eGenix.com

# mxTextTools

Fast Text Parsing
and Processing
for Python

Version 3.1

Printed in Germany.

# Contents

# 1. Introduction

mxTextTools is a collection of high-speed string manipulation routines and new Python objects for dealing with common text processing tasks.

One of the major features of this package is the integrated *mxTextTools Tagging Engine* which allows accessing the speed of compiled C programs while maintaining the portability of Python. The Tagging Engine uses byte code "programs" written in form of Python tuples. These programs are then translated into an internal binary form which gets processed by a very fast virtual machine designed specifically for scanning text data.

As a result, the Tagging Engine allows parsing text at higher speeds than e.g. regular expression packages while still maintaining the flexibility of programming the parser in Python. Callbacks and user-defined matching functions extends this approach far beyond what you could do with other common text processing methods.

A note about the word *tagging*: this originated from what is done in SGML, HTML and XML to mark some text with a certain extra information. The Tagging Engine extends this notion to assigning Python objects to text substrings. Every substring marked in this way carries a 'tag' (the tag object) which can be used to do all kinds of useful things.

Two other major features are the search and character set objects provided by the package. Both are implemented in C to give you maximum performance on all supported platforms.

If you are looking for more tutorial style documentation of mxTextTools, there's a new book by David Mertz about *Text Processing with Python* which covers mxTextTools and other text oriented tools at great length.

# 2. mxTextTools Tagging Engine

The *Tagging Engine* is a low-level virtual machine (VM) which executes text search specific byte codes. This byte code is passed to the engine in form of Tag Tables which define the "program" to execute in terms of commands and command arguments.

The Tagging Engine is capable of handling 8-bit text and Unicode (with some minor exceptions). Even combinations of the two string formats are accepted, but should be avoided for performance reasons in production code.

## 2.1 Tag List

Marking certain parts of a text should not involve storing hundreds of small strings. This is why the Tagging Engine uses a specially formatted list of tuples to return the results.

A *Tag List* is a list of tuples marking certain slices of a text. The tuples always have the format

```
(object, left_index, right_index, subtags)
```

with the following meanings:

- `object` contains information about the slice `[left_index:right_index]` in a referenced text
- `left_index` and `right_index` are indexes into a referenced text
- `subtags` is either another Tag List created by recursively invoking the Tagging Engine or `None`.

  Note: Only the commands `Table` and `TableInList` create new Tag Lists and make them available via `subtags` and then only if the Tagging Engine was not called with `None` as value for the `taglist`. All other commands set this tuple entry to `None`. This is important to know if you want to analyze a generated Tag List, since it may require recursing into the `subtags` Tag List if that entry is not `None`.

## 2.2 Tag Table

To create such Tag Lists, you have to define a Tag Table and let the Tagging Engine use it to mark the text.

*Tag Table*s are defined using standard Python tuples containing other tuples in a specific format:

