3,1}$ .

**bleak\_alpha** The element  $\alpha$  of  $V$  used by Bleak *et al* [*Bleak*] to illustrate their train tracks and flow graphs.

**cyclic\_order\_six** An introductory example drawn on the board by NB. It is purely periodic of order six.

**example\_4\_1** This example is used in the paper to illustrate the meaning of a semi-normal form (Example 4.11) and of a tree pair diagram (Example 4.1).

**example\_4\_11** This example is used in the paper to illustrate the meaning of a semi-normal form (Example 4.11) and of a tree pair diagram (Example 4.1).

**example\_4\_12** This example is used in the paper to illustrate the meaning of a semi-normal form. This example needs to be expanded from its minimal representation to find a semi-normal (in fact quasi-normal) form.

**example\_4\_17** Nathan's example of an automorphism containing a pond. This was simpler than the first example found.

**example\_4\_33** Nathan's example of an automorphism containing a pond. This was simpler than the first example found.

**example\_4\_35** Nathan's example of an automorphism containing a pond. This was simpler than the first example found.

**example\_4\_5** This example is used to illustrate the different types of *components/orbits* in the paper.

**example\_4\_7** This example is used to illustrate the different types of *components/orbits* in the paper.

**example\_5\_12\_phi** The example used in the paper to demonstrate the *periodic conjugacy test*.

**example\_5\_12\_psi** The example used in the paper to demonstrate the *periodic conjugacy test*.

**example\_5\_15** This example is used in the paper to illustrate the process of finding the *quasi-normal basis*.

**example\_5\_18** This example is used in the paper to illustrate the process of finding the *quasi-normal basis*.

**example\_5\_2** This example is used to illustrate the different types of *components/orbits* in the paper.

**example\_5\_26\_phi** This example is used in the paper to illustrate the process of finding the *quasi-normal basis*.

**example\_5\_26\_psi** The example used in the paper to demonstrate the *infinite conjugacy test*.

**example\_5\_3** This example is used to demonstrate how we can break down an automorphism into its *free\_factors()*.

**example\_5\_9** This example is used in the paper to illustrate the definition of *cycle types*.

**example\_6\_2** Example 6.2 of the paper, used to illustrate lemma 6.1.

**example\_6\_8\_phi** A purely infinite automorphism which illustrates the bounds  $\hat{a}, \hat{b}$  for power conjugacy.

**example\_6\_8\_psi** This example is used in the paper to illustrate the meaning of a semi-normal form (Example 4.11) and of a tree pair diagram (Example 4.1).

**example\_6\_9\_phi** Example 6.2 of the paper, used to illustrate lemma 6.1.

**example\_6\_9\_psi** This example is used in the paper to illustrate the meaning of a semi-normal form (Example 4.11) and of a tree pair diagram (Example 4.1).

**first\_pond\_example\_phi** The first example of a pond orbit, found by a random search. Its banks are  $x\alpha_1^4\alpha_2$  and  $x\alpha_1\alpha_2^2$ .

**first\_pond\_example\_psi** The element which was conjugate to the first example of a pond orbit, found by a random search.

**four\_minimal\_revealing\_pairs** I think this automorphism has 4 minimal revealing pairs. It has an orbit of characteristic  $(-4, a1^4)$  spread apart over four components of  $A \setminus B$ . Brin's algorithm would have you add three of these components to the tree, and I think the component you DON'T add is the one that determines the revealing pair.

**inf\_conj\_phi** An example used to debug the *infinite conjugacy test*.

**inf\_conj\_psi** An example used to debug the *infinite conjugacy test*.

**mixed\_pconj\_phi** An example of mixed power conjugacy found by random search. Should have  $\psi^{-4} \sim \phi^2$ .

**mixed\_pconj\_psi** An example of mixed power conjugacy found by random search. Should have  $\psi^{-4} \sim \phi^2$ .

**multiple\_classes** A purely infinite automorphism for which there are two equivalence classes defined on X. This was the first example found (via random search) of such an automorphism. Photographic notes are contained in /thompson/examples.

**multiple\_classes\_smaller** A purely infinite automorphism for which there are two equivalence classes defined on X. It was more straightforward to see that this automorphism had two equivalence classes.

**nathan1\_example** One of Nathan's examples. This is *almost*, but not quite conjugate to nathan\_pond\_example. Nathan had this to say: "*I untwisted one of the infinite orbits after a conjugation and so they definitely should not be conjugate.*"

**nathan\_pond\_example** Nathan's example of an automorphism containing a pond. This was simpler than the first example found.

**no\_minimal\_revealing** An example from section 3.6 of [SD10]. This is an automorphism  $f \in G_{2,1}$  for which there is no minimal revealing pair from which every other revealing pair can be obtained via rollings.

**non\_dyadic\_fixed\_point** In their paper introducing strand diagrams, Belk and Matucci provide an example of an element of  $F$  which has a non-dyadic fixed point. This is a slight simplification of that example.

**non\_revealing** A purely infinite automorphism for which - The minimal representation does NOT correspond to a revealing pair; and - The quasnormal basis does NOT correspond to a revealing pair.

**not\_conjugate\_f** At first glance this automorphism looks like it could be conjugate to not\_conjugate\_g; indeed this satisfies the criteria of theorem 2 in [SD10]. However the *infinite conjugacy test* only yields one potential conjugator  $\rho$  which turns out not to be an automorphism (in fact, just an endomorphism) of  $V_{2,1}$ .

**not\_conjugate\_g** At first glance this automorphism looks like it could be conjugate to not\_conjugate\_f; indeed this satisfies the criteria of theorem 2 in [SD10]. However the *infinite conjugacy test* only yields one potential conjugator  $\rho$  which turns out not to be an automorphism (in fact, just an endomorphism) of  $V_{2,1}$ .

**olga\_f** An automorphism described in example 5 of [SD10]. Nathan had this to say: "*olga\_f is conjugate to olga\_g, via a rho that can be found using your program or olga\_h. Note that characteristics are not the same for two of the semi-infinite orbits.*"

**olga\_g** An automorphism described in example 5 of [SD10]. Nathan had this to say: "*olga\_f is conjugate to olga\_g, via a rho that can be found using your program or olga\_h. Note that characteristics are not the same for two of the semi-infinite orbits.*"

**olga\_gdash** An automorphism described in example 5 of [SD10]. The paper claims that olga\_f and olga\_gdash are not conjugate.

**olga\_h** An automorphism described in example 5 of [SD10]. Nathan had this to say: “*olga\_f is conjugate to olga\_g, via a rho that can be found using your program or olga\_h. Note that characteristics are not the same for two of the semi-infinite orbits.*”

**periodic\_QNB\_206** This purely periodic automorphism (again found by random search) has a quasinormal basis of size 206.

**periodic\_QNB\_344** This purely periodic automorphism (again found by random search) has a quasinormal basis of size 344.

**pond\_width\_4** An example found by random search of a pond orbit of width 4. Prior to this we had only seen ponds of width 1 or 2.

**power\_smaller\_QNB** The square of this automorphism appears to have a QNB above the QNB of the original automorphism.

**semi\_inf\_c** An example of an automorphism whose *quasi-normal basis*  $X$  contains an element  $x\alpha_2\alpha_1$  which belongs to a semi-infinite  $X$ -component which is not characteristic.

### 3.10.2 Randomly generated examples

#### Words

`thompson.examples.random_signature()`

Randomly generates a *Signature*  $(n, r)$  for use in the functions below. The values of  $n$  and  $r$  are chosen (uniformly) at random from  $n \in \{2, 3, 4\}$  and  $r \in \{1, 2, 3, 4, 5\}$ , respectively.

---

**Note:** This function is used to select a random signature when no *signature* argument is provided to the following random functions.

---

`thompson.examples.random_simple_word(signature=None)`

Randomly generates a *simple Word* belonging to the algebra with the given *signature*. The word consists of an  $x_i$  followed by 0–15 descendant operators  $\alpha_j$ . The length and the index of each  $\alpha_j$  is chosen (uniformly) at random.

```
>>> random_simple_word().is_simple()
True
```

`thompson.examples.random_basis(signature=None, num_expansions=None, *args, **kwargs)`

Randomly generates a basis for the algebra with the given *signature*  $(n, r)$ . The basis is generated by expanding the *standard\_basis()*  $num\_expansions$  times. The expansion point is chosen (uniformly) at random each time. If  $num\_expansions$  is not provided, a value from  $\{1, 2, 3, 4, 5\}$  is chosen (uniformly) at random.

```
>>> random_basis().is_basis()
True
```

---

**Note:** This does not generate *bases* uniformly at random. For instance, take  $V_{n,r} = V_{2,1}$  and let  $num\_expansions = 3$ . The first expansion always gives the basis  $[x\alpha_1, x\alpha_2]$ . Expanding this twice produces give six bases, one of which appears twice. (To see this, enumerate the rooted binary trees with four leaves.)

---

## Automorphisms

`thompson.examples.random_automorphism` (*signature=None*, *num\_expansions=None*, *\*args*, *\*\*kwargs*)

Randomly generates an automorphism for the algebra with the given signature. Two bases *domain* and *range* are generated by `random_basis()`. Then the *range* is manipulated according to the group we're interesting in. For `group == 'V'`, we randomly shuffle the *range*; for `group == 'T'`, we cyclicly permute the *range*; for `group == 'F'` we do nothing.

```
>>> random_automorphism(group='T').cycles_order()
True
>>> random_automorphism(group='F').preserves_order()
True
```

An automorphism is returned which maps the elements of *domain* to those of *range* in whichever order results.

---

**Note:** The bases may be reduced when the automorphism is created (if they contain redundancy).

---

`thompson.examples.random_periodic_automorphism` (*signature=None*, *num\_expansions=None*, *\*args*, *\*\*kwargs*)

Randomly generates an automorphism for the algebra with the given signature. A basis *domain* is generated by `random_basis()`. A copy *range* is made of *domain*, and is then `randomly shuffled`. An automorphism is returned which maps the elements of *domain* to those of *range* in order.

---

**Note:** The bases may be reduced when the automorphism is created (if they contain redundancy).

---

`thompson.examples.random_infinite_automorphism` (*signature=None*, *num\_expansions=None*, *\*args*, *\*\*kwargs*)

Randomly generates an infinite automorphism—either by chance, or by extracting an infinite *free\_factor()*.

`thompson.examples.random_conjugate_pair` (*signature=None*, *num\_expansions=None*, *\*args*, *\*\*kwargs*)

Calls `random_automorphism()` to create two automorphisms  $\psi$  and  $\rho$ . Returns the pair  $(\psi, \rho^{-1}\psi\rho)$ , which are conjugate by definition.

`thompson.examples.random_conjugate_periodic_pair` (*signature=None*, *num\_expansions=None*, *\*args*, *\*\*kwargs*)

The same as `random_conjugate_pair()`, except  $\psi$  is chosen to be a `random_periodic_automorphism()`.

`thompson.examples.random_conjugate_infinite_pair` (*signature=None*, *num\_expansions=None*, *\*args*, *\*\*kwargs*)

The same as `random_conjugate_pair()`, except  $\psi$  is chosen to be a `random_infinite_automorphism()`.

`thompson.examples.random_power_conjugate_pair` (*signature=None*, *num\_expansions=None*, *\*args*, *\*\*kwargs*)

Generates  $\psi, \rho$  using `random_automorphism()` and random powers  $a, b$ . Returns the tuple  $(\Psi, \Phi) = (\psi^b, \rho^{-1}\psi^a\rho)$ . Note that  $\Psi^a$  is conjugate to  $\Phi^b$  via  $\rho$ .

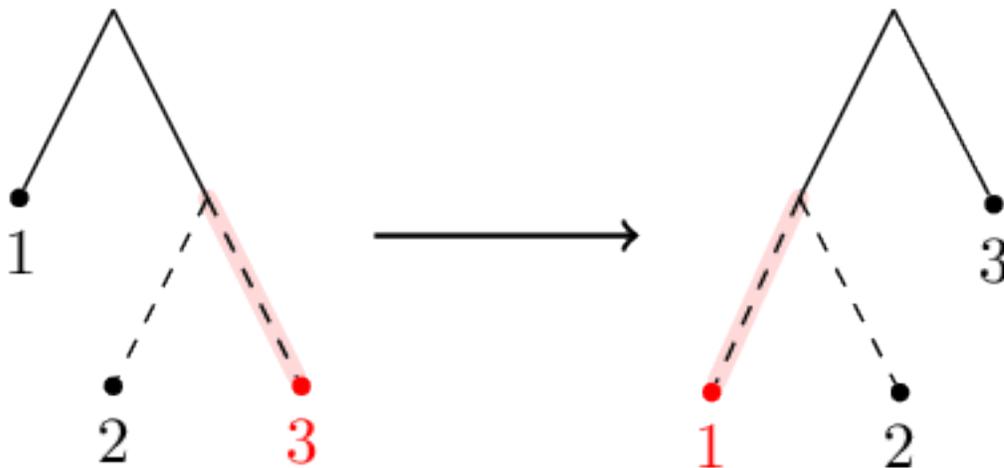
## 3.11 Drawing automorphisms

It can be helpful to see automorphisms rather than just read a description of them. To that end we introduce functions for rendering automorphisms as tree pair diagrams. The output looks best for automorphisms with a small arity (2–5) and a reasonable number of leaves.

### 3.11.1 Examples

First we take a well-behaved automorphism. The solid caret are those belonging to both the domain and range trees. All other caret are dotted. Some edges are highlighted in red—these correspond to the repellers and attractors discussed by [SD10].

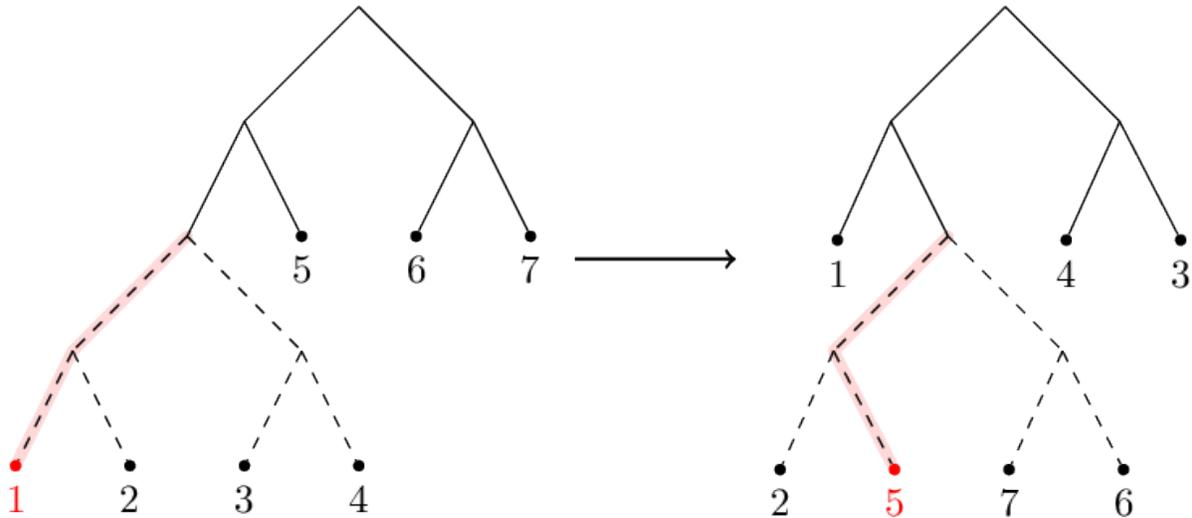
```
>>> from thompson import *
>>> x0 = standard_generator(0)
>>> plot(x0)
>>> forest(x0)
```



Discontinu-

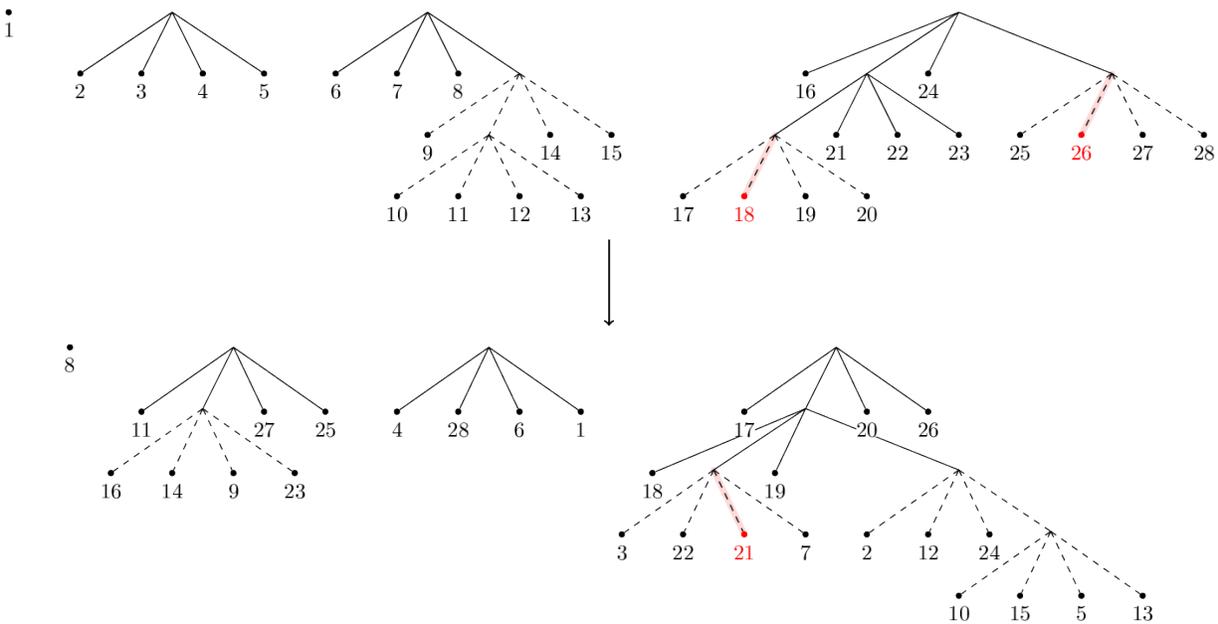
ities are fine too. Here's `example_4_17` for instance.

```
>>> pond = load_example('example_4_17')
>>> plot(pond, discontinuities=True)
>>> forest(pond)
```



Let's aim for something more chaotic. This example is complicated enough that the drawings don't really give us much insight into the automorphism.

```
>>> random = random_automorphism() #different every time!
>>> plot(random)
>>> forest(random, horiz=False)
```



### 3.11.2 Drawing functions

`thompson.drawing.display_file(filepath, format=None)`

Display the image at *filepath* to the user. This function behaves differently, depending on whether or not we execute it in a Jupyter notebook.

If we are **not** in a notebook, this function opens the given image using the operating system's default application for that file.

If we **are** in a notebook, this returns an IPython object corresponding to the given image. If this object is the last expression of a notebook code block, the image will be displayed. The image is handled differently depending on its *format*, which must be specified when the function is called in a notebook. Only the formats `'svg'` and `'pdf'` are accepted.

---

**Note:** PDF files are displayed in a notebook as a rendered PNG. The conversion is made using the `convert` program provided by `ImageMagick`, which must be available on the `PATH` for this function to work with PDF files.

---

---

### Todo

use a Python binding to ImageMagick rather than just shelling out?

---

`thompson.drawing.plot` (*aut*, *dest=None*, *display=True*, *endpoints=False*)

Plots the given *automorphism* as a function  $[0, 1] \rightarrow [0, 1]$ . The image is rendered as an SVG using `svgwrite`.

#### Parameters

- **dest** (*str*) – the destination filepath to save the SVG to. If *None*, the SVG is saved to a temporary file location.
- **display** (*bool*) – if True, automatically call `display_file()` to display the SVG to the user. Otherwise does nothing.
- **endpoints** (*bool*) – if True, open and closed endpoints are drawn on the plot to emphasise any discontinuities that *aut* has.

**Returns** the filepath where the SVG was saved. If *dest* is *None* this is a temporary file; otherwise the return value is simply *dest*.

`thompson.drawing.forest` (*aut*, *jobname=None*, *name=''*, *display=True*, *horiz=True*)

Draws the given *Automorphism* as a forest-pair diagram. We use the *minimal expansion* of the *quasi-normal basis* as the leaves of the domain forest.

The image is rendered as an PDF using the *tikz* graph drawing libraries and *lualatex*.

#### Parameters

- **jobname** (*str*) – the destination filepath to save the PDF to. A file extension should **not** be provided. If *None*, the SVG is saved to a temporary file location.
- **name** (*str*) – The label used for the arrow between domain and range forests.
- **display** (*bool*) – if True, automatically call `display_file()` to display the PDF to the user. Otherwise does nothing.
- **horiz** (*bool*) – if True, draws the range forest to the right of the domain forest. If false, draws the range forest below the range forest.

**Returns** the filepath where the PDF was saved. If *dest* is *None* this is a temporary file; otherwise the return value is simply *jobname* + `'pdf'`.

---

**Note:** The graph drawing is done via a TikZ and LaTeX. The source file is compiled using *lualatex*, which must be available on the `PATH` for this function to work.

---

## 3.12 References

The main reference is to the paper describing theory behind these algorithms.

### 3.12.1 From the paper

### 3.12.2 Other references

## 3.13 Todo list

---

**Note:** There are also many #TODO comments scattered throughout the python files.

---

**Warning:** Auto-generated page! This does not look pretty.

---

### Todo

Make this available under an open-source license.

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/docs/index.rst`, line 38.)

---

### Todo

This could be made more efficient for large values of *power* by using knowledge of the component containing *key*.

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/thompson/automorphism.py:docstring of thompson.automorphism.Automorphism.repeated_image`, line 18.)

---

### Todo

This should be renamed to `component_type`.

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/thompson/automorphism.py:docstring of thompson.automorphism.Automorphism.orbit_type`, line 87.)

---

### Todo

This method is badly named; something like `test_revealed_by(basis)` would be better.

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/thompson/automorphism.py:docstring of thompson.automorphism.Automorphism.test_revealing`, line 21.)

---

### Todo

This method is badly named; something like `is_revealed_by(basis)` would be better.

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/thompson/automorphism.py:docstring of thompson.automorphism.Automorphism.is_revealing`, line 19.)

---

### Todo

use a Python binding to ImageMagick rather than just shelling out?

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/thompson/drawing/__init__.py:docstring of thompson.drawing.display_file`, line 14.)

---

### Todo

Add the named generators A, B, C of Thompson's T and V. Analogues for general  $G_{n,r}$ ?

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/docs/thompson.examples.rst`, line 19.)

---

### Todo

Maybe this should be 0 to order if the identity is permitted and 1 to order otherwise?

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/thompson/periodic.py:docstring of thompson.periodic.PeriodicAut.power_conjugacy_bounds`, line 5.)

---

### Todo

More convenient ways of inputting words, e.g. `x a1^3` instead of `x a1 a1 a1`.

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/thompson/word.py:docstring of thompson.word.from_string`, line 15.)

---

### Todo

I think this is a total order—try to prove this. I based the idea on [\[Zaks\]](#) (section 2, definition 1) which describes a total order on  $k$ -ary trees. On another note, isn't this similar to shortlex order?

---

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/thompsons-v/checkouts/stable/thompson/word.py:docstring of thompson.word.Word.__lt__`, line 53.)

---

---

**Indices**

---

- genindex



- 
- [BDR] N. Barker, A. J. Duncan, and D. M. Robertson, “The power conjugacy problem in Higman-Thompson groups”. (2015) Preprint available at [arXiv:1503.01032](https://arxiv.org/abs/1503.01032) [math.GR].
- [AS74] M. Anshel and P. Stebe, “The solvability of the conjugacy problem for certain HNN groups”, *Bull. Amer. Math. Soc.* , **80** (2) (1974) 266–270.
- [B87] K. S. Brown, “Finiteness properties of groups”, *Journal of Pure and Applied Algebra*, **44** (1987) 45-75.
- [BCR] J. Burillo, S. Cleary and C. E. Röver, “Obstructions for subgroups of Thompson’s group  $V$ ”, [arxiv.org/abs/1402.3860](https://arxiv.org/abs/1402.3860)
- [BGG11] C. Bleak, H. Bowman, A. Gordon, G. Graham, J. Hughes, F. Matucci and J. Sapir, “Centralizers in R.Thompson’s group  $V_n$ ”, *Groups, Geometry and Dynamics* **7** , No. 4 (2013), 821–865.
- [BK08] V.N. Bezverkhii, A.N. Kuznetsova, “Solvability of the power conjugacy problem for words in Artin groups of extra large type”, *Chebyshevskii Sb.* **9** (1) (2008) 50–68.
- [BM07] J. M. Belk and F. Matucci, “Conjugacy and dynamics in Thompson’s groups”, *Geom. Dedicata* **169** (1) (2014) 239–261.
- [BMOV] O. Bogopolski, A. Martino, O. Maslakova and E. Ventura, “The conjugacy problem is solvable in free-by-cyclic groups”, *Bulletin of the London Mathematical Society* , **38** , (10) (2006) 787–794.
- [Bar] N. Barker, “Topics in Algebra: The Higman-Thompson Group  $G_{2,1}$  and Beauville  $p$ -groups”, *Thesis, Newcastle University* (2014)
- [Bez14] N.V. Bezverkhii, “Ring Diagrams with Periodic Labels and Power Conjugacy Problem in Groups with Small Cancellation Conditions C (3) -T (6)”, *Science and Education of the Bauman MSTU* , **14** (11) (2014).
- [Brin04] M. G. Brin, “Higher dimensional Thompson groups”, *Geom. Dedicata* , **108** (2004) 163–192.
- [CFP96] J. W. Cannon, W.J. Floyd and W. R. Parry, “Introductory notes on Richard Thompson’s groups”, *Enseign. Math.* , (2) **42** (3–4) (1996) 215–256.
- [Cohn81] P. M. Cohn, “Universal Algebra”. Mathematics and its Applications, 6, D. Reidel Pub. Company, (1981).
- [Cohn91] P. M. Cohn, “Algebra, Volume 3”. J. Wiley, (1991).
- [Com77] L. P. J. Comerford, A note on power-conjugacy, *Houston J. Math.* **3** (1977), no. 3, 337—341.
- [DMP14] W. Dicks, C. Martínez-Pérez, “Isomorphisms of Brin-Higman-Thompson groups”, *Israel Journal of Mathematics* , **199** (2014), 189–218.
- [Fes14] A.V. Fesenko, “Vulnerability of Cryptographic Primitives Based on the Power Conjugacy Search Problem in Quantum Computing”, *Cybernetics and Systems Analysis* , **50** (5) (2014) 815–816.
-

- [Hig74] G. Higman, “Finitely presented infinite simple groups”, *Notes on Pure Mathematics*, Vol. 8 (1974).
- [JT61] B. Jónsson and A. Tarski, “On two properties of free algebras”, *Math. Scand.*, **9** (1961) 95–101.
- [KA09] D. Kahrobaei and M. Anshel, “Decision and Search in Non-Abelian Cramer-Shoup Public Key Cryptosystem”, *Groups-Complexity-Cryptology*, **1** (2) (2009) 217–225.
- [LM71] S. Lipschutz and C.F. Miller, “Groups with certain solvable and unsolvable decision problems”, *Comm. Pure Appl. Math.*, **24** (1971) 7–15.
- [Lot83] M. Lothaire, “Combinatorics on Words”, Addison-Wesley, Advanced Book Program, World Science Division, (1983).
- [MPN13] C. Martinez-Perez, B. Nucinkis, “Bredon cohomological finiteness conditions for generalisations of Thompson’s groups”, *Groups Geom. Dyn.* **7** (4) (2013) 931–959.
- [MT73] R. McKenzie and R. J. Thompson, “An elementary construction of unsolvable word problems in group theory”, *Word problems: decision problems and the Burnside problem in group theory*, Studies in Logic and the Foundations of Math., 71, pp. 457–478. North-Holland, Amsterdam, (1973).
- [Pa11] E. Pardo, “The isomorphism problem for Higman-Thompson groups”, *Journal of Algebra*, **344** (2011), 172–183.
- [Pr08] S.J. Pride, “On the residual finiteness and other properties of (relative) one-relator groups”, *Proc. Amer. Math. Soc.* **136** (2) (2008) 377–386.
- [R15] D. M. Robertson, “`thompson`: a package for Python 3.3+ to work with elements of the Higman-Thompson groups  $G_{n,r}$ ”. Source code available from [https://github.com/DMRRobertson/thompsons\\_v](https://github.com/DMRRobertson/thompsons_v) and documentation available from <http://thompsons-v.readthedocs.org/>.
- [SD10] O. P. Salazar-Diaz, “Thompson’s group  $V$  from a dynamical viewpoint”, *Internat. J. Algebra Comput.*, **1**, 39–70, 20, (2010).
- [Tho] R. J. Thompson, unpublished notes. <http://www.math.binghamton.edu/matt/thompson/index.html>
- [Bleak] C. Bleak *et al*, “Centralisers in the R. Thompson group  $V_n$ ”, *Groups, Geometry, and Dynamics* **7**, no. 4, pp. 821–865 (2013).
- [Kogan] R. Kogan, “`nVTrees Applet`” (2008). Accessed 17th September, 2014. Source code available on [GitHub](#).
- [Zaks] S. Zaks, “Lexicographic generation of ordered trees”, *Theoretical Computer Science* **10**: 63-82 (1980).

**t**

`thompson.automorphism`, 30  
`thompson.drawing`, 57  
`thompson.examples`, 50  
`thompson.generators`, 21  
`thompson.homomorphism`, 26  
`thompson.infinite`, 47  
`thompson.mixed`, 43  
`thompson.number_theory`, 9  
`thompson.orbits`, 42  
`thompson.periodic`, 46  
`thompson.word`, 12



## Symbols

`__and__()` (thompson.orbits.SolutionSet method), 42  
`__contains__()` (thompson.orbits.SolutionSet method), 41  
`__contains__()` (thompson.word.Signature method), 11  
`__eq__()` (thompson.homomorphism.Homomorphism method), 27  
`__getnewargs__()` (thompson.orbits.Characteristic method), 43  
`__getstate__()` (thompson.orbits.Characteristic method), 43  
`__init__()` (thompson.automorphism.Automorphism method), 31  
`__init__()` (thompson.generators.Generators method), 21  
`__init__()` (thompson.homomorphism.Homomorphism method), 26  
`__invert__()` (thompson.automorphism.Automorphism method), 32  
`__lt__()` (thompson.word.Word method), 16  
`__mul__()` (thompson.homomorphism.Homomorphism method), 28  
`__new__()` (thompson.orbits.Characteristic static method), 43  
`__new__()` (thompson.word.Signature static method), 11  
`__new__()` (thompson.word.Word static method), 16  
`__pow__()` (thompson.automorphism.Automorphism method), 32  
`__repr__()` (thompson.orbits.Characteristic method), 43  
`__str__()` (thompson.homomorphism.Homomorphism method), 29

## A

`add_relabellers()` (thompson.homomorphism.Homomorphism method), 29  
`alpha()` (thompson.word.Word method), 19  
`append()` (thompson.generators.Generators method), 21  
`are_contractible()` (in module thompson.word), 14  
`area_to_identity()` (thompson.automorphism.Automorphism method), 37

`as_interval()` (thompson.word.Word method), 18  
`Automorphism` (class in thompson.automorphism), 30  
`available_examples()` (in module thompson.examples), 51

## C

`centralise_period()` (thompson.automorphism.Automorphism method), 38  
`Characteristic` (class in thompson.orbits), 43  
`complete_infinite()` (thompson.orbits.ComponentType class method), 40  
`ComponentType` (class in thompson.orbits), 39  
`compute_bounds()` (thompson.infinite.InfiniteAut static method), 50  
`compute_quasinormal_basis()` (thompson.automorphism.Automorphism method), 33  
`copy()` (thompson.generators.Generators method), 21  
`cycled()` (thompson.generators.Generators method), 25  
`cycles_order()` (thompson.automorphism.Automorphism method), 36

## D

`descendants_above()` (thompson.generators.Generators method), 24  
`display_file()` (in module thompson.drawing), 57  
`divisors()` (in module thompson.number\_theory), 10  
`dump_QNB()` (thompson.automorphism.Automorphism method), 35

## E

`equivalence_data()` (thompson.infinite.InfiniteAut method), 48  
`expand()` (thompson.generators.Generators method), 25  
`expand()` (thompson.word.Word method), 19  
`expand_to_size()` (thompson.generators.Generators method), 25  
`extend()` (thompson.word.Word method), 19  
`extended_gcd()` (in module thompson.number\_theory), 9

## F

filter() (thompson.generators.Generators method), 21  
 find\_power\_conjugators() (thompson.infinite.InfiniteAut method), 49  
 find\_power\_conjugators() (thompson.mixed.MixedAut method), 45  
 find\_power\_conjugators() (thompson.periodic.PeriodicAut method), 47  
 forest() (in module thompson.drawing), 58  
 format() (in module thompson.word), 12  
 free\_factor() (thompson.mixed.MixedAut method), 44  
 free\_factors() (thompson.mixed.MixedAut method), 44  
 free\_monoid\_on\_alphas() (in module thompson.word), 15  
 from\_file() (thompson.homomorphism.Homomorphism class method), 27  
 from\_string() (in module thompson.word), 12  
 from\_string() (thompson.homomorphism.Homomorphism class method), 27

## G

gcd() (in module thompson.number\_theory), 9  
 generates\_algebra() (thompson.generators.Generators method), 23  
 Generators (class in thompson.generators), 21

## H

Homomorphism (class in thompson.homomorphism), 26

## I

identity() (thompson.homomorphism.Homomorphism class method), 28  
 image() (thompson.automorphism.Automorphism method), 31  
 image() (thompson.homomorphism.Homomorphism method), 28  
 image\_of\_set() (thompson.automorphism.Automorphism method), 32  
 image\_of\_set() (thompson.homomorphism.Homomorphism method), 29  
 incomplete() (thompson.orbits.ComponentType class method), 40  
 InfiniteAut (class in thompson.infinite), 47  
 is\_above() (thompson.generators.Generators method), 23  
 is\_above() (thompson.word.Word method), 17  
 is\_basis() (thompson.generators.Generators method), 23  
 is\_conjugate\_to() (thompson.automorphism.Automorphism method), 36  
 is\_empty() (thompson.orbits.SolutionSet method), 41  
 is\_free() (thompson.generators.Generators method), 22  
 is\_identity() (thompson.homomorphism.Homomorphism method), 28

is\_incomplete() (thompson.orbits.ComponentType method), 40  
 is\_isomorphic\_to() (thompson.word.Signature method), 11  
 is\_revealing() (thompson.automorphism.Automorphism method), 37  
 is\_semi\_infinite() (thompson.orbits.ComponentType method), 40  
 is\_sequence() (thompson.orbits.SolutionSet method), 41  
 is\_simple() (thompson.word.Word method), 17  
 is\_singleton() (thompson.orbits.SolutionSet method), 41  
 is\_the\_integers() (thompson.orbits.SolutionSet method), 41  
 is\_type\_A() (thompson.orbits.ComponentType method), 40  
 is\_type\_B() (thompson.orbits.ComponentType method), 40  
 is\_type\_C() (thompson.orbits.ComponentType method), 40

