

**NAME**

XML::Twig – A perl module for processing huge XML documents in tree mode.

**SYNOPSIS**

Note that this documentation is intended as a reference to the module.

Complete docs, including a tutorial, examples, an easier to use HTML version, a quick reference card and a FAQ are available at <<http://www.xmltwig.org/xmltwig>>

Small documents (loaded in memory as a tree):

```
my $twig=XML::Twig->new();      # create the twig
$twig->parsefile( 'doc.xml');    # build it
my_process( $twig);             # use twig methods to process it
$twig->print;                    # output the twig
```

Huge documents (processed in combined stream/tree mode):

```
# at most one div will be loaded in memory
my $twig=XML::Twig->new(
    twig_handlers =>
        { title    => sub { $_->set_tag( 'h2') }, # change title tags to h2
          # $_ is the current element
          para     => sub { $_->set_tag( 'p') }, # change para to p
          hidden   => sub { $_->delete; },      # remove hidden elements
          list     => \&my_list_process,        # process list elements
          div      => sub { $_[0]->flush; },    # output and free memory
        },
    pretty_print => 'indented',                # output will be nicely formatted
    empty_tags  => 'html',                     # outputs <empty_tag />
);
$twig->parsefile( 'my_big.xml');

sub my_list_process
{ my( $twig, $list)= @_;
  # ...
}
```

See XML::Twig 101 for other ways to use the module, as a filter for example.

**DESCRIPTION**

This module provides a way to process XML documents. It is build on top of XML::Parser.

The module offers a tree interface to the document, while allowing you to output the parts of it that have been completely processed.

It allows minimal resource (CPU and memory) usage by building the tree only for the parts of the documents that need actual processing, through the use of the `twig_roots` and `twig_print_outside_roots` options. The `finish` and `finish_print` methods also help to increase performances.

XML::Twig tries to make simple things easy so it tries its best to takes care of a lot of the (usually) annoying (but sometimes necessary) features that come with XML and XML::Parser.

**TOOLS**

XML::Twig comes with a few command-line utilities:

**xml\_pp – xml pretty-printer**

XML pretty printer using XML::Twig

**xml\_grep – grep XML files looking for specific elements**

`xml_grep` does a grep on XML files. Instead of using regular expressions it uses XPath expressions (in fact the subset of XPath supported by XML::Twig).

**xml\_split – cut a big XML file into smaller chunks**

`xml_split` takes a (presumably big) XML file and split it in several smaller files, based on various criteria (level in the tree, size or an XPath expression)

**xml\_merge – merge back XML files split with `xml_split`**

`xml_merge` takes several xml files that have been split using `xml_split` and recreates a single file.

**xml\_spellcheck – spellcheck XML files**

`xml_spellcheck` lets you spell check the content of an XML file. It extracts the text (the content of elements and optionally of attributes), call a spell checker on it and then recreates the XML document.

**XML::Twig 101**

XML::Twig can be used either on “small” XML documents (that fit in memory) or on huge ones, by processing parts of the document and outputting or discarding them once they are processed.

**Loading an XML document and processing it**

```
my $t= XML::Twig->new();
$t->parse( '<d><title>title</title><para>p 1</para><para>p 2</para></d>');
my $root= $t->root;
$root->set_tag( 'html');           # change doc to html
$title= $root->first_child( 'title'); # get the title
$title->set_tag( 'h1');           # turn it into h1
my @para= $root->children( 'para'); # get the para children
foreach my $para (@para)
    { $para->set_tag( 'p'); }      # turn them into p
$t->print;                        # output the document
```

Other useful methods include:

`att: $elt->{'att'}->{'foo'}` return the `foo` attribute for an element,

`set_att: $elt->set_att( foo => "bar")` sets the `foo` attribute to the `bar` value,

`next_sibling: $elt->{next_sibling}` return the next sibling in the document (in the example `$title->{next_sibling}` is the first `para`, you can also (and actually should) use `$elt->next_sibling( 'para')` to get it

The document can also be transformed through the use of the `cut`, `copy`, `paste` and `move` methods: `$title->cut; $title->paste( after => $p);` for example

And much, much more, see `XML::Twig::Elt`.

**Processing an XML document chunk by chunk**

One of the strengths of XML::Twig is that it let you work with files that do not fit in memory (BTW storing an XML document in memory as a tree is quite memory-expensive, the expansion factor being often around 10).

To do this you can define handlers, that will be called once a specific element has been completely parsed. In these handlers you can access the element and process it as you see fit, using the navigation and the `cut-n-paste` methods, plus lots of convenient ones like `prefix` . Once the element is completely processed you can then `flush` it, which will output it and free the memory. You can also `purge` it if you don't need to output it (if you are just extracting some data from the document for example). The handler will be called again once the next relevant element has been parsed.

```
my $t= XML::Twig->new( twig_handlers =>
    { section => \&section,
      para   => sub { $_->set_tag( 'p'); }
    },
    );
$t->parsefile( 'doc.xml');

# the handler is called once a section is completely parsed, ie when
```

```
# the end tag for section is found, it receives the twig itself and
# the element (including all its sub-elements) as arguments
sub section
{ my( $t, $section)= @_;      # arguments for all twig_handlers
  $section->set_tag( 'div');    # change the tag name
  # let's use the attribute nb as a prefix to the title
  my $title= $section->first_child( 'title'); # find the title
  my $nb= $title->{'att'}->{'nb'}; # get the attribute
  $title->prefix( "$nb - ");    # easy isn't it?
  $section->flush;              # outputs the section and frees memory
}
```

There is of course more to it: you can trigger handlers on more elaborate conditions than just the name of the element, section/title for example.

```
my $t= XML::Twig->new( twig_handlers =>
                      { 'section/title' => sub { $_->print } }
                      )
->parsefile( 'doc.xml');
```

Here `sub { $_->print }` simply prints the current element (`$_` is aliased to the element in the handler).

You can also trigger a handler on a test on an attribute:

```
my $t= XML::Twig->new( twig_handlers =>
                      { 'section[@level="1"]' => sub { $_->print } }
                      );
->parsefile( 'doc.xml');
```

You can also use `start_tag_handlers` to process an element as soon as the start tag is found. Besides `prefix` you can also use `suffix` ,

### Processing just parts of an XML document

The `twig_roots` mode builds only the required sub-trees from the document. Anything outside of the twig roots will just be ignored:

```
my $t= XML::Twig->new(
  # the twig will include just the root and selected titles
  twig_roots => { 'section/title' => \&print_n_purge,
                  'annex/title'  => \&print_n_purge
                }
);
$t->parsefile( 'doc.xml');

sub print_n_purge
{ my( $t, $elt)= @_;
  print $elt->text;      # print the text (including sub-element texts)
  $t->purge;             # frees the memory
}
```

You can use that mode when you want to process parts of a documents but are not interested in the rest and you don't want to pay the price, either in time or memory, to build the tree for the it.

### Building an XML filter

You can combine the `twig_roots` and the `twig_print_outside_roots` options to build filters, which let you modify selected elements and will output the rest of the document as is.

This would convert prices in \$ to prices in Euro in a document:

```

my $t= XML::Twig->new(
    twig_roots    => { 'price' => \&convert, },    # process prices
    twig_print_outside_roots => 1,                # print the rest
    );
$t->parsefile( 'doc.xml' );

sub convert
{ my( $t, $price)= @_;
  my $currency= $price->{'att'}->{'currency'};      # get the currency
  if( $currency eq 'USD')
  { $usd_price= $price->text;                        # get the price
    # %rate is just a conversion table
    my $euro_price= $usd_price * $rate{usd2euro};
    $price->set_text( $euro_price);                  # set the new price
    $price->set_att( currency => 'EUR');              # don't forget this!
  }
  $price->print;                                     # output the price
}

```

### XML::Twig and various versions of Perl, XML::Parser and expat:

XML::Twig is a lot more sensitive to variations in versions of perl, XML::Parser and expat than to the OS, so this should cover some reasonable configurations.

The “recommended configuration” is perl 5.8.3+ (for good Unicode support), XML::Parser 2.31+ and expat 1.95.5+

See <<http://testers.cpan.org/search?request=dist&dist=XML-Twig>> for the CPAN testers reports on XML::Twig, which list all tested configurations.

An Atom feed of the CPAN Testers results is available at <[http://xmlltwig.org/rss/twig\\_testers.rss](http://xmlltwig.org/rss/twig_testers.rss)>

Finally:

XML::Twig does **NOT** work with expat 1.95.4

XML::Twig only works with XML::Parser 2.27 in perl 5.6.\*

Note that I can’t compile XML::Parser 2.27 anymore, so I can’t guarantee that it still works

XML::Parser 2.28 does not really work

When in doubt, upgrade expat, XML::Parser and Scalar::Util

Finally, for some optional features, XML::Twig depends on some additional modules. The complete list, which depends somewhat on the version of Perl that you are running, is given by running `t/zz_dump_config.t`

## Simplifying XML processing

### Whitespaces

Whitespaces that look non-significant are discarded, this behaviour can be controlled using the `keep_spaces`, `keep_spaces_in` and `discard_spaces_in` options.

### Encoding

You can specify that you want the output in the same encoding as the input (provided you have valid XML, which means you have to specify the encoding either in the document or when you create the Twig object) using the `keep_encoding` option

You can also use `output_encoding` to convert the internal UTF-8 format to the required encoding.

### Comments and Processing Instructions (PI)

Comments and PI’s can be hidden from the processing, but still appear in the output (they are carried by the “real” element closer to them)

### Pretty Printing

XML::Twig can output the document pretty printed so it is easier to read for us humans.

### Surviving an untimely death

XML parsers are supposed to react violently when fed improper XML. XML::Parser just dies.

XML::Twig provides the `safe_parse` and the `safe_parsefile` methods which wrap the `parse` in an `eval` and return either the parsed twig or 0 in case of failure.

### Private attributes

Attributes with a name starting with `#` (illegal in XML) will not be output, so you can safely use them to store temporary values during processing. Note that you can store anything in a private attribute, not just text, it's just a regular Perl variable, so a reference to an object or a huge data structure is perfectly fine.

## CLASSES

XML::Twig uses a very limited number of classes. The ones you are most likely to use are `XML::Twig` of course, which represents a complete XML document, including the document itself (the root of the document itself is `root`), its handlers, its input or output filters... The other main class is `XML::Twig::Elt`, which models an XML element. Element here has a very wide definition: it can be a regular element, or but also text, with an element tag of `#PCDATA` (or `#CDATA`), an entity (tag is `#ENT`), a Processing Instruction (`#PI`), a comment (`#COMMENT`).

Those are the 2 commonly used classes.

You might want to look the `elt_class` option if you want to subclass `XML::Twig::Elt`.

Attributes are just attached to their parent element, they are not objects per se. (Please use the provided methods `att` and `set_att` to access them, if you access them as a hash, then your code becomes implementation dependent and might break in the future).

Other classes that are seldom used are `XML::Twig::Entity_list` and `XML::Twig::Entity`.

If you use `XML::Twig::XPath` instead of `XML::Twig`, elements are then created as `XML::Twig::XPath::Elt`

## METHODS

### XML::Twig

A twig is a subclass of `XML::Parser`, so all `XML::Parser` methods can be called on a twig object, including `parse` and `parsefile`. `setHandlers` on the other hand cannot be used, see `BUGS`

#### new

This is a class method, the constructor for `XML::Twig`. Options are passed as keyword value pairs. Recognized options are the same as `XML::Parser`, plus some (in fact a lot!) `XML::Twig` specifics.

#### New Options:

##### twig\_handlers

This argument consists of a hash { `expression = \&handler` }> where `expression` is an *XPath-like expression* (+ some others).

XPath expressions are limited to using the child and descendant axis (indeed you can't specify an axis), and predicates cannot be nested. You can use the `string`, or `string(<tag>)` function (except in `twig_roots` triggers).

Additionally you can use regexps (/ delimited) to match attribute and string values.

Examples:

```

foo
foo/bar
foo//bar
/foo/bar
/foo//bar
/foo/bar[@att1 = "val1" and @att2 = "val2"]/baz[@a >= 1]
foo[string()=~ /^duh!+/]
/foo[string(bar)=~ /\d+\/]/baz[@att != 3]

```

#CDATA can be used to call a handler for a CDATA section. #COMMENT can be used to call a handler for comments

Some additional (non-XPath) expressions are also provided for convenience:

processing instructions

'?' or '#PI' triggers the handler for any processing instruction, and '?<target>' or '#PI <target>' triggers a handler for processing instruction with the given target( ex: '#PI xml-styleSheet').

level(<level>)

Triggers the handler on any element at that level in the tree (root is level 1)

\_all\_

Triggers the handler for **all** elements in the tree

\_default\_

Triggers the handler for each element that does NOT have any other handler.

Expressions are evaluated against the input document. Which means that even if you have changed the tag of an element (changing the tag of a parent element from a handler for example) the change will not impact the expression evaluation. There is an exception to this: “private” attributes (which name start with a '#', and can only be created during the parsing, as they are not valid XML) are checked against the current twig.

Handlers are triggered in fixed order, sorted by their type (xpath expressions first, then regexps, then level), then by whether they specify a full path (starting at the root element) or not, then by number of steps in the expression, then number of predicates, then number of tests in predicates. Handlers where the last step does not specify a step (foo/bar/\*) are triggered after other XPath handlers. Finally \_all\_ handlers are triggered last.

**Important:** once a handler has been triggered if it returns 0 then no other handler is called, except a \_all\_ handler which will be called anyway.

If a handler returns a true value and other handlers apply, then the next applicable handler will be called. Repeat, rinse, lather.; The exception to that rule is when the do\_not\_chain\_handlers option is set, in which case only the first handler will be called.

Note that it might be a good idea to explicitly return a short true value (like 1) from handlers: this ensures that other applicable handlers are called even if the last statement for the handler happens to evaluate to false. This might also speedup the code by avoiding the result of the last statement of the code to be copied and passed to the code managing handlers. It can really pay to have 1 instead of a long string returned.

When the closing tag for an element is parsed the corresponding handler is called, with 2 arguments: the twig and the Element . The twig includes the document tree that has been built so far, the element is the complete sub-tree for the element. The fact that the handler is called only when the closing tag for the element is found means that handlers for inner elements are called before handlers for outer elements.

\$\_ is also set to the element, so it is easy to write inline handlers like

```
para => sub { $_->set_tag( 'p'); }
```

Text is stored in elements whose tag name is #PCDATA (due to mixed content, text and sub-element in an element there is no way to store the text as just an attribute of the enclosing element).

**Warning:** if you have used `purge` or `flush` on the twig the element might not be complete, some of its children might have been entirely flushed or purged, and the start tag might even have been printed (by `flush`) already, so changing its tag might not give the expected result.

#### twig\_roots

This argument let's you build the tree only for those elements you are interested in.

```
Example: my $t= XML::Twig->new( twig_roots => { title => 1, subtitle => 1 },
    $t->parsefile( file);
my $t= XML::Twig->new( twig_roots => { 'section/title' => 1});
$t->parsefile( file);
```

return a twig containing a document including only `title` and `subtitle` elements, as children of the root element.

You can use *generic\_attribute\_condition*, *attribute\_condition*, *full\_path*, *partial\_path*, *tag*, *tag\_regexp*, *\_default\_* and *\_all\_* to trigger the building of the twig. *string\_condition* and *regexp\_condition* cannot be used as the content of the element, and the string, have not yet been parsed when the condition is checked.

**WARNING:** path are checked for the document. Even if the `twig_roots` option is used they will be checked against the full document tree, not the virtual tree created by `XML::Twig`

**WARNING:** `twig_roots` elements should NOT be nested, that would hopelessly confuse `XML::Twig` ;--(

Note: you can set handlers (`twig_handlers`) using `twig_roots`

```
Example: my $t= XML::Twig->new( twig_roots =>
    { title => sub { $_[1]->print; },
      subtitle => \&process_subtitle
    }
);
$t->parsefile( file);
```

#### twig\_print\_outside\_roots

To be used in conjunction with the `twig_roots` argument. When set to a true value this will print the document outside of the `twig_roots` elements.

```
Example: my $t= XML::Twig->new( twig_roots => { title => \&number_title },
    twig_print_outside_roots => 1,
);
$t->parsefile( file);
{ my $nb;
sub number_title
{ my( $twig, $title);
  $nb++;
  $title->prefix( "$nb ");
  $title->print;
}
}
```

This example prints the document outside of the title element, calls `number_title` for each `title` element, prints it, and then resumes printing the document. The twig is built only for the `title` elements.

If the value is a reference to a file handle then the document outside the `twig_roots` elements will be output to this file handle:

```
open( my $out, '>', 'out_file.xml') or die "cannot open out file.xml out_
my $t= XML::Twig->new( twig_roots => { title => \&number_title },
                        # default output to $out
                        twig_print_outside_roots => $out,
                      );

{ my $nb;
  sub number_title
  { my( $twig, $title);
    $nb++;
    $title->prefix( "$nb ");
    $title->print( $out);      # you have to print to \*OUT here
  }
}
```

#### start\_tag\_handlers

A hash { `expression = \&handler`}>. Sets element handlers that are called when the element is open (at the end of the XML::Parser Start handler). The handlers are called with 2 params: the twig and the element. The element is empty at that point, its attributes are created though.

You can use *generic\_attribute\_condition*, *attribute\_condition*, *full\_path*, *partial\_path*, *tag*, *tag\_regexp*, *\_default\_* and *\_all\_* to trigger the handler.

*string\_condition* and *regexp\_condition* cannot be used as the content of the element, and the string, have not yet been parsed when the condition is checked.

The main uses for those handlers are to change the tag name (you might have to do it as soon as you find the open tag if you plan to flush the twig at some point in the element, and to create temporary attributes that will be used when processing sub-element with *twig\_handlers*.

You should also use it to change tags if you use *flush*. If you change the tag in a regular *twig\_handler* then the start tag might already have been flushed.

**Note:** *start\_tag* handlers can be called outside of *twig\_roots* if this argument is used, in this case handlers are called with the following arguments: *\$t* (the twig), *\$tag* (the tag of the element) and *%att* (a hash of the attributes of the element).

If the *twig\_print\_outside\_roots* argument is also used, if the last handler called returns a *true* value, then the start tag will be output as it appeared in the original document, if the handler returns a *false* value then the start tag will **not** be printed (so you can print a modified string yourself for example).

Note that you can use the *ignore* method in *start\_tag\_handlers* (and only there).

#### end\_tag\_handlers

A hash { `expression = \&handler`}>. Sets element handlers that are called when the element is closed (at the end of the XML::Parser End handler). The handlers are called with 2 params: the twig and the tag of the element.

*twig\_handlers* are called when an element is completely parsed, so why have this redundant option? There is only one use for *end\_tag\_handlers*: when using the *twig\_roots* option, to trigger a handler for an element **outside** the roots. It is for example very useful to number titles in a document using nested sections:



```

my @no= (0);
my $no;
my $t= XML::Twig->new(
    start_tag_handlers =>
        { section => sub { $no[$#no]++; $no= join '.', @no; push @no, 0; },
    twig_roots          =>
        { title    => sub { $_[1]->prefix( $no); $_[1]->print; } },
    end_tag_handlers    => { section => sub { pop @no; } },
    twig_print_outside_roots => 1
);
$t->parsefile( $file);

```

Using the `end_tag_handlers` argument without `twig_roots` will result in an error.

#### `do_not_chain_handlers`

If this option is set to a true value, then only one handler will be called for each element, even if several satisfy the condition

Note that the `_all_` handler will still be called regardless

#### `ignore_elts`

This option lets you ignore elements when building the twig. This is useful in cases where you cannot use `twig_roots` to ignore elements, for example if the element to ignore is a sibling of elements you are interested in.

Example:

```

my $twig= XML::Twig->new( ignore_elts => { elt => 'discard' });
$twig->parsefile( 'doc.xml');

```

This will build the complete twig for the document, except that all `elt` elements (and their children) will be left out.

The keys in the hash are triggers, limited to the same subset as `start_tag_handlers`. The values can be `discard`, to discard the element, `print`, to output the element as-is, `string` to store the text of the ignored element(s), including markup, in a field of the twig: `$t->{twig_buffered_string}` or a reference to a scalar, in which case the text of the ignored element(s), including markup, will be stored in the scalar. Any other value will be treated as `discard`.

#### `char_handler`

A reference to a subroutine that will be called every time PCDATA is found.

The subroutine receives the string as argument, and returns the modified string:

```

# we want all strings in upper case
sub my_char_handler
{ my( $text)= @_;
  $text= uc( $text);
  return $text;
}

```

#### `elt_class`

The name of a class used to store elements. this class should inherit from `XML::Twig::Elt` (and by default it is `XML::Twig::Elt`). This option is used to subclass the element class and extend it with new methods.

This option is needed because during the parsing of the XML, elements are created by `XML::Twig`, without any control from the user code.

**keep\_atts\_order**

Setting this option to a true value causes the attribute hash to be tied to a `Tie::IxHash` object. This means that `Tie::IxHash` needs to be installed for this option to be available. It also means that the hash keeps its order, so you will get the attributes in order. This allows outputting the attributes in the same order as they were in the original document.

**keep\_encoding**

This is a (slightly?) evil option: if the XML document is not UTF-8 encoded and you want to keep it that way, then setting `keep_encoding` will use the `Expat` `original_string` method for character, thus keeping the original encoding, as well as the original entities in the strings.

See the `t/test6.t` test file to see what results you can expect from the various encoding options.

**WARNING:** if the original encoding is multi-byte then attribute parsing will be EXTREMELY unsafe under any Perl before 5.6, as it uses regular expressions which do not deal properly with multi-byte characters. You can specify an alternate function to parse the start tags with the `parse_start_tag` option (see below)

**WARNING:** this option is NOT used when parsing with the non-blocking parser (`parse_start`, `parse_more`, `parse_done` methods) which you probably should not use with `XML::Twig` anyway as they are totally untested!

**output\_encoding**

This option generates an `output_filter` using `Encode`, `Text::Iconv` or `Unicode::Map8` and `Unicode::Strings`, and sets the encoding in the XML declaration. This is the easiest way to deal with encodings, if you need more sophisticated features, look at `output_filter` below

**output\_filter**

This option is used to convert the character encoding of the output document. It is passed either a string corresponding to a predefined filter or a subroutine reference. The filter will be called every time a document or element is processed by the “print” functions (`print`, `sprint`, `flush`).

Pre-defined filters:

**latin1**

uses either `Encode`, `Text::Iconv` or `Unicode::Map8` and `Unicode::String` or a `regex` (which works only with `XML::Parser 2.27`), in this order, to convert all characters to ISO-8859-15 (usually `latin1` is synonym to ISO-8859-1, but in practice it seems that ISO-8859-15, which includes the euro sign, is more useful and probably what most people want).

**html**

does the same conversion as `latin1`, plus encodes entities using `HTML::Entities` (oddly enough you will need to have `HTML::Entities` installed for it to be available). This should only be used if the tags and attribute names themselves are in US-ASCII, or they will be converted and the output will not be valid XML any more

**safe**

converts the output to ASCII (US) only plus *character entities* (`&#nnn;`) this should be used only if the tags and attribute names themselves are in US-ASCII, or they will be converted and the output will not be valid XML any more

**safe\_hex**

same as `safe` except that the character entities are in hex (`&#xnnn;`)

**encode\_convert (\$encoding)**

Return a subref that can be used to convert utf8 strings to `$encoding`. Uses `Encode`.

```
my $conv = XML::Twig::encode_convert( 'latin1');
my $t = XML::Twig->new(output_filter => $conv);
```

#### iconv\_convert(\$encoding)

this function is used to create a filter subroutine that will be used to convert the characters to the target encoding using `Text::Iconv` (which needs to be installed, look at the documentation for the module and for the `iconv` library to find out which encodings are available on your system)

```
my $conv = XML::Twig::iconv_convert( 'latin1');
my $t = XML::Twig->new(output_filter => $conv);
```

#### unicode\_convert(\$encoding)

this function is used to create a filter subroutine that will be used to convert the characters to the target encoding using `Unicode::Strings` and `Unicode::Map8` (which need to be installed, look at the documentation for the modules to find out which encodings are available on your system)

```
my $conv = XML::Twig::unicode_convert( 'latin1');
my $t = XML::Twig->new(output_filter => $conv);
```

The `text` and `att` methods do not use the filter, so their result are always in unicode.

Those predeclared filters are based on subroutines that can be used by themselves (as `XML::Twig::foo`).

#### html\_encode(\$string)

Use `HTML::Entities` to encode a utf8 string

#### safe\_encode(\$string)

Use either a regexp (perl < 5.8) or `Encode` to encode non-ascii characters in the string in `&#<nnnn>;` format

#### safe\_encode\_hex(\$string)

Use either a regexp (perl < 5.8) or `Encode` to encode non-ascii characters in the string in `&#x<nnnn>;` format

#### regexp2latin1(\$string)

Use a regexp to encode a utf8 string into latin 1 (ISO-8859-1). Does not work with Perl 5.8.0!

#### output\_text\_filter

same as `output_filter`, except it doesn't apply to the brackets and quotes around attribute values. This is useful for all filters that could change the tagging, basically anything that does not just change the encoding of the output. `html`, `safe` and `safe_hex` are better used with this option.

#### input\_filter

This option is similar to `output_filter` except the filter is applied to the characters before they are stored in the twig, at parsing time.

#### remove\_cdata

Setting this option to a true value will force the twig to output CDATA sections as regular (escaped) PCDATA

#### parse\_start\_tag

If you use the `keep_encoding` option then this option can be used to replace the default parsing function. You should provide a coderef (a reference to a subroutine) as the argument, this subroutine takes the original tag (given by `XML::Parser::Expat original_string()` method) and returns a tag and the attributes in a hash (or in a list `attribute_name/attribute value`).

#### expand\_external\_ents

When this option is used external entities (that are defined) are expanded when the document is output using "print" functions such as `print`, `sprint`, `flush` and `xml_string`.

Note that in the twig the entity will be stored as an element with a tag '#ENT', the entity will not be expanded there, so you might want to process the entities before outputting it.

If an external entity is not available, then the parse will fail.

A special case is when the value of this option is `-1`. In that case a missing entity will not cause the parser to die, but its name, `sysid` and `pubid` will be stored in the twig as `$twig->{twig_missing_system_entities}` (a reference to an array of hashes { `name => <name>`, `sysid => <sysid>`, `pubid => <pubid>` }). Yes, this is a bit of a hack, but it's useful in some cases.

**WARNING:** setting `expand_external_ents` to `0` or `-1` currently doesn't work as expected; cf. <https://rt.cpan.org/Public/Bug/Display.html?id=118097>. To completely turn off expanding external entities use `no_xxe`.

#### `no_xxe`

If this argument is set to a true value, expanding of external entities is turned off.

#### `load_DTD`

If this argument is set to a true value, `parse` or `parsefile` on the twig will load the DTD information. This information can then be accessed through the twig, in a `DTD_handler` for example. This will load even an external DTD.

Default and fixed values for attributes will also be filled, based on the DTD.

Note that to do this the module will generate a temporary file in the current directory. If this is a problem let me know and I will add an option to specify an alternate directory.

See "DTD Handling" for more information

#### `DTD_base <path_to_DTD_directory>`

If the DTD is in a different directory, looks for it there, useful to make up somewhat for the lack of catalog suport in `expat`. You still need a `SYSTEM` declaration

#### `DTD_handler`

Set a handler that will be called once the doctype (and the DTD) have been loaded, with 2 arguments, the twig and the DTD.

#### `no_prolog`

Does not output a prolog (XML declaration and DTD)

**id** This optional argument gives the name of an attribute that can be used as an ID in the document. Elements whose ID is known can be accessed through the `elt_id` method. `id` defaults to `'id'`. See **BUGS**

#### `discard_spaces`

If this optional argument is set to a true value then spaces are discarded when they look non-significant: strings containing only spaces and at least one line feed are discarded. This argument is set to true by default.

The exact algorithm to drop spaces is: strings including only spaces (`perl \s`) and at least one `\n` right before an open or close tag are dropped.

#### `discard_all_spaces`

If this argument is set to a true value, spaces are discarded more aggressively than with `discard_spaces`: strings not including a `\n` are also dropped. This option is appropriate for data-oriented XML.

#### `keep_spaces`

If this optional argument is set to a true value then all spaces in the document are kept, and stored as `PCDATA`.

**Warning:** adding this option can result in changes in the twig generated: space that was previously discarded might end up in a new text element. see the difference by calling the

following code with 0 and 1 as arguments:

```
perl -MXML::Twig -e'print XML::Twig->new( keep_spaces => shift)->parse( '
```

keep\_spaces and discard\_spaces cannot be both set.

#### discard\_spaces\_in

This argument sets keep\_spaces to true but will cause the twig builder to discard spaces in the elements listed.

The syntax for using this argument is:

```
XML::Twig->new( discard_spaces_in => [ 'elt1', 'elt2' ] );
```

#### keep\_spaces\_in

This argument sets discard\_spaces to true but will cause the twig builder to keep spaces in the elements listed.

The syntax for using this argument is:

```
XML::Twig->new( keep_spaces_in => [ 'elt1', 'elt2' ] );
```

**Warning:** adding this option can result in changes in the twig generated: space that was previously discarded might end up in a new text element.

#### pretty\_print

Set the pretty print method, amongst 'none' (default), 'nsgmls', 'nice', 'indented', 'indented\_c', 'indented\_a', 'indented\_close\_tag', 'cvs', 'wrapped', 'record' and 'record\_c'

pretty\_print formats:

##### none

The document is output as one ling string, with no line breaks except those found within text elements

##### nsgmls

Line breaks are inserted in safe places: that is within tags, between a tag and an attribute, between attributes and before the > at the end of a tag.

This is quite ugly but better than none, and it is very safe, the document will still be valid (conforming to its DTD).

This is how the SGML parser sgmls splits documents, hence the name.

##### nice

This option inserts line breaks before any tag that does not contain text (so element with textual content are not broken as the \n is the significant).

**WARNING:** this option leaves the document well-formed but might make it invalid (not conformant to its DTD). If you have elements declared as

```
<!ELEMENT foo (#PCDATA|bar)>
```

then a foo element including a bar one will be printed as

```
<foo>
<bar>bar is just pcddata</bar>
</foo>
```

This is invalid, as the parser will take the line break after the foo tag as a sign that the element contains PCDATA, it will then die when it finds the bar tag. This may or may not be important for you, but be aware of it!

**indented**

Same as `nice` (and with the same warning) but indents elements according to their level

**indented\_c**

Same as `indented` but a little more compact: the closing tags are on the same line as the preceding text

**indented\_close\_tag**

Same as `indented` except that the closing tag is also indented, to line up with the tags within the element

**indented\_a**

This formats XML files in a line-oriented version control friendly way. The format is described in [<http://tinyurl.com/2kwscq>](http://tinyurl.com/2kwscq) (that's an Oracle document with an insanely long URL).

Note that to be totally conformant to the “spec”, the order of attributes should not be changed, so if they are not already in alphabetical order you will need to use the `keep_atts_order` option.

`cvs` Same as `indented_a`.

**wrapped**

Same as `indented_c` but lines are wrapped using `Text::Wrap::wrap`. The default length for lines is the default for `$Text::Wrap::columns`, and can be changed by changing that variable.

**record**

This is a record-oriented pretty print, that display data in records, one field per line (which looks a LOT like `indented`)

**record\_c**

Stands for record compact, one record per line

**empty\_tags**

Set the empty tag display style (`'normal'`, `'html'` or `'expand'`).

`normal` outputs an empty tag `'<tag/>'`, `html` adds a space `'<tag />'` for elements that can be empty in XHTML and `expand` outputs `'<tag></tag>'`

**quote**

Set the quote character for attributes (`'single'` or `'double'`).

**escape\_gt**

By default `XML::Twig` does not escape the character `>` in its output, as it is not mandated by the XML spec. With this option on, `>` will be replaced by `&gt;`;

**comments**

Set the way comments are processed: `'drop'` (default), `'keep'` or `'process'`

Comments processing options:

**drop**

drops the comments, they are not read, nor printed to the output

**keep**

comments are loaded and will appear on the output, they are not accessible within the twig and will not interfere with processing though

**Note:** comments in the middle of a text element such as

```
<p>text <!-- comment --> more text --></p>
```

are kept at their original position in the text. Using “print” methods like `print` or `sprint` will return the comments in the text. Using `text` or `field` on the other hand will not.

Any use of `set_pCDATA` on the `#PCDATA` element (directly or through other methods like `set_content`) will delete the comment(s).

#### `process`

comments are loaded in the twig and will be treated as regular elements (their tag is `#COMMENT`) this can interfere with processing if you expect `$elt->{first_child}` to be an element but find a comment there. Validation will not protect you from this as comments can happen anywhere. You can use `$elt->first_child( 'tag' )` (which is a good habit anyway) to get where you want.

Consider using `process` if you are outputting SAX events from `XML::Twig`.

**pi** Set the way processing instructions are processed: `'drop'`, `'keep'` (default) or `'process'`

Note that you can also set PI handlers in the `twig_handlers` option:

```
'?'          => \&handler
'?target'    => \&handler 2
```

The handlers will be called with 2 parameters, the twig and the PI element if `pi` is set to `process`, and with 3, the twig, the target and the data if `pi` is set to `keep`. Of course they will not be called if `pi` is set to `drop`.

If `pi` is set to `keep` the handler should return a string that will be used as-is as the PI text (it should look like `"<?target data?>"` or `""` if you want to remove the PI),

Only one handler will be called, `?target` or `?` if no specific handler for that target is available.

#### `map_xmlns`

This option is passed a hashref that maps uri's to prefixes. The prefixes in the document will be replaced by the ones in the map. The mapped prefixes can (actually have to) be used to trigger handlers, navigate or query the document.

Here is an example:

```
my $t= XML::Twig->new( map_xmlns => {'http://www.w3.org/2000/svg' => "svg",
                                     twig_handlers =>
                                     { 'svg:circle' => sub { $_->set_att( r => 20) } },
                                     pretty_print => 'indented',
                                   )
->parse( '<doc xmlns:gr="http://www.w3.org/2000/svg">
        <gr:circle cx="10" cy="90" r="10"/>
      </doc>'
       )
->print;
```

This will output:

```
<doc xmlns:svg="http://www.w3.org/2000/svg">
  <svg:circle cx="10" cy="90" r="20"/>
</doc>
```

#### `keep_original_prefix`

When used with `map_xmlns` this option will make `XML::Twig` use the original namespace prefixes when outputting a document. The mapped prefix will still be used for triggering handlers and in navigation and query methods.

```

my $t= XML::Twig->new( map_xmlns => {'http://www.w3.org/2000/svg' => "svg"},
                        twig_handlers =>
                        { 'svg:circle' => sub { $_->set_att( r => 20) } },
                        keep_original_prefix => 1,
                        pretty_print => 'indented',
                        )
->parse( '<doc xmlns:gr="http://www.w3.org/2000/svg">
        <gr:circle cx="10" cy="90" r="10"/>
      </doc>'
        )
->print;

```

This will output:

```

<doc xmlns:gr="http://www.w3.org/2000/svg">
  <gr:circle cx="10" cy="90" r="20"/>
</doc>

```

`original_uri ($prefix)`

called within a handler, this will return the uri bound to the namespace prefix in the original document.

`index ($arrayref or $hashref)`

This option creates lists of specific elements during the parsing of the XML. It takes a reference to either a list of triggering expressions or to a hash name => expression, and for each one generates the list of elements that match the expression. The list can be accessed through the `index` method.

example:

```

# using an array ref
my $t= XML::Twig->new( index => [ 'div', 'table' ] )
->parsefile( "foo.xml");
my $divs= $t->index( 'div');
my $first_div= $divs->[0];
my $last_table= $t->index( table => -1);

# using a hashref to name the indexes
my $t= XML::Twig->new( index => { email => 'a[@href=~/^ \s*mailto:/' } )
->parsefile( "foo.xml");
my $last_emails= $t->index( email => -1);

```

Note that the index is not maintained after the parsing. If elements are deleted, renamed or otherwise hurt during processing, the index is NOT updated. (changing the id element OTOH will update the index)

`att_accessors <list of attribute names>`

creates methods that give direct access to attribute:

```

my $t= XML::Twig->new( att_accessors => [ 'href', 'src' ] )
->parsefile( $file);
my $first_href= $t->first_elt( 'img')->src; # same as ->att( 'src')
$t->first_elt( 'img')->src( 'new_logo.png') # changes the attribute value

```

`elt_accessors`

creates methods that give direct access to the first child element (in scalar context) or the list of elements (in list context):

the list of accessors to create can be given 1 2 different ways: in an array, or in a hash alias => expression



```

my $t= XML::Twig->new( elt_accessors => [ 'head' ])
    ->parsefile( $file);
my $title_text= $t->root->head->field( 'title');
# same as $title_text= $t->root->first_child( 'head')->field( 'title');

my $t= XML::Twig->new( elt_accessors => { warnings => 'p[@class="warning'
    ->parsefile( $file);
my $body= $t->first_elt( 'body');
my @warnings= $body->warnings; # same as $body->children( 'p[@class="warning'
my $s2= $body->d2; # same as $body->first_child( 'div[2]')

```

#### field\_accessors

creates methods that give direct access to the first child element text:

```

my $t= XML::Twig->new( field_accessors => [ 'h1' ])
    ->parsefile( $file);
my $div_title_text= $t->first_elt( 'div')->title;
# same as $title_text= $t->first_elt( 'div')->field( 'title');

```

#### use\_tidy

set this option to use HTML::Tidy instead of HTML::TreeBuilder to convert HTML to XML. HTML, especially real (real “crap”) HTML found in the wild, so depending on the data, one module or the other does a better job at the conversion. Also, HTML::Tidy can be a bit difficult to install, so XML::Twig offers both option. TIMTOWTDI

#### output\_html\_doctype

when using HTML::TreeBuilder to convert HTML, this option causes the DOCTYPE declaration to be output, which may be important for some legacy browsers. Without that option the DOCTYPE definition is NOT output. Also if the definition is completely wrong (ie not easily parsable), it is not output either.

**Note:** I \_HATE\_ the Java-like name of arguments used by most XML modules. So in pure TIMTOWTDI fashion all arguments can be written either as UglyJavaLikeName or as readable\_perl\_name: twig\_print\_outside\_roots or TwigPrintOutsideRoots (or even twigPrintOutsideRoots {shudder}). XML::Twig normalizes them before processing them.

#### parse( \$source)

The \$source parameter should either be a string containing the whole XML document, or it should be an open IO::Handle (aka a filehandle).

A die call is thrown if a parse error occurs. Otherwise it will return the twig built by the parse. Use safe\_parse if you want the parsing to return even when an error occurs.

If this method is called as a class method (XML::Twig->parse( \$some\_xml\_or\_html)) then an XML::Twig object is created, using the parameters except the last one (eg XML::Twig->parse( pretty\_print => 'indented', \$some\_xml\_or\_html)) and xparse is called on it.

Note that when parsing a filehandle, the handle should NOT be open with an encoding (ie open with open( my \$in, '<', \$filename)). The file will be parsed by expat, so specifying the encoding actually causes problems for the parser (as in: it can crash it, see <https://rt.cpan.org/Ticket/Display.html?id=78877>). For parsing a file it is actually recommended to use parsefile on the file name, instead of <parse> on the open file.

#### parsestring

This is just an alias for parse for backwards compatibility.

#### parsefile (FILE [, OPT => OPT\_VALUE [...]])

Open FILE for reading, then call parse with the open handle. The file is closed no matter how parse returns.

A die call is thrown if a parse error occurs. Otherwise it will return the twig built by the parse. Use

`safe_parsefile` if you want the parsing to return even when an error occurs.

`parsefile_inplace ( $file, $optional_extension)`

Parse and update a file “in place”. It does this by creating a temp file, selecting it as the default for **print()** statements (and methods), then parsing the input file. If the parsing is successful, then the temp file is moved to replace the input file.

If an extension is given then the original file is backed-up (the rules for the extension are the same as the rule for the `-i` option in perl).

`parsefile_html_inplace ( $file, $optional_extension)`

Same as `parsefile_inplace`, except that it parses HTML instead of XML

`parseurl ($url $optional_user_agent)`

Gets the data from `$url` and parse it. The data is piped to the parser in chunks the size of the `XML::Parser::Expat` buffer, so memory consumption and hopefully speed are optimal.

Note that `parseurl` forks a child process that calls `exit` once the data has been retrieved, which can interfere with locks. If that’s a problem, see below:

For most (read “small”) XML it is probably as efficient (and easier to debug) to just get the XML file and then parse it as a string.

```
use XML::Twig;
use LWP::Simple;
my $twig= XML::Twig->new();
$twig->parse( LWP::Simple::get( $URL ) );
```

or more simply to call `nparsed`

```
use XML::Twig;
my $twig= XML::Twig->nparsed( $URL );
```

If the `$optional_user_agent` argument is passed to the method then it is used, otherwise a new one is created.

`safe_parse ( SOURCE [, OPT => OPT_VALUE [...]] )`

This method is similar to `parse` except that it wraps the parsing in an `eval` block. It returns the twig on success and 0 on failure (the twig object also contains the parsed twig). `$@` contains the error message on failure.

Note that the parsing still stops as soon as an error is detected, there is no way to keep going after an error.

`safe_parsefile (FILE [, OPT => OPT_VALUE [...]] )`

This method is similar to `parsefile` except that it wraps the parsing in an `eval` block. It returns the twig on success and 0 on failure (the twig object also contains the parsed twig) . `$@` contains the error message on failure

Note that the parsing still stops as soon as an error is detected, there is no way to keep going after an error.

`safe_parseurl ($url $optional_user_agent)`

Same as `parseurl` except that it wraps the parsing in an `eval` block. It returns the twig on success and 0 on failure (the twig object also contains the parsed twig) . `$@` contains the error message on failure

`parse_html ($string_or_fh)`

parse an HTML string or file handle (by converting it to XML using `HTML::TreeBuilder`, which needs to be available).

This works nicely, but some information gets lost in the process: newlines are removed, and (at least on the version I use), comments get an extra CDATA section inside ( `<!-- foo -->` becomes `<!--> <![CDATA[ foo ]]> -->`

**parsefile\_html (\$file)**

parse an HTML file (by converting it to XML using `HTML::TreeBuilder`, which needs to be available, or `HTML::Tidy` if the `use_tidy` option was used). The file is loaded completely in memory and converted to XML before being parsed.

this method is to be used with caution though, as it doesn't know about the file encoding, it is usually better to use `parse_html`, which gives you a chance to open the file with the proper encoding layer.

**parseurl\_html (\$url \$optional\_user\_agent)**

parse an URL as html the same way `parse_html` does

**safe\_parseurl\_html (\$url \$optional\_user\_agent)**

Same as `parseurl_html` except that it wraps the parsing in an `eval` block. It returns the twig on success and 0 on failure (the twig object also contains the parsed twig) . `$@` contains the error message on failure

**safe\_parsefile\_html (\$file \$optional\_user\_agent)**

Same as `parsefile_html` except that it wraps the parsing in an `eval` block. It returns the twig on success and 0 on failure (the twig object also contains the parsed twig) . `$@` contains the error message on failure

**safe\_parse\_html (\$string\_or\_fh)**

Same as `parse_html` except that it wraps the parsing in an `eval` block. It returns the twig on success and 0 on failure (the twig object also contains the parsed twig) . `$@` contains the error message on failure

**xparse (\$thing\_to\_parse)**

parse the `$thing_to_parse`, whether it is a filehandle, a string, an HTML file, an HTML URL, an URL or a file.

Note that this is mostly a convenience method for one-off scripts. For example files that end in `'.htm'` or `'.html'` are parsed first as XML, and if this fails as HTML. This is certainly not the most efficient way to do this in general.

**nparsed (\$optional\_twig\_options, \$thing\_to\_parse)**

create a twig with the `$optional_options`, and parse the `$thing_to_parse`, whether it is a filehandle, a string, an HTML file, an HTML URL, an URL or a file.

Examples:

```
XML::Twig->nparsed( "file.xml");
XML::Twig->nparsed( error_context => 1, "file://file.xml");
```

**nparsed\_pp (\$optional\_twig\_options, \$thing\_to\_parse)**

same as `nparsed` but also sets the `pretty_print` option to `indented`.

**nparsed\_e (\$optional\_twig\_options, \$thing\_to\_parse)**

same as `nparsed` but also sets the `error_context` option to 1.

**nparsed\_ppe (\$optional\_twig\_options, \$thing\_to\_parse)**

same as `nparsed` but also sets the `pretty_print` option to `indented` and the `error_context` option to 1.

**parser**

This method returns the `expat` object (actually the `XML::Parser::Expat` object) used during parsing. It is useful for example to call `XML::Parser::Expat` methods on it. To get the line of a tag for example use `$t->parser->current_line`.

**setTwigHandlers (\$handlers)**

Set the `twig_handlers`. `$handlers` is a reference to a hash similar to the one in the `twig_handlers` option of `new`. All previous handlers are unset. The method returns the reference to the previous handlers.

`setTwigHandler ($exp $handler)`

Set a single `twig_handler` for elements matching `$exp`. `$handler` is a reference to a subroutine. If the handler was previously set then the reference to the previous handler is returned.

`setStartTagHandlers ($handlers)`

Set the `start_tag` handlers. `$handlers` is a reference to a hash similar to the one in the `start_tag_handlers` option of `new`. All previous handlers are unset. The method returns the reference to the previous handlers.

`setStartTagHandler ($exp $handler)`

Set a single `start_tag` handlers for elements matching `$exp`. `$handler` is a reference to a subroutine. If the handler was previously set then the reference to the previous handler is returned.

`setEndTagHandlers ($handlers)`

Set the `end_tag` handlers. `$handlers` is a reference to a hash similar to the one in the `end_tag_handlers` option of `new`. All previous handlers are unset. The method returns the reference to the previous handlers.

`setEndTagHandler ($exp $handler)`

Set a single `end_tag` handlers for elements matching `$exp`. `$handler` is a reference to a subroutine. If the handler was previously set then the reference to the previous handler is returned.

`setTwigRoots ($handlers)`

Same as using the `twig_roots` option when creating the twig

`setCharHandler ($exp $handler)`

Set a `char_handler`

`setIgnoreEltsHandler ($exp)`

Set a `ignore_elt` handler (elements that match `$exp` will be ignored)

`setIgnoreEltsHandlers ($exp)`

Set all `ignore_elt` handlers (previous handlers are replaced)

`dtd` Return the dtd (an XML::`Twig::DTD` object) of a twig

`xmldecl`

Return the XML declaration for the document, or a default one if it doesn't have one

`doctype`

Return the doctype for the document

`doctype_name`

returns the doctype of the document from the doctype declaration

`system_id`

returns the system value of the DTD of the document from the doctype declaration

`public_id`

returns the public doctype of the document from the doctype declaration

`internal_subset`

returns the internal subset of the DTD

`dtd_text`

Return the DTD text

`dtd_print`

Print the DTD

`model ($tag)`

Return the model (in the DTD) for the element `$tag`

`root`

Return the root element of a twig

`set_root ($elt)`  
Set the root of a twig

`first_elt ($optional_condition)`  
Return the first element matching `$optional_condition` of a twig, if no condition is given then the root is returned

`last_elt ($optional_condition)`  
Return the last element matching `$optional_condition` of a twig, if no condition is given then the last element of the twig is returned

`elt_id ($id)`  
Return the element whose `id` attribute is `$id`

`getEltById`  
Same as `elt_id`

`index ($index_name, $optional_index)`  
If the `$optional_index` argument is present, return the corresponding element in the index (created using the `index` option for `XML::Twig-new`)  
  
If the argument is not present, return an arrayref to the index

`normalize`  
merge together all consecutive `pcdata` elements in the document (if for example you have turned some elements into `pcdata` using `erase`, this will give you a “clean” document in which there all text elements are as long as possible).

`encoding`  
This method returns the encoding of the XML document, as defined by the `encoding` attribute in the XML declaration (ie it is `undef` if the attribute is not defined)

`set_encoding`  
This method sets the value of the `encoding` attribute in the XML declaration. Note that if the document did not have a declaration it is generated (with an XML version of 1.0)

`xml_version`  
This method returns the XML version, as defined by the `version` attribute in the XML declaration (ie it is `undef` if the attribute is not defined)

`set_xml_version`  
This method sets the value of the `version` attribute in the XML declaration. If the declaration did not exist it is created.

`standalone`  
This method returns the value of the `standalone` declaration for the document

`set_standalone`  
This method sets the value of the `standalone` attribute in the XML declaration. Note that if the document did not have a declaration it is generated (with an XML version of 1.0)

`set_output_encoding`  
Set the encoding “attribute” in the XML declaration

`set_doctype ($name, $system, $public, $internal)`  
Set the doctype of the element. If an argument is `undef` (or not present) then its former value is retained, if a false (” or 0) value is passed then the former value is deleted;

`entity_list`  
Return the entity list of a twig

`entity_names`  
Return the list of all defined entities

`entity ($entity_name)`

Return the entity

`change_gi ($old_gi, $new_gi)`

Performs a (very fast) global change. All elements `$old_gi` are now `$new_gi`. This is a bit dangerous though and should be avoided if < possible, as the new tag might be ignored in subsequent processing.

See BUGS

`flush ($optional_filehandle, %options)`

Flushes a twig up to (and including) the current element, then deletes all unnecessary elements from the tree that's kept in memory. `flush` keeps track of which elements need to be open/closed, so if you flush from handlers you don't have to worry about anything. Just keep flushing the twig every time you're done with a sub-tree and it will come out well-formed. After the whole parsing don't forget to flush one more time to print the end of the document. The doctype and entity declarations are also printed.

`flush` take an optional filehandle as an argument.

If you use `flush` at any point during parsing, the document will be flushed one last time at the end of the parsing, to the proper filehandle.

options: use the `update_DTD` option if you have updated the (internal) DTD and/or the entity list and you want the updated DTD to be output

The `pretty_print` option sets the pretty printing of the document.

```
Example: $t->flush( Update_DTD => 1);
          $t->flush( $filehandle, pretty_print => 'indented');
          $t->flush( \*FILE);
```

`flush_up_to ($elt, $optional_filehandle, %options)`

Flushes up to the `$elt` element. This allows you to keep part of the tree in memory when you flush.

options: see `flush`.

`purge`

Does the same as a `flush` except it does not print the twig. It just deletes all elements that have been completely parsed so far.

`purge_up_to ($elt)`

Purges up to the `$elt` element. This allows you to keep part of the tree in memory when you purge.

`print ($optional_filehandle, %options)`

Prints the whole document associated with the twig. To be used only AFTER the parse.

options: see `flush`.

`print_to_file ($filename, %options)`

Prints the whole document associated with the twig to file `$filename`. To be used only AFTER the parse.

options: see `flush`.

`safe_print_to_file ($filename, %options)`

Prints the whole document associated with the twig to file `$filename`. This variant, which probably only works on \*nix prints to a temp file, then move the temp file to overwrite the original file.

This is a bit safer when 2 processes an potentially write the same file: only the last one will succeed, but the file won't be corrupted. I often use this for cron jobs, so testing the code doesn't interfere with the cron job running at the same time.

options: see `flush`.

**sprint**

Return the text of the whole document associated with the twig. To be used only AFTER the parse.

options: see `flush`.

**trim**

Trim the document: gets rid of initial and trailing spaces, and replaces multiple spaces by a single one.

**toSAX1 (\$handler)**

Send SAX events for the twig to the SAX1 handler `$handler`

**toSAX2 (\$handler)**

Send SAX events for the twig to the SAX2 handler `$handler`

**flush\_toSAX1 (\$handler)**

Same as `flush`, except that SAX events are sent to the SAX1 handler `$handler` instead of the twig being printed

**flush\_toSAX2 (\$handler)**

Same as `flush`, except that SAX events are sent to the SAX2 handler `$handler` instead of the twig being printed

**ignore**

This method should be called during parsing, usually in `start_tag_handlers`. It causes the element to be skipped during the parsing: the twig is not built for this element, it will not be accessible during parsing or after it. The element will not take up any memory and parsing will be faster.

Note that this method can also be called on an element. If the element is a parent of the current element then this element will be ignored (the twig will not be built any more for it and what has already been built will be deleted).

**set\_pretty\_print (\$style)**

Set the pretty print method, amongst `'none'` (default), `'nsgmls'`, `'nice'`, `'indented'`, `indented_c`, `'wrapped'`, `'record'` and `'record_c'`

**WARNING:** the pretty print style is a **GLOBAL** variable, so once set it's applied to **ALL** print's (and sprint's). Same goes if you use `XML::Twig` with `mod_perl`. This should not be a problem as the XML that's generated is valid anyway, and XML processors (as well as HTML processors, including browsers) should not care. Let me know if this is a big problem, but at the moment the performance/cleanliness trade-off clearly favors the global approach.

**set\_empty\_tag\_style (\$style)**

Set the empty tag display style (`'normal'`, `'html'` or `'expand'`). As with `set_pretty_print` this sets a global flag.

`normal` outputs an empty tag `'<tag/>'`, `html` adds a space `'<tag />'` for elements that can be empty in XHTML and `expand` outputs `'<tag></tag>'`

**set\_remove\_cdata (\$flag)**

set (or unset) the flag that forces the twig to output CDATA sections as regular (escaped) PCDATA

**print\_prolog (\$optional\_filehandle, %options)**

Prints the prolog (XML declaration + DTD + entity declarations) of a document.

options: see `flush`.

**prolog (\$optional\_filehandle, %options)**

Return the prolog (XML declaration + DTD + entity declarations) of a document.

options: see `flush`.

**finish**

Call Expat `finish` method. Unsets all handlers (including internal ones that set context), but expat continues parsing to the end of the document or until it finds an error. It should finish up a lot faster than with the handlers set.

**finish\_print**

Stops twig processing, flush the twig and proceed to finish printing the document as fast as possible. Use this method when modifying a document and the modification is done.

**finish\_now**

Stops twig processing, does not finish parsing the document (which could actually be not well-formed after the point where `finish_now` is called). Execution resumes after the `Lparse>` or `parsefile` call. The content of the twig is what has been parsed so far (all open elements at the time `finish_now` is called are considered closed).

**set\_expand\_external\_entities**

Same as using the `expand_external_ents` option when creating the twig

**set\_input\_filter**

Same as using the `input_filter` option when creating the twig

**set\_keep\_atts\_order**

Same as using the `keep_atts_order` option when creating the twig

**set\_keep\_encoding**

Same as using the `keep_encoding` option when creating the twig

**escape\_gt**

usually XML::Twig does not escape `>` in its output. Using this option makes it replace `>` by `&gt;`;

**do\_not\_escape\_gt**

reverts XML::Twig behavior to its default of not escaping `>` in its output.

**set\_output\_filter**

Same as using the `output_filter` option when creating the twig

**set\_output\_text\_filter**

Same as using the `output_text_filter` option when creating the twig

**add\_stylesheet (\$type, @options)**

Adds an external stylesheet to an XML document.

Supported types and options:

`xsl` option: the url of the stylesheet

Example:

```
$t->add_stylesheet( xsl => "xsl_style.xsl");
```

will generate the following PI at the beginning of the document:

```
<?xml-stylesheet type="text/xsl" href="xsl_style.xsl"?>
```

`css` option: the url of the stylesheet

**active\_twig**

a class method that returns the last processed twig, so you don't necessarily need the object to call methods on it.

**Methods inherited from XML::Parser::Expat**

A twig inherits all the relevant methods from XML::Parser::Expat. These methods can only be used during the parsing phase (they will generate a fatal error otherwise).

Inherited methods are:

**depth**

Returns the size of the context list.

**in\_element**

Returns true if NAME is equal to the name of the innermost currently opened element. If namespace processing is being used and you want to check against a name that may be in a



namespace, then use the `generate_ns_name` method to create the NAME argument.

`within_element`

Returns the number of times the given name appears in the context list. If namespace processing is being used and you want to check against a name that may be in a namespace, then use the `generate_ns_name` method to create the NAME argument.

`context`

Returns a list of element names that represent open elements, with the last one being the innermost. Inside start and end tag handlers, this will be the tag of the parent element.

`current_line`

Returns the line number of the current position of the parse.

`current_column`

Returns the column number of the current position of the parse.

`current_byte`

Returns the current position of the parse.

`position_in_context`

Returns a string that shows the current parse position. LINES should be an integer  $\geq 0$  that represents the number of lines on either side of the current parse line to place into the returned string.

`base ([NEWBASE])`

Returns the current value of the base for resolving relative URIs. If NEWBASE is supplied, changes the base to that value.

`current_element`

Returns the name of the innermost currently opened element. Inside start or end handlers, returns the parent of the element associated with those tags.

`element_index`

Returns an integer that is the depth-first visit order of the current element. This will be zero outside of the root element. For example, this will return 1 when called from the start handler for the root element start tag.

`recognized_string`

Returns the string from the document that was recognized in order to call the current handler. For instance, when called from a start handler, it will give us the start-tag string. The string is encoded in UTF-8. This method doesn't return a meaningful string inside declaration handlers.

`original_string`

Returns the verbatim string from the document that was recognized in order to call the current handler. The string is in the original document encoding. This method doesn't return a meaningful string inside declaration handlers.

`xpcroak`

Concatenate onto the given message the current line number within the XML document plus the message implied by `ErrorContext`. Then croak with the formed message.

`xpcarp`

Concatenate onto the given message the current line number within the XML document plus the message implied by `ErrorContext`. Then carp with the formed message.

`xml_escape(TEXT [, CHAR [, CHAR ...]])`

Returns TEXT with markup characters turned into character entities. Any additional characters provided as arguments are also turned into character references where found in TEXT.

(this method is broken on some versions of `expat/XML::Parser`)

`path ( $optional_tag )`

Return the element context in a form similar to XPath's short form: `'/root/tag1/../tag'`

`get_xpath ( $optional_array_ref, $xpath, $optional_offset )`

Performs a `get_xpath` on the document root (see `<Elt["Elt"]>`)

If the `$optional_array_ref` argument is used the array must contain elements. The `$xpath` expression is applied to each element in turn and the result is union of all results. This way a first query can be refined in further steps.

`find_nodes ( $optional_array_ref, $xpath, $optional_offset )`

same as `get_xpath`

`findnodes ( $optional_array_ref, $xpath, $optional_offset )`

same as `get_xpath` (similar to the `XML::LibXML` method)

`findvalue ( $optional_array_ref, $xpath, $optional_offset )`

Return the join of all texts of the results of applying `get_xpath` to the node (similar to the `XML::LibXML` method)

`findvalues ( $optional_array_ref, $xpath, $optional_offset )`

Return an array of all texts of the results of applying `get_xpath` to the node

`subs_text ($regex, $replace)`

`subs_text` does text substitution on the whole document, similar to perl's `s///` operator.

`dispose`

Useful only if you don't have `Scalar::Util` or `WeakRef` installed.

Reclaims properly the memory used by an `XML::Twig` object. As the object has circular references it never goes out of scope, so if you want to parse lots of XML documents then the memory leak becomes a problem. Use `$twig->dispose` to clear this problem.

`att_accessors (list_of_attribute_names)`

A convenience method that creates l-valued accessors for attributes. So `$twig->create_accessors( 'foo' )` will create a `foo` method that can be called on elements:

```
$elt->foo;           # equivalent to $elt->{'att'}->{'foo'};
$elt->foo( 'bar' ); # equivalent to $elt->set_att( foo => 'bar' );
```

The methods are l-valued only under those perl's that support this feature (5.6 and above)

`create_accessors (list_of_attribute_names)`

Same as `att_accessors`

`elt_accessors (list_of_attribute_names)`

A convenience method that creates accessors for elements. So `$twig->create_accessors( 'foo' )` will create a `foo` method that can be called on elements:

```
$elt->foo;           # equivalent to $elt->first_child( 'foo' );
```

`field_accessors (list_of_attribute_names)`

A convenience method that creates accessors for element values (`field`). So `$twig->create_accessors( 'foo' )` will create a `foo` method that can be called on elements:

```
$elt->foo;           # equivalent to $elt->field( 'foo' );
```

`set_do_not_escape_amp_in_atts`

An evil method, that I only document because `Test::Pod::Coverage` complains otherwise, but really, you don't want to know about it.

**XML::Twig::Elt**

**new** (\$optional\_tag, \$optional\_atts, @optional\_content)

The tag is optional (but then you can't have a content), the \$optional\_atts argument is a reference to a hash of attributes, the content can be just a string or a list of strings and element. A content of '#EMPTY' creates an empty element;

```
Examples: my $elt= XML::Twig::Elt->new();
          my $elt= XML::Twig::Elt->new( para => { align => 'center' });
          my $elt= XML::Twig::Elt->new( para => { align => 'center' }, 'foo');
          my $elt= XML::Twig::Elt->new( br   => '#EMPTY');
          my $elt= XML::Twig::Elt->new( 'para');
          my $elt= XML::Twig::Elt->new( para => 'this is a para');
          my $elt= XML::Twig::Elt->new( para => $elt3, 'another para');
```

The strings are not parsed, the element is not attached to any twig.

**WARNING:** if you rely on ID's then you will have to set the id yourself. At this point the element does not belong to a twig yet, so the ID attribute is not known so it won't be stored in the ID list.

Note that #COMMENT, #PCDATA or #CDATA are valid tag names, that will create text elements.

To create an element foo containing a CDATA section:

```
my $foo= XML::Twig::Elt->new( '#CDATA' => "content of the CDATA section"
                              ->wrap_in( 'foo');
```

An attribute of '#CDATA', will create the content of the element as CDATA:

```
my $elt= XML::Twig::Elt->new( 'p' => { '#CDATA' => 1}, 'foo < bar');
```

creates an element

```
<p><![CDATA[foo < bar]]></p>
```

**parse** (\$string, %args)

Creates an element from an XML string. The string is actually parsed as a new twig, then the root of that twig is returned. The arguments in %args are passed to the twig. As always if the parse fails the parser will die, so use an eval if you want to trap syntax errors.

As obviously the element does not exist beforehand this method has to be called on the class:

```
my $elt= parse XML::Twig::Elt( "<a> string to parse, with <sub/>
                                <elements>, actually tons of </elements>
                                h</a>");
```

**set\_inner\_xml** (\$string)

Sets the content of the element to be the tree created from the string

**set\_inner\_html** (\$string)

Sets the content of the element, after parsing the string with an HTML parser (HTML::Parser)

**set\_outer\_xml** (\$string)

Replaces the element with the tree created from the string

**print** (\$optional\_filehandle, \$optional\_pretty\_print\_style)

Prints an entire element, including the tags, optionally to a \$optional\_filehandle, optionally with a \$pretty\_print\_style.

The print outputs XML data so base entities are escaped.

**print\_to\_file** (\$filename, %options)

Prints the element to file \$filename.

options: see flush. =item sprint (\$elt, \$optional\_no\_enclosing\_tag)

Return the xml string for an entire element, including the tags. If the optional second argument is true

then only the string inside the element is returned (the start and end tag for `$elt` are not). The text is XML-escaped: base entities (`&` and `<` in text, `&` `<` and `"` in attribute values) are turned into entities.

`gi` Return the `gi` of the element (the `gi` is the `generic identifier` the tag name in SGML parlance).

`tag` and `name` are synonyms of `gi`.

`tag` Same as `gi`

`name`

Same as `tag`

`set_gi` (`$tag`)

Set the `gi` (`tag`) of an element

`set_tag` (`$tag`)

Set the `tag` (`=tag`) of an element

`set_name` (`$name`)

Set the `name` (`=tag`) of an element

`root`

Return the root of the twig in which the element is contained.

`twig`

Return the twig containing the element.

`parent` (`$optional_condition`)

Return the parent of the element, or the first ancestor matching the `$optional_condition`

`first_child` (`$optional_condition`)

Return the first child of the element, or the first child matching the `$optional_condition`

`has_child` (`$optional_condition`)

Return the first child of the element, or the first child matching the `$optional_condition` (same as `first_child`)

`has_children` (`$optional_condition`)

Return the first child of the element, or the first child matching the `$optional_condition` (same as `first_child`)

`first_child_text` (`$optional_condition`)

Return the text of the first child of the element, or the first child matching the `$optional_condition`. If there is no `first_child` then returns `''`. This avoids getting the child, checking for its existence then getting the text for trivial cases.

Similar methods are available for the other navigation methods:

`last_child_text`

`prev_sibling_text`

`next_sibling_text`

`prev_elt_text`

`next_elt_text`

`child_text`

`parent_text`

All this methods also exist in “trimmed” variant:

`first_child_trimmed_text`

`last_child_trimmed_text`

`prev_sibling_trimmed_text`

`next_sibling_trimmed_text`

`prev_elt_trimmed_text`

`next_elt_trimmed_text`  
`child_trimmed_text`  
`parent_trimmed_text`  
`field ($condition)`  
 Same method as `first_child_text` with a different name  
`fields ($condition_list)`  
 Return the list of field (text of first child matching the conditions), missing fields are returned as the empty string.  
 Same method as `first_child_text` with a different name  
`trimmed_field ($optional_condition)`  
 Same method as `first_child_trimmed_text` with a different name  
`set_field ($condition, $optional_atts, @list_of_elt_and_strings)`  
 Set the content of the first child of the element that matches `$condition`, the rest of the arguments is the same as for `set_content`  
 If no child matches `$condition_and_` if `$condition` is a valid XML element name, then a new element by that name is created and inserted as the last child.  
`first_child_matches ($optional_condition)`  
 Return the element if the first child of the element (if it exists) passes the `$optional_condition` `undef` otherwise  

```
if( $elt->first_child_matches( 'title' ) ) ...
```

  
 is equivalent to  

```
if( $elt->{first_child} && $elt->{first_child}->passes( 'title' ) )
```

  
`first_child_is` is another name for this method  
 Similar methods are available for the other navigation methods:  
`last_child_matches`  
`prev_sibling_matches`  
`next_sibling_matches`  
`prev_elt_matches`  
`next_elt_matches`  
`child_matches`  
`parent_matches`  
`is_first_child ($optional_condition)`  
 returns true (the element) if the element is the first child of its parent (optionally that satisfies the `$optional_condition`)  
`is_last_child ($optional_condition)`  
 returns true (the element) if the element is the last child of its parent (optionally that satisfies the `$optional_condition`)  
`prev_sibling ($optional_condition)`  
 Return the previous sibling of the element, or the previous sibling matching `$optional_condition`  
`next_sibling ($optional_condition)`  
 Return the next sibling of the element, or the first one matching `$optional_condition`.  
`next_elt ($optional_elt, $optional_condition)`  
 Return the next elt (optionally matching `$optional_condition`) of the element. This is defined as the next element which opens after the current element opens. Which usually means the first child of the element. Counter-intuitive as it might look this allows you to loop through the whole document by starting from the root.

The `$optional_elt` is the root of a subtree. When the `next_elt` is out of the subtree then the method returns undef. You can then walk a sub-tree with:

```
my $elt= $subtree_root;
while( $elt= $elt->next_elt( $subtree_root))
{ # insert processing code here
}
```

`prev_elt ($optional_condition)`

Return the previous elt (optionally matching `$optional_condition`) of the element. This is the first element which opens before the current one. It is usually either the last descendant of the previous sibling or simply the parent

`next_n_elt ($offset, $optional_condition)`

Return the `$offset`-th element that matches the `$optional_condition`

`following_elt`

Return the following element (as per the XPath following axis)

`preceding_elt`

Return the preceding element (as per the XPath preceding axis)

`following_elts`

Return the list of following elements (as per the XPath following axis)

`preceding_elts`

Return the list of preceding elements (as per the XPath preceding axis)

`children ($optional_condition)`

Return the list of children (optionally which matches `$optional_condition`) of the element. The list is in document order.

`children_count ($optional_condition)`

Return the number of children of the element (optionally which matches `$optional_condition`)

`children_text ($optional_condition)`

In array context, returns an array containing the text of children of the element (optionally which matches `$optional_condition`)

In scalar context, returns the concatenation of the text of children of the element

`children_trimmed_text ($optional_condition)`

In array context, returns an array containing the trimmed text of children of the element (optionally which matches `$optional_condition`)

In scalar context, returns the concatenation of the trimmed text of children of the element

`children_copy ($optional_condition)`

Return a list of elements that are copies of the children of the element, optionally which matches `$optional_condition`

`descendants ($optional_condition)`

Return the list of all descendants (optionally which matches `$optional_condition`) of the element. This is the equivalent of the `getElementsByTagName` of the DOM (by the way, if you are really a DOM addict, you can use `getElementsByTagName` instead)

`getElementsByTagName ($optional_condition)`

Same as `descendants`

`find_by_tag_name ($optional_condition)`

Same as `descendants`

`descendants_or_self ($optional_condition)`

Same as `descendants` except that the element itself is included in the list if it matches the `$optional_condition`

`first_descendant ($optional_condition)`

Return the first descendant of the element that matches the condition

`last_descendant ($optional_condition)`

Return the last descendant of the element that matches the condition

`ancestors ($optional_condition)`

Return the list of ancestors (optionally matching `$optional_condition`) of the element. The list is ordered from the innermost ancestor to the outermost one

NOTE: the element itself is not part of the list, in order to include it you will have to use `ancestors_or_self`

`ancestors_or_self ($optional_condition)`

Return the list of ancestors (optionally matching `$optional_condition`) of the element, including the element (if it matches the condition). The list is ordered from the innermost ancestor to the outermost one

`passes ($condition)`

Return the element if it passes the `$condition`

`att ($att)`

Return the value of attribute `$att` or `undef`

`latt ($att)`

Return the value of attribute `$att` or `undef`

this method is an lvalue, so you can do `$elt->latt( 'foo')= 'bar'` or `$elt->latt( 'foo')++`;

`set_att ($att, $att_value)`

Set the attribute of the element to the given value

You can actually set several attributes this way:

```
$elt->set_att( att1 => "val1", att2 => "val2");
```

`del_att ($att)`

Delete the attribute for the element

You can actually delete several attributes at once:

```
$elt->del_att( 'att1', 'att2', 'att3');
```

`att_exists ($att)`

Returns true if the attribute `$att` exists for the element, false otherwise

`cut` Cut the element from the tree. The element still exists, it can be copied or pasted somewhere else, it is just not attached to the tree anymore.

Note that the “old” links to the parent, previous and next siblings can still be accessed using the `former_*` methods

`former_next_sibling`

Returns the former next sibling of a cut node (or `undef` if the node has not been cut)

This makes it easier to write loops where you cut elements:

```
my $child= $parent->first_child( 'achild');
while( $child->{'att'}->{'cut'})
{ $child->cut; $child= ($child->{former} && $child->{former}->{next_sibling})
```

`former_prev_sibling`

Returns the former previous sibling of a cut node (or `undef` if the node has not been cut)

`former_parent`

Returns the former parent of a cut node (or undef if the node has not been cut)

`cut_children ($optional_condition)`

Cut all the children of the element (or all of those which satisfy the `$optional_condition`).

Return the list of children

`cut_descendants ($optional_condition)`

Cut all the descendants of the element (or all of those which satisfy the `$optional_condition`).

Return the list of descendants

`copy ($elt)`

Return a copy of the element. The copy is a “deep” copy: all sub-elements of the element are duplicated.

`paste ($optional_position, $ref)`

Paste a (previously cut or newly generated) element. Die if the element already belongs to a tree.

Note that the calling element is pasted:

```
$child->paste( first_child => $existing_parent);
$new_sibling->paste( after => $this_sibling_is_already_in_the_tree);
```

or

```
my $new_elt= XML::Twig::Elt->new( tag => $content);
$new_elt->paste( $position => $existing_elt);
```

Example:

```
my $t= XML::Twig->new->parse( 'doc.xml')
my $toc= $t->root->new( 'toc');
$toc->paste( $t->root); # $toc is pasted as first child of the root
foreach my $title ($t->findnodes( '/doc/section/title'))
{ my $title_toc= $title->copy;
  # paste $title_toc as the last child of toc
  $title_toc->paste( last_child => $toc)
}
```

Position options:

`first_child` (default)

The element is pasted as the first child of `$ref`

`last_child`

The element is pasted as the last child of `$ref`

`before`

The element is pasted before `$ref`, as its previous sibling.

`after`

The element is pasted after `$ref`, as its next sibling.

`within`

In this case an extra argument, `$offset`, should be supplied. The element will be pasted in the reference element (or in its first text child) at the given offset. To achieve this the reference element will be split at the offset.

Note that you can call directly the underlying method:

`paste_before`



`paste_after`  
`paste_first_child`  
`paste_last_child`  
`paste_within`  
`move` (\$optional\_position, \$ref)  
 Move an element in the tree. This is just a cut then a paste. The syntax is the same as `paste`.  
`replace` (\$ref)  
 Replaces an element in the tree. Sometimes it is just not possible to cut an element then paste another in its place, so `replace` comes in handy. The calling element replaces \$ref.  
`replace_with` (@elts)  
 Replaces the calling element with one or more elements  
`delete`  
 Cut the element and frees the memory.  
`prefix` (\$text, \$optional\_option)  
 Add a prefix to an element. If the element is a PCDATA element the text is added to the pcddata, if the elements first child is a PCDATA then the text is added to it's pcddata, otherwise a new PCDATA element is created and pasted as the first child of the element.  
 If the option is `asis` then the prefix is added asis: it is created in a separate PCDATA element with an `asis` property. You can then write:  

```
$elt1->prefix( '<b>', 'asis');
```

 to create a `<b>` in the output of `print`.  
`suffix` (\$text, \$optional\_option)  
 Add a suffix to an element. If the element is a PCDATA element the text is added to the pcddata, if the elements last child is a PCDATA then the text is added to it's pcddata, otherwise a new PCDATA element is created and pasted as the last child of the element.  
 If the option is `asis` then the suffix is added asis: it is created in a separate PCDATA element with an `asis` property. You can then write:  

```
$elt2->suffix( '</b>', 'asis');
```

`trim`  
 Trim the element in-place: spaces at the beginning and at the end of the element are discarded and multiple spaces within the element (or its descendants) are replaced by a single space.  
 Note that in some cases you can still end up with multiple spaces, if they are split between several elements:  

```
<doc> text <b> hah! </b> yep</doc>
```

 gets trimmed to  

```
<doc>text <b> hah! </b> yep</doc>
```

 This is somewhere in between a bug and a feature.  
`normalize`  
 merge together all consecutive pcddata elements in the element (if for example you have turned some elements into pcddata using `erase`, this will give you a “clean” element in which there all text fragments are as long as possible).  
`simplify` (%options)  
 Return a data structure suspiciously similar to XML::Simple's. Options are identical to XMLin options, see XML::Simple doc for more details (or use DATA::dumper or YAML to dump the data structure)  
**Note:** there is no magic here, if you write `$twig->parsefile( $file )->simplify();`

then it will load the entire document in memory. I am afraid you will have to put some work into it to get just the bits you want and discard the rest. Look at the synopsis or the XML::Twig 101 section at the top of the docs for more information.

content\_key  
forcearray  
keyattr  
noattr  
normalize\_space  
    aka normalise\_space

variables (%var\_hash)  
    %var\_hash is a hash { name => value }

This option allows variables in the XML to be expanded when the file is read. (there is no facility for putting the variable names back if you regenerate XML using XMLout).

A 'variable' is any text of the form \${name} (or \$name) which occurs in an attribute value or in the text content of an element. If 'name' matches a key in the supplied hashref, \${name} will be replaced with the corresponding value from the hashref. If no matching key is found, the variable will not be replaced.

var\_att (\$attribute\_name)

This option gives the name of an attribute that will be used to create variables in the XML:

```
<dirs>
  <dir name="prefix">/usr/local</dir>
  <dir name="exec_prefix">$prefix/bin</dir>
</dirs>
```

use var => 'name' to get \$prefix replaced by /usr/local in the generated data structure

By default variables are captured by the following regexp: /\$(\w+)/

var\_regexp (regexp)

This option changes the regexp used to capture variables. The variable name should be in \$1

group\_tags { grouping tag => grouped tag, grouping tag 2 => grouped tag 2... }

Option used to simplify the structure: elements listed will not be used. Their children will be, they will be considered children of the element parent.

If the element is:

```
<config host="laptop.xmltwig.org">
  <server>localhost</server>
  <dirs>
    <dir name="base">/home/mrodrigu/standards</dir>
    <dir name="tools">$base/tools</dir>
  </dirs>
  <templates>
    <template name="std_def">std_def.templ</template>
    <template name="dummy">dummy</template>
  </templates>
</config>
```

Then calling simplify with group\_tags => { dirs => 'dir', templates => 'template' } makes the data structure be exactly as if the start and end tags for dirs and templates were not there.

A YAML dump of the structure

```

base: '/home/mrodrigu/standards'
host: laptop.xmltwig.org
server: localhost
template:
  - std_def.templ
  - dummy.templ
tools: '$base/tools'

```

`split_at` (\$offset)

Split a text (PCDATA or CDATA) element in 2 at \$offset, the original element now holds the first part of the string and a new element holds the right part. The new element is returned

If the element is not a text element then the first text child of the element is split

`split` (\$optional\_regexp, \$tag1, \$atts1, \$tag2, \$atts2...)

Split the text descendants of an element in place, the text is split using the \$regexp, if the regexp includes () then the matched separators will be wrapped in elements. \$1 is wrapped in \$tag1, with attributes \$atts1 if \$atts1 is given (as a hashref), \$2 is wrapped in \$tag2...

if \$elt is <p>tati tata <b>tutu tati titi</b> tata tati tata</p>

```
$elt->split( qr/(ta)ti/, 'foo', {type => 'toto'} )
```

will change \$elt to

```

<p><foo type="toto">ta</foo> tata <b>tutu <foo type="toto">ta</foo>
titi</b> tata <foo type="toto">ta</foo> tata</p>

```

The regexp can be passed either as a string or as qr// (perl 5.005 and later), it defaults to \s+ just as the `split` built-in (but this would be quite a useless behaviour without the \$optional\_tag parameter)

\$optional\_tag defaults to PCDATA or CDATA, depending on the initial element type

The list of descendants is returned (including un-touched original elements and newly created ones)

`mark` (\$regexp, \$optional\_tag, \$optional\_attribute\_ref)

This method behaves exactly as `split`, except only the newly created elements are returned

`wrap_children` (\$regexp\_string, \$tag, \$optional\_attribute\_hashref)

Wrap the children of the element that match the regexp in an element \$tag. If \$optional\_attribute\_hashref is passed then the new element will have these attributes.

The \$regexp\_string includes tags, within pointy brackets, as in <title><para>+ and the usual Perl modifiers (+\*?...). Tags can be further qualified with attributes: <para type="warning" classif="cosmic\_secret">+. The values for attributes should be xml-escaped: <candy type="M&Ms">\* (<, & > and " should be escaped).

Note that elements might get extra id attributes in the process. See `add_id`. Use `strip_att` to remove unwanted id's.

Here is an example:

If the element \$elt has the following content:

```

<elt>
  <p>para 1</p>
  <l_l1_1>list 1 item 1 para 1</l_l1_1>
    <l_l1>list 1 item 1 para 2</l_l1>
  <l_l1_n>list 1 item 2 para 1 (only para)</l_l1_n>
  <l_l1_n>list 1 item 3 para 1</l_l1_n>
    <l_l1>list 1 item 3 para 2</l_l1>
    <l_l1>list 1 item 3 para 3</l_l1>
  <l_l1_1>list 2 item 1 para 1</l_l1_1>
    <l_l1>list 2 item 1 para 2</l_l1>
  <l_l1_n>list 2 item 2 para 1 (only para)</l_l1_n>
  <l_l1_n>list 2 item 3 para 1</l_l1_n>
    <l_l1>list 2 item 3 para 2</l_l1>
    <l_l1>list 2 item 3 para 3</l_l1>
</elt>

```

Then the code

```

$elt->wrap_children( q{<l_l1_1><l_l1>*} , li => { type => "ul1" });
$elt->wrap_children( q{<l_l1_n><l_l1>*} , li => { type => "ul" });

$elt->wrap_children( q{<li type="ul1"><li type="ul">+}, "ul");
$elt->strip_att( 'id');
$elt->strip_att( 'type');
$elt->print;

```

will output:

```

<elt>
  <p>para 1</p>
  <ul>
    <li>
      <l_l1_1>list 1 item 1 para 1</l_l1_1>
      <l_l1>list 1 item 1 para 2</l_l1>
    </li>
    <li>
      <l_l1_n>list 1 item 2 para 1 (only para)</l_l1_n>
    </li>
    <li>
      <l_l1_n>list 1 item 3 para 1</l_l1_n>
      <l_l1>list 1 item 3 para 2</l_l1>
      <l_l1>list 1 item 3 para 3</l_l1>
    </li>
  </ul>
  <ul>
    <li>
      <l_l1_1>list 2 item 1 para 1</l_l1_1>
      <l_l1>list 2 item 1 para 2</l_l1>
    </li>
    <li>
      <l_l1_n>list 2 item 2 para 1 (only para)</l_l1_n>
    </li>
    <li>
      <l_l1_n>list 2 item 3 para 1</l_l1_n>
      <l_l1>list 2 item 3 para 2</l_l1>
      <l_l1>list 2 item 3 para 3</l_l1>
    </li>
  </ul>

```

```

        </li>
    </ul>
</elt>

```

`subs_text($regexp, $replace)`

`subs_text` does text substitution, similar to perl's `s///` operator.

`$regexp` must be a perl regexp, created with the `qr` operator.

`$replace` can include `$1`, `$2...` from the `$regexp`. It can also be used to create element and entities, by using `&elt( tag => { att => val }, text)` (similar syntax as `new`) and `&ent( name)`.

Here is a rather complex example:

```

$elt->subs_text( qr{(?<!do not )link to (http://([^\s,]*) )},
    'see &elt( a =>{ href => $1 }, $2) '
);

```

This will replace text like `link to http://www.xmltwig.org` by `see <a href="http://www.xmltwig.org">http://www.xmltwig.org</a>`, but not `do not link to...`

Generating entities (here replacing spaces with `&nbsp;`):

```

$elt->subs_text( qr{ }, '&ent( "&nbsp;" )' );

```

or, using a variable:

```

my $ent="&nbsp;";
$elt->subs_text( qr{ }, "&ent( '$ent' )" );

```

Note that the substitution is always global, as in using the `g` modifier in a perl substitution, and that it is performed on all text descendants of the element.

**Bug:** in the `$regexp`, you can only use `\1`, `\2...` if the replacement expression does not include elements or attributes. eg

```

$t->subs_text( qr/((t[aiou])\2)/, '$2' ); # ok, replaces toto, ta
$t->subs_text( qr/((t[aiou])\2)/, '&elt(p => $1)' ); # NOK, does not find to

```

`add_id($optional_coderef)`

Add an id to the element.

The id is an attribute, `id` by default, see the `id` option for XML::Twig `new` to change it. Use an id starting with `#` to get an id that's not output by `print`, `flush` or `sprint`, yet that allows you to use the `elt_id` method to get the element easily.

If the element already has an id, no new id is generated.

By default the method create an id of the form `twig_id_<nnnn>`, where `<nnnn>` is a number, incremented each time the method is called successfully.

`set_id_seed($prefix)`

by default the id generated by `add_id` is `twig_id_<nnnn>`, `set_id_seed` changes the prefix to `$prefix` and resets the number to 1

`strip_att($att)`

Remove the attribute `$att` from all descendants of the element (including the element)

Return the element

`change_att_name($old_name, $new_name)`

Change the name of the attribute from `$old_name` to `$new_name`. If there is no attribute `$old_name` nothing happens.

`lc_attnames`

Lower cases the name all the attributes of the element.

`sort_children_on_value( %options)`

Sort the children of the element in place according to their text. All children are sorted.

Return the element, with its children sorted.

%options are

```
type   : numeric | alpha      (default: alpha)
order  : normal  | reverse    (default: normal)
```

Return the element, with its children sorted

`sort_children_on_att( $att, %options)`

Sort the children of the element in place according to attribute \$att. %options are the same as for `sort_children_on_value`

Return the element.

`sort_children_on_field( $tag, %options)`

Sort the children of the element in place, according to the field \$tag (the text of the first child of the child with this tag). %options are the same as for `sort_children_on_value`.

Return the element, with its children sorted

`sort_children( $get_key, %options)`

Sort the children of the element in place. The \$get\_key argument is a reference to a function that returns the sort key when passed an element.

For example:

```
$elt->sort_children( sub { $_[0]->{'att'}->{"nb"} + $_[0]->text },
                    type => 'numeric', order => 'reverse'
                  );
```

`field_to_att( $cond, $att)`

Turn the text of the first sub-element matched by \$cond into the value of attribute \$att of the element. If \$att is omitted then \$cond is used as the name of the attribute, which makes sense only if \$cond is a valid element (and attribute) name.

The sub-element is then cut.

`att_to_field( $att, $tag)`

Take the value of attribute \$att and create a sub-element \$tag as first child of the element. If \$tag is omitted then \$att is used as the name of the sub-element.

`get_xpath( $xpath, $optional_offset)`

Return a list of elements satisfying the \$xpath. \$xpath is an XPATH-like expression.

A subset of the XPATH abbreviated syntax is covered:

```
tag
tag[1] (or any other positive number)
tag[last()]
tag[@att] (the attribute exists for the element)
tag[@att="val"]
tag[@att=~ /regexp/]
tag[att1="val1" and att2="val2"]
tag[att1="val1" or att2="val2"]
tag[string()="toto"] (returns tag elements which text (as per the text method)
                    is toto)
tag[string()=~ /regexp/] (returns tag elements which text (as per the text
```

```

        method) matches regexp)
expressions can start with / (search starts at the document root)
expressions can start with . (search starts at the current element)
// can be used to get all descendants instead of just direct children
* matches any tag

```

So the following examples from the *XPath recommendation* <<http://www.w3.org/TR/xpath.html#path-abbrev>> work:

```

para selects the para element children of the context node
* selects all element children of the context node
para[1] selects the first para child of the context node
para[last()] selects the last para child of the context node
*/para selects all para grandchildren of the context node
/doc/chapter[5]/section[2] selects the second section of the fifth chapter
  of the doc
chapter//para selects the para element descendants of the chapter element
  children of the context node
//para selects all the para descendants of the document root and thus select
  all para elements in the same document as the context node
//olist/item selects all the item elements in the same document as the
  context node that have an olist parent
../para selects the para element descendants of the context node
.. selects the parent of the context node
para[@type="warning"] selects all para children of the context node that hav
  a type attribute with value warning
employee[@secretary and @assistant] selects all the employee children of the
  context node that have both a secretary attribute and an assistant
  attribute

```

The elements will be returned in the document order.

If \$optional\_offset is used then only one element will be returned, the one with the appropriate offset in the list, starting at 0

Quoting and interpolating variables can be a pain when the Perl syntax and the XPATH syntax collide, so use alternate quoting mechanisms like q or qq (I like q{ } and qq{ } myself).

Here are some more examples to get you started:

```

my $p1= "p1";
my $p2= "p2";
my @res= $t->get_xpath( qq{p[string( "$p1" ) or string( "$p2" )]});

my $a= "a1";
my @res= $t->get_xpath( qq{//*[att="$a"]});

my $val= "a1";
my $exp= qq{p[ \@att='$val']}; # you need to use \@ or you will get a warn
my @res= $t->get_xpath( $exp);

```

Note that the only supported regexps delimiters are / and that you must backslash all / in regexps AND in regular strings.

XML::Twig does not provide natively full XPATH support, but you can use XML::Twig::XPath to get findnodes to use XML::XPath as the XPath engine, with full coverage of the spec.

XML::Twig::XPath to get findnodes to use XML::XPath as the XPath engine, with full coverage of the spec.

`find_nodes`  
 same as `get_xpath`

`findnodes`  
 same as `get_xpath`

`text @optional_options`  
 Return a string consisting of all the PCDATA and CDATA in an element, without any tags. The text is not XML-escaped: base entities such as `&` and `<` are not escaped.

The `'no_recurse'` option will only return the text of the element, not of any included sub-elements (same as `text_only`).

`text_only`  
 Same as `text` except that the text returned doesn't include the text of sub-elements.

`trimmed_text`  
 Same as `text` except that the text is trimmed: leading and trailing spaces are discarded, consecutive spaces are collapsed

`set_text ($string)`  
 Set the text for the element: if the element is a PCDATA, just set its text, otherwise cut all the children of the element and create a single PCDATA child for it, which holds the text.

`merge ($elt2)`  
 Move the content of `$elt2` within the element

`insert ($tag1, [$optional_atts1], $tag2, [$optional_atts2],...)`  
 For each tag in the list inserts an element `$tag` as the only child of the element. The element gets the optional attributes in `$optional_atts<n>`. All children of the element are set as children of the new element. The upper level element is returned.

```
$p->insert( table => { border=> 1}, 'tr', 'td')
```

put `$p` in a table with a visible border, a single `tr` and a single `td` and return the `table` element:

```
<p><table border="1"><tr><td>original content of p</td></tr></table></p>
```

`wrap_in (@tag)`  
 Wrap elements in `@tag` as the successive ancestors of the element, returns the new element.  
`$elt->wrap_in( 'td', 'tr', 'table')` wraps the element as a single cell in a table for example.

Optionally each tag can be followed by a hashref of attributes, that will be set on the wrapping element:

```
$elt->wrap_in( p => { class => "advisory" }, div => { class => "intro", id => "div1" })
```

`insert_new_elt ($opt_position, $tag, $opt_atts_hashref, @opt_content)`  
 Combines a `new` and a `paste` : creates a new element using `$tag`, `$opt_atts_hashref` and `@opt_content` which are arguments similar to those for `new`, then paste it, using `$opt_position` or `'first_child'`, relative to `$elt`.

Return the newly created element

`erase`  
 Erase the element: the element is deleted and all of its children are pasted in its place.

`set_content ( $optional_atts, @list_of_elt_and_strings) ( $optional_atts, '#EMPTY')`  
 Set the content for the element, from a list of strings and elements. Cuts all the element children, then pastes the list elements as the children. This method will create a PCDATA element for any strings in the list.

The `$optional_atts` argument is the ref of a hash of attributes. If this argument is used then the



previous attributes are deleted, otherwise they are left untouched.

**WARNING:** if you rely on ID's then you will have to set the id yourself. At this point the element does not belong to a twig yet, so the ID attribute is not known so it won't be stored in the ID list.

A content of '#EMPTY' creates an empty element;

`namespace ($optional_prefix)`

Return the URI of the namespace that `$optional_prefix` or the element name belongs to. If the name doesn't belong to any namespace, `undef` is returned.

`local_name`

Return the local name (without the prefix) for the element

`ns_prefix`

Return the namespace prefix for the element

`current_ns_prefixes`

Return a list of namespace prefixes valid for the element. The order of the prefixes in the list has no meaning. If the default namespace is currently bound, "" appears in the list.

`inherit_att ($att, @optional_tag_list)`

Return the value of an attribute inherited from parent tags. The value returned is found by looking for the attribute in the element then in turn in each of its ancestors. If the `@optional_tag_list` is supplied only those ancestors whose tag is in the list will be checked.

`all_children_are ($optional_condition)`

return 1 if all children of the element pass the `$optional_condition`, 0 otherwise

`level ($optional_condition)`

Return the depth of the element in the twig (root is 0). If `$optional_condition` is given then only ancestors that match the condition are counted.

**WARNING:** in a tree created using the `twig_roots` option this will not return the level in the document tree, level 0 will be the document root, level 1 will be the `twig_roots` elements. During the parsing (in a `twig_handler`) you can use the `depth` method on the twig object to get the real parsing depth.

`in ($potential_parent)`

Return true if the element is in the `potential_parent` (`$potential_parent` is an element)

`in_context ($cond, $optional_level)`

Return true if the element is included in an element which passes `$cond` optionally within `$optional_level` levels. The returned value is the including element.

`pcdata`

Return the text of a PCDATA element or `undef` if the element is not PCDATA.

`pcdata_xml_string`

Return the text of a PCDATA element or `undef` if the element is not PCDATA. The text is "XML-escaped" ('&' and '<' are replaced by '&amp;' and '&lt;')

`set_pcdata ($text)`

Set the text of a PCDATA element. This method does not check that the element is indeed a PCDATA so usually you should use `set_text` instead.

`append_pcdata ($text)`

Add the text at the end of a PCDATA element.

`is_cdata`

Return 1 if the element is a CDATA element, returns 0 otherwise.

`is_text`

Return 1 if the element is a CDATA or PCDATA element, returns 0 otherwise.

**cdata**

Return the text of a CDATA element or undef if the element is not CDATA.

**cdata\_string**

Return the XML string of a CDATA element, including the opening and closing markers.

**set\_cdata (\$text)**

Set the text of a CDATA element.

**append\_cdata (\$text)**

Add the text at the end of a CDATA element.

**remove\_cdata**

Turns all CDATA sections in the element into regular PCDATA elements. This is useful when converting XML to HTML, as browsers do not support CDATA sections.

**extra\_data**

Return the extra\_data (comments and PI's) attached to an element

**set\_extra\_data (\$extra\_data)**

Set the extra\_data (comments and PI's) attached to an element

**append\_extra\_data (\$extra\_data)**

Append extra\_data to the existing extra\_data before the element (if no previous extra\_data exists then it is created)

**set\_asis**

Set a property of the element that causes it to be output without being XML escaped by the print functions: if it contains a `< b` it will be output as such and not as `a &lt; b`. This can be useful to create text elements that will be output as markup. Note that all PCDATA descendants of the element are also marked as having the property (they are the ones that are actually impacted by the change).

If the element is a CDATA element it will also be output asis, without the CDATA markers. The same goes for any CDATA descendant of the element

**set\_not\_asis**

Unsets the asis property for the element and its text descendants.

**is\_asis**

Return the asis property status of the element ( 1 or undef)

**closed**

Return true if the element has been closed. Might be useful if you are somewhere in the tree, during the parse, and have no idea whether a parent element is completely loaded or not.

**get\_type**

Return the type of the element: `'#ELT'` for “real” elements, or `'#PCDATA'`, `'#CDATA'`, `'#COMMENT'`, `'#ENT'`, `'#PI'`

**is\_elt**

Return the tag if the element is a “real” element, or 0 if it is PCDATA, CDATA...

**contains\_only\_text**

Return 1 if the element does not contain any other “real” element

**contains\_only (\$exp)**

Return the list of children if all children of the element match the expression \$exp

```
if( $para->contains_only( 'tt' )) { ... }
```

**contains\_a\_single (\$exp)**

If the element contains a single child that matches the expression \$exp returns that element. Otherwise returns 0.

`is_field`  
     same as `contains_only_text`

`is_pCDATA`  
     Return 1 if the element is a PCDATA element, returns 0 otherwise.

`is_ent`  
     Return 1 if the element is an entity (an unexpanded entity) element, return 0 otherwise.

`is_empty`  
     Return 1 if the element is empty, 0 otherwise

`set_empty`  
     Flags the element as empty. No further check is made, so if the element is actually not empty the output will be messed. The only effect of this method is that the output will be `<tag att="value" />`.

`set_not_empty`  
     Flags the element as not empty. if it is actually empty then the element will be output as `<tag att="value"></tag>`

`is_pi`  
     Return 1 if the element is a processing instruction (`#PI`) element, return 0 otherwise.

`target`  
     Return the target of a processing instruction

`set_target ($target)`  
     Set the target of a processing instruction

`data`  
     Return the data part of a processing instruction

`set_data ($data)`  
     Set the data of a processing instruction

`set_pi ($target, $data)`  
     Set the target and data of a processing instruction

`pi_string`  
     Return the string form of a processing instruction (`<?target data?>`)

`is_comment`  
     Return 1 if the element is a comment (`#COMMENT`) element, return 0 otherwise.

`set_comment ($comment_text)`  
     Set the text for a comment

`comment`  
     Return the content of a comment (just the text, not the `<!--` and `-->`)

`comment_string`  
     Return the XML string for a comment (`<!-- comment -->`)

    Note that an XML comment cannot start or end with a `'--'`, or include `'--'` (<http://www.w3.org/TR/2008/REC-xml-20081126/#sec-comments>), if that is the case (because you have created the comment yourself presumably, as it could not be in the input XML), then a space will be inserted before an initial `'--'`, after a trailing one or between two `'--'` in the comment (which could presumably mangle javascript `"hidden"` in an XHTML comment);

`set_ent ($entity)`  
     Set an (non-expanded) entity (`#ENT`). `$entity` is the entity text (`&ent;`)

`ent` Return the entity for an entity (`#ENT`) element (`&ent;`)

`ent_name`

Return the entity name for an entity (`#ENT`) element (`ent`)

`ent_string`

Return the entity, either expanded if the expanded version is available, or non-expanded (`&ent;`) otherwise

`child($offset, $optional_condition)`

Return the `$offset`-th child of the element, optionally the `$offset`-th child that matches `$optional_condition`. The children are treated as a list, so `$elt->child( 0)` is the first child, while `$elt->child( -1)` is the last child.

`child_text($offset, $optional_condition)`

Return the text of a child or `undef` if the sibling does not exist. Arguments are the same as `child`.

`last_child ($optional_condition)`

Return the last child of the element, or the last child matching `$optional_condition` (ie the last of the element children matching the condition).

`last_child_text ($optional_condition)`

Same as `first_child_text` but for the last child.

`sibling($offset, $optional_condition)`

Return the next or previous `$offset`-th sibling of the element, or the `$offset`-th one matching `$optional_condition`. If `$offset` is negative then a previous sibling is returned, if `$offset` is positive then a next sibling is returned. `$offset=0` returns the element if there is no condition or if the element matches the condition, `undef` otherwise.

`sibling_text($offset, $optional_condition)`

Return the text of a sibling or `undef` if the sibling does not exist. Arguments are the same as `sibling`.

`prev_siblings($optional_condition)`

Return the list of previous siblings (optionally matching `$optional_condition`) for the element. The elements are ordered in document order.

`next_siblings($optional_condition)`

Return the list of siblings (optionally matching `$optional_condition`) following the element. The elements are ordered in document order.

`siblings($optional_condition)`

Return the list of siblings (optionally matching `$optional_condition`) of the element (excluding the element itself). The elements are ordered in document order.

`pos($optional_condition)`

Return the position of the element in the children list. The first child has a position of 1 (as in XPath).

If the `$optional_condition` is given then only siblings that match the condition are counted. If the element itself does not match the condition then 0 is returned.

`atts` Return a hash ref containing the element attributes

`set_atts ({ att1=>$att1_val, att2=> $att2_val... })`

Set the element attributes with the hash ref supplied as the argument. The previous attributes are lost (ie the attributes set by `set_atts` replace all of the attributes of the element).

You can also pass a list instead of a hashref: `$elt->set_atts( att1 => 'val1', ...)`

`del_atts`

Deletes all the element attributes.

`att_nb`

Return the number of attributes for the element

`has_atts`

Return true if the element has attributes (in fact return the number of attributes, thus being an alias to `att_nb`)

`has_no_atts`

Return true if the element has no attributes, false (0) otherwise

`att_names`

return a list of the attribute names for the element

`att_xml_string ($att, $options)`

Return the attribute value, where '&', '<' and quote (\" or the value of the quote option at twig creation) are XML-escaped.

The options are passed as a hashref, setting `escape_gt` to a true value will also escape '>' (`$elt('myatt', { escape_gt => 1 })`);

`set_id ($id)`

Set the `id` attribute of the element to the value. See `elt_id` to change the id attribute name

`id` Gets the id attribute value

`del_id ($id)`

Deletes the `id` attribute of the element and remove it from the id list for the document

`class`

Return the `class` attribute for the element (methods on the `class` attribute are quite convenient when dealing with XHTML, or plain XML that will eventually be displayed using CSS)

`lclass`

same as `class`, except that this method is an lvalue, so you can do `$elt->lclass= "foo"`

`set_class ($class)`

Set the `class` attribute for the element to `$class`

`add_class ($class)`

Add `$class` to the element `class` attribute: the new class is added only if it is not already present.

Note that classes are then sorted alphabetically, so the `class` attribute can be changed even if the class is already there

`remove_class ($class)`

Remove `$class` from the element `class` attribute.

Note that classes are then sorted alphabetically, so the `class` attribute can be changed even if the class is already there

`add_to_class ($class)`

alias for `add_class`

`att_to_class ($att)`

Set the `class` attribute to the value of attribute `$att`

`add_att_to_class ($att)`

Add the value of attribute `$att` to the `class` attribute of the element

`move_att_to_class ($att)`

Add the value of attribute `$att` to the `class` attribute of the element and delete the attribute

`tag_to_class`

Set the `class` attribute of the element to the element tag

`add_tag_to_class`

Add the element tag to its `class` attribute

`set_tag_class ($new_tag)`  
 Add the element tag to its `class` attribute and sets the tag to `$new_tag`

`in_class ($class)`  
 Return true (1) if the element is in the class `$class` (if `$class` is one of the tokens in the element `class` attribute)

`tag_to_span`  
 Change the element tag to `span` and set its class to the old tag

`tag_to_div`  
 Change the element tag to `div` and set its class to the old tag

**DESTROY**  
 Frees the element from memory.

`start_tag`  
 Return the string for the start tag for the element, including the `</>` at the end of an empty element tag

`end_tag`  
 Return the string for the end tag of an element. For an empty element, this returns the empty string (`''`).

`xml_string @optional_options`  
 Equivalent to `$elt->sprint( 1)`, returns the string for the entire element, excluding the element's tags (but nested element tags are present)

The `'no_recurse'` option will only return the text of the element, not of any included sub-elements (same as `xml_text_only`).

`inner_xml`  
 Another synonym for `xml_string`

`outer_xml`  
 Another synonym for `sprint`

`xml_text`  
 Return the text of the element, encoded (and processed by the current `output_filter` or `output_encoding` options, without any tag.

`xml_text_only`  
 Same as `xml_text` except that the text returned doesn't include the text of sub-elements.

`set_pretty_print ($style)`  
 Set the pretty print method, amongst `'none'` (default), `'nsgmls'`, `'nice'`, `'indented'`, `'record'` and `'record_c'`

pretty\_print styles:

- `none`  
 the default, no `\n` is used
- `nsgmls`  
 nsgmls style, with `\n` added within tags
- `nice`  
 adds `\n` wherever possible (NOT SAFE, can lead to invalid XML)
- `indented`  
 same as `nice` plus indents elements (NOT SAFE, can lead to invalid XML)
- `record`  
 table-oriented pretty print, one field per line

`record_c`  
 table-oriented pretty print, more compact than `record`, one record per line

`set_empty_tag_style ($style)`  
 Set the method to output empty tags, amongst 'normal' (default), 'html', and 'expand',  
 normal outputs an empty tag '<tag/>', html adds a space '<tag />' for elements that can be empty in XHTML and expand outputs '<tag></tag>'

`set_remove_cdata ($flag)`  
 set (or unset) the flag that forces the twig to output CDATA sections as regular (escaped) PCDATA

`set_indent ($string)`  
 Set the indentation for the indented pretty print style (default is 2 spaces)

`set_quote ($quote)`  
 Set the quotes used for attributes. can be 'double' (default) or 'single'

`cmp ($elt)`  
 Compare the order of the 2 elements in a twig.

`C<$a>` is the `<A>...</A>` element, `C<$b>` is the `<B>...</B>` element

document	<code>\$a-&gt;cmp( \$b)</code>
<code>&lt;A&gt; ... &lt;/A&gt; ... &lt;B&gt; ... &lt;/B&gt;</code>	-1
<code>&lt;A&gt; ... &lt;B&gt; ... &lt;/B&gt; ... &lt;/A&gt;</code>	-1
<code>&lt;B&gt; ... &lt;/B&gt; ... &lt;A&gt; ... &lt;/A&gt;</code>	1
<code>&lt;B&gt; ... &lt;A&gt; ... &lt;/A&gt; ... &lt;/B&gt;</code>	1
<code>\$a == \$b</code>	0
<code>\$a</code> and <code>\$b</code> not in the same tree	undef

`before ($elt)`  
 Return 1 if `$elt` starts before the element, 0 otherwise. If the 2 elements are not in the same twig then return undef.

```
if( $a->cmp( $b) == -1) { return 1; } else { return 0; }
```

`after ($elt)`  
 Return 1 if `$elt` starts after the element, 0 otherwise. If the 2 elements are not in the same twig then return undef.

```
if( $a->cmp( $b) == -1) { return 1; } else { return 0; }
```

other comparison methods

- `lt`
- `le`
- `gt`
- `ge`

`path`  
 Return the element context in a form similar to XPath's short form: `'/root/tag1/..../tag'`

`xpath`  
 Return a unique XPath expression that can be used to find the element again.

It looks like `/doc/sect[3]/title`: unique elements do not have an index, the others do.

`flush`  
 flushes the twig up to the current element (strictly equivalent to `$elt->root->flush`)

private methods  
 Low-level methods on the twig:

```

set_parent      ($parent)
set_first_child ($first_child)
set_last_child  ($last_child)
set_prev_sibling ($prev_sibling)
set_next_sibling ($next_sibling)
set_twig_current
del_twig_current
twig_current
contains_text

```

Those methods should not be used, unless of course you find some creative and interesting, not to mention useful, ways to do it.

### cond

Most of the navigation functions accept a condition as an optional argument. The first element (or all elements for `children` or `ancestors` ) that passes the condition is returned.

The condition is a single step of an XPath expression using the XPath subset defined by `get_xpath`. Additional conditions are:

The condition can be

**#ELT**

return a “real” element (not a PCDATA, CDATA, comment or pi element)

**#TEXT**

return a PCDATA or CDATA element

**regular expression**

return an element whose tag matches the regexp. The regexp has to be created with `qr//` (hence this is available only on perl 5.005 and above)

**code reference**

applies the code, passing the current element as argument, if the code returns true then the element is returned, if it returns false then the code is applied to the next candidate.

### XML::Twig::XPath

XML::Twig implements a subset of XPath through the `get_xpath` method.

If you want to use the whole XPath power, then you can use `XML::Twig::XPath` instead. In this case `XML::Twig` uses `XML::XPath` to execute XPath queries. You will of course need `XML::XPath` installed to be able to use `XML::Twig::XPath`.

See `XML::XPath` for more information.

The methods you can use are:

**findnodes** (\$path)

return a list of nodes found by \$path.

**findnodes\_as\_string** (\$path)

return the nodes found reproduced as XML. The result is not guaranteed to be valid XML though.

**findvalue** (\$path)

return the concatenation of the text content of the result nodes

In order for `XML::XPath` to be used as the XPath engine the following methods are included in `XML::Twig`:

in `XML::Twig`

`getRootNode`

`getParentNode`



```

getChildNodes
in XML::Twig::Elt
string_value
toString
getName
getRootNode
getNextSibling
getPreviousSibling
isElementNode
isTextNode
isPI
isPINode
isProcessingInstructionNode
isComment
isCommentNode
getTarget
getChildNodes
getElementById

```

### **XML::Twig::XPath::Elt**

The methods you can use are the same as on `XML::Twig::XPath` elements:

```

findnodes      ($path)
    return a list of nodes found by $path.

findnodes_as_string ($path)
    return the nodes found reproduced as XML. The result is not guaranteed to be valid XML though.

findvalue      ($path)
    return the concatenation of the text content of the result nodes

```

### **XML::Twig::Entity\_list**

```

new
    Create an entity list.

add      ($ent)
    Add an entity to an entity list.

add_new_ent ($name, $val, $sysid, $pubid, $ndata, $param)
    Create a new entity and add it to the entity list

delete    ($ent or $tag).
    Delete an entity (defined by its name or by the Entity object) from the list.

print     ($optional_filehandle)
    Print the entity list.

list      Return the list as an array

```

### **XML::Twig::Entity**

```

new      ($name, $val, $sysid, $pubid, $ndata, $param)
    Same arguments as the Entity handler for XML::Parser.

print     ($optional_filehandle)
    Print an entity declaration.

name
    Return the name of the entity

val      Return the value of the entity

```

**sysid**  
Return the system id for the entity (for NDATA entities)

**pubid**  
Return the public id for the entity (for NDATA entities)

**ndata**  
Return true if the entity is an NDATA entity

**param**  
Return true if the entity is a parameter entity

**text** Return the entity declaration text.

## EXAMPLES

Additional examples (and a complete tutorial) can be found on the *XML::Twig* Page <<http://www.xmltwig.org/xmltwig/>>

To figure out what flush does call the following script with an XML file and an element name as arguments

```
use XML::Twig;

my ($file, $elt) = @ARGV;
my $t = XML::Twig->new( twig_handlers =>
    { $elt => sub {$_[0]->flush; print "\n[flushed here]\n";} });
$t->parsefile( $file, ErrorContext => 2);
$t->flush;
print "\n";
```

## NOTES

### Subclassing XML::Twig

Useful methods:

#### elt\_class

In order to subclass XML::Twig you will probably need to subclass also XML::Twig::Elt. Use the `elt_class` option when you create the XML::Twig object to get the elements created in a different class (which should be a subclass of XML::Twig::Elt).

#### add\_options

If you inherit XML::Twig new method but want to add more options to it you can use this method to prevent XML::Twig to issue warnings for those additional options.

### DTD Handling

There are 3 possibilities here. They are:

#### No DTD

No doctype, no DTD information, no entity information, the world is simple...

#### Internal DTD

The XML document includes an internal DTD, and maybe entity declarations.

If you use the `load_DTD` option when creating the twig the DTD information and the entity declarations can be accessed.

The DTD and the entity declarations will be `flush`'ed (or `print`'ed) either as is (if they have not been modified) or as reconstructed (poorly, comments are lost, order is not kept, due to it's content this DTD should not be viewed by anyone) if they have been modified. You can also modify them directly by changing the `$twig->{twig_doctype}->{internal}` field (straight from XML::Parser, see the `Doctype` handler doc)

#### External DTD

The XML document includes a reference to an external DTD, and maybe entity declarations.

If you use the `load_DTD` when creating the twig the DTD information and the entity declarations can

be accessed. The entity declarations will be `flush`'ed (or `print`'ed) either as is (if they have not been modified) or as reconstructed (badly, comments are lost, order is not kept).

You can change the doctype through the `$twig->set_doctype` method and print the dtd through the `$twig->dtd_text` or `$twig->dtd_print` methods.

If you need to modify the entity list this is probably the easiest way to do it.

### Flush

Remember that element handlers are called when the element is CLOSED, so if you have handlers for nested elements the inner handlers will be called first. It makes it for example trickier than it would seem to number nested sections (or clauses, or divs), as the titles in the inner sections are handled before the outer sections.

## BUGS

### segfault during parsing

This happens when parsing huge documents, or lots of small ones, with a version of Perl before 5.16.

This is due to a bug in the way weak references are handled in Perl itself.

The fix is either to upgrade to Perl 5.16 or later (`perlbrew` is a great tool to manage several installations of perl on the same machine).

An other, NOT RECOMMENDED, way of fixing the problem, is to switch off weak references by writing `XML::Twig::_set_weakrefs( 0 );` at the top of the code. This is totally unsupported, and may lead to other problems though,

### entity handling

Due to `XML::Parser` behaviour, non-base entities in attribute values disappear if they are not declared in the document: `att="val&ent;"` will be turned into `att => val`, unless you use the `keep_encoding` argument to `XML::Twig->new`

### DTD handling

The DTD handling methods are quite bugged. No one uses them and it seems very difficult to get them to work in all cases, including with several slightly incompatible versions of `XML::Parser` and of `libexpat`.

Basically you can read the DTD, output it back properly, and update entities, but not much more.

So use `XML::Twig` with standalone documents, or with documents referring to an external DTD, but don't expect it to properly parse and even output back the DTD.

### memory leak

If you use a REALLY old Perl (5.005!) and a lot of twigs you might find that you leak quite a lot of memory (about 2Ks per twig). You can use the `dispose` method to free that memory after you are done.

If you create elements the same thing might happen, use the `delete` method to get rid of them.

Alternatively installing the `Scalar::Util` (or `WeakRef`) module on a version of Perl that supports it (>5.6.0) will get rid of the memory leaks automatically.

### ID list

The ID list is NOT updated when elements are cut or deleted.

### change\_gi

This method will not function properly if you do:

```
$twig->change_gi( $old1, $new );
$twig->change_gi( $old2, $new );
$twig->change_gi( $new, $even_newer );
```

sanity check on XML::Parser method calls

XML::Twig should really prevent calls to some XML::Parser methods, especially the `setHandlers` method.

pretty printing

Pretty printing (at least using the 'indented' style) is hard to get right! Only elements that belong to the document will be properly indented. Printing elements that do not belong to the twig makes it impossible for XML::Twig to figure out their depth, and thus their indentation level.

Also there is an unavoidable bug when using `flush` and pretty printing for elements with mixed content that start with an embedded element:

```
<elt><b>b</b>toto<b>bold</b></elt>
```

will be output as

```
<elt>
  <b>b</b>toto<b>bold</b></elt>
```

if you flush the twig when you find the `<b>` element

## Globals

These are the things that can mess up calling code, especially if threaded. They might also cause problem under `mod_perl`.

Exported constants

Whether you want them or not you get them! These are subroutines to use as constant when creating or testing elements

```
PCDATA    return '#PCDATA'
CDATA     return '#CDATA'
PI        return '#PI', I had the choice between PROC and PI :--(
```

Module scoped values: constants

these should cause no trouble:

```
%base_ent= ( '>' => '&gt; ',
              '<' => '&lt; ',
              '&' => '&amp; ',
              "'" => '&apos; ',
              '"' => '&quot; ',
            );
CDATA_START = "<![CDATA[";
CDATA_END   = "]]>";
PI_START    = "<?";
PI_END      = "?>";
COMMENT_START = "<!--";
COMMENT_END   = "-->";
```

pretty print styles

```
( $NSGMLS, $NICE, $INDENTED, $INDENTED_C, $WRAPPED, $RECORD1, $RECORD2) = (1.
```

empty tag output style

```
( $HTML, $EXPAND) = (1..2);
```

Module scoped values: might be changed

Most of these deal with pretty printing, so the worst that can happen is probably that XML output does not look right, but is still valid and processed identically by XML processors.

`$empty_tag_style` can mess up HTML browsers though and changing `$ID` would most likely

create problems.

```
$pretty=0;           # pretty print style
$quote='\"';         # quote for attributes
$INDENT= '  ';       # indent for indented pretty print
$empty_tag_style= 0; # how to display empty tags
$ID           # attribute used as an id ('id' by default)
```

Module scoped values: definitely changed

These 2 variables are used to replace tags by an index, thus saving some space when creating a twig. If they really cause you too much trouble, let me know, it is probably possible to create either a switch or at least a version of XML::Twig that does not perform this optimization.

```
%gi2index;    # tag => index
@index2gi;    # list of tags
```

If you need to manipulate all those values, you can use the following methods on the XML::Twig object:

`global_state`

Return a hashref with all the global variables used by XML::Twig

The hash has the following fields: `pretty`, `quote`, `indent`, `empty_tag_style`, `keep_encoding`, `expand_external_entities`, `output_filter`, `output_text_filter`, `keep_atts_order`

`set_global_state ($state)`

Set the global state, `$state` is a hashref

`save_global_state`

Save the current global state

`restore_global_state`

Restore the previously saved (using `Lsave_global_state`) state

## TODO

SAX handlers

Allowing XML::Twig to work on top of any SAX parser

multiple twigs are not well supported

A number of twig features are just global at the moment. These include the ID list and the “tag pool” (if you use `change_gi` then you change the tag for ALL twigs).

A future version will try to support this while trying not to be too hard on performance (at least when a single twig is used!).

## AUTHOR

Michel Rodriguez <mirod@cpan.org>

## LICENSE

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Bug reports should be sent using: RT <<http://rt.cpan.org/NoAuth/Bugs.html?Dist=XML-Twig>>

Comments can be sent to [mirod@cpan.org](mailto:mirod@cpan.org)

The XML::Twig page is at <<http://www.xmltwig.org/xmltwig/>> It includes the development version of the module, a slightly better version of the documentation, examples, a tutorial and a: *Processing XML efficiently with Perl and XML::Twig*: <<http://www.xmltwig.org/xmltwig/tutorial/index.html>>

## SEE ALSO

Complete docs, including a tutorial, examples, an easier to use HTML version of the docs, a quick reference card and a FAQ are available at <<http://www.xmltwig.org/xmltwig/>>

git repository at <<http://github.com/mirod/xmltwig>>

XML::Parser, XML::Parser::Expat, XML::XPath, Encode, Text::Iconv, Scalar::Utils

### Alternative Modules

XML::Twig is not the only XML::Processing module available on CPAN (far from it!).

The main alternative I would recommend is XML::LibXML.

Here is a quick comparison of the 2 modules:

XML::LibXML, actually libxml2 on which it is based, sticks to the standards, and implements a good number of them in a rather strict way: XML, XPath, DOM, RelaxNG, I must be forgetting a couple (XInclude?). It is fast and rather frugal memory-wise.

XML::Twig is older: when I started writing it XML::Parser/expat was the only game in town. It implements XML and that's about it (plus a subset of XPath, and you can use XML::Twig::XPath if you have XML::XPathEngine installed for full support). It is slower and requires more memory for a full tree than XML::LibXML. On the plus side (yes, there is a plus side!) it lets you process a big document in chunks, and thus let you tackle documents that couldn't be loaded in memory by XML::LibXML, and it offers a lot (and I mean a LOT!) of higher-level methods, for everything, from adding structure to "low-level" XML, to shortcuts for XHTML conversions and more. It also DWIMs quite a bit, getting comments and non-significant whitespaces out of the way but preserving them in the output for example. As it does not stick to the DOM, is also usually leads to shorter code than in XML::LibXML.

Beyond the pure features of the 2 modules, XML::LibXML seems to be preferred by "XML-purists", while XML::Twig seems to be more used by Perl Hackers who have to deal with XML. As you have noted, XML::Twig also comes with quite a lot of docs, but I am sure if you ask for help about XML::LibXML here or on Perlmonks you will get answers.

Note that it is actually quite hard for me to compare the 2 modules: on one hand I know XML::Twig inside-out and I can get it to do pretty much anything I need to (or I improve it ;—), while I have a very basic knowledge of XML::LibXML. So feature-wise, I'd rather use XML::Twig ;—). On the other hand, I am painfully aware of some of the deficiencies, potential bugs and plain ugly code that lurk in XML::Twig, even though you are unlikely to be affected by them (unless for example you need to change the DTD of a document programmatically), while I haven't looked much into XML::LibXML so it still looks shinny and clean to me.

That said, if you need to process a document that is too big to fit memory and XML::Twig is too slow for you, my reluctant advice would be to use "bare" XML::Parser. It won't be as easy to use as XML::Twig: basically with XML::Twig you trade some speed (depending on what you do from a factor 3 to... none) for ease-of-use, but it will be easier IMHO than using SAX (albeit not standard), and at this point a LOT faster (see the last test in <[http://www.xmltwig.org/article/simple\\_benchmark/](http://www.xmltwig.org/article/simple_benchmark/)>).