## L

lambda\_arguments() (in module thompson.word), 14  
 lcm() (in module thompson.number\_theory), 9  
 load\_all\_examples() (in module thompson.examples), 51  
 load\_example() (in module thompson.examples), 51  
 load\_example\_pair() (in module thompson.examples), 51

## M

max\_depth\_to() (thompson.word.Word method), 18  
 minimal\_expansion\_for() (thompson.generators.Generators method), 24  
 minimal\_partition() (thompson.infinite.InfiniteAut method), 49  
 MixedAut (class in thompson.mixed), 44  
 multiplier (thompson.orbits.Characteristic attribute), 43

## O

orbit\_type() (thompson.automorphism.Automorphism method), 33

## P

periodic() (thompson.orbits.ComponentType class method), 40  
 PeriodicAut (class in thompson.periodic), 46  
 plot() (in module thompson.drawing), 58  
 potential\_image\_endpoints() (thompson.infinite.InfiniteAut method), 48  
 power (thompson.orbits.Characteristic attribute), 43  
 power\_conjugacy\_bounds() (thompson.infinite.InfiniteAut method), 49  
 power\_conjugacy\_bounds() (thompson.periodic.PeriodicAut method), 47  
 preorder\_traversal() (thompson.generators.Generators method), 22

- preserves\_order() (thompson.automorphism.Automorphism method), 36  
 print\_characteristics() (thompson.automorphism.Automorphism method), 35  
 print\_component\_types() (in module thompson.orbits), 42  
 prod() (in module thompson.number\_theory), 10
- ## R
- random\_automorphism() (in module thompson.examples), 55  
 random\_basis() (in module thompson.examples), 54  
 random\_conjugate\_infinite\_pair() (in module thompson.examples), 55  
 random\_conjugate\_pair() (in module thompson.examples), 55  
 random\_conjugate\_periodic\_pair() (in module thompson.examples), 55  
 random\_infinite\_automorphism() (in module thompson.examples), 55  
 random\_periodic\_automorphism() (in module thompson.examples), 55  
 random\_power\_conjugate\_pair() (in module thompson.examples), 55  
 random\_signature() (in module thompson.examples), 54  
 random\_simple\_word() (in module thompson.examples), 54  
 relabel() (thompson.homomorphism.Homomorphism method), 29  
 repeated\_image() (thompson.automorphism.Automorphism method), 32  
 rewrite\_set() (in module thompson.generators), 26  
 rewrite\_word() (in module thompson.generators), 26  
 root() (in module thompson.word), 15  
 rsplit() (thompson.word.Word method), 20
- ## S
- save\_to\_file() (thompson.homomorphism.Homomorphism method), 27  
 semi\_infinite() (thompson.orbits.ComponentType class method), 40  
 semi\_infinite\_end\_points() (thompson.automorphism.Automorphism method), 34  
 seminormal\_form\_start\_point() (thompson.automorphism.Automorphism method), 33  
 share\_orbit() (thompson.automorphism.Automorphism method), 36  
 Signature (class in thompson.word), 11  
 simple\_words\_above() (thompson.generators.Generators method), 22  
 SolutionSet (class in thompson.orbits), 40  
 solve\_linear\_congruence() (in module thompson.number\_theory), 10  
 solve\_linear\_diophantine() (in module thompson.number\_theory), 10  
 sort\_mapping\_pair() (thompson.generators.Generators class method), 25  
 split() (thompson.word.Word method), 20  
 standard\_basis() (thompson.generators.Generators class method), 24  
 standard\_generator() (in module thompson.examples), 51  
 standardise() (in module thompson.word), 13  
 subwords() (thompson.word.Word method), 20
- ## T
- test\_above() (thompson.generators.Generators method), 23  
 test\_above() (thompson.word.Word method), 18  
 test\_conjugate\_to() (thompson.infinite.InfiniteAut method), 48  
 test\_conjugate\_to() (thompson.mixed.MixedAut method), 44  
 test\_conjugate\_to() (thompson.periodic.PeriodicAut method), 46  
 test\_free() (thompson.generators.Generators method), 21  
 test\_generates\_algebra() (thompson.generators.Generators method), 22  
 test\_power\_conjugate\_to() (thompson.infinite.InfiniteAut method), 49  
 test\_power\_conjugate\_to() (thompson.mixed.MixedAut method), 45  
 test\_power\_conjugate\_to() (thompson.periodic.PeriodicAut method), 47  
 test\_revealing() (thompson.automorphism.Automorphism method), 37  
 test\_semi\_infinite() (thompson.automorphism.Automorphism method), 34  
 thompson.automorphism (module), 30  
 thompson.drawing (module), 57  
 thompson.examples (module), 50  
 thompson.generators (module), 21  
 thompson.homomorphism (module), 26  
 thompson.infinite (module), 47  
 thompson.mixed (module), 43  
 thompson.number\_theory (module), 9  
 thompson.orbits (module), 42  
 thompson.periodic (module), 46  
 thompson.word (module), 12

## V

`validate()` (in module `thompson.word`), 12

## W

`Word` (class in `thompson.word`), 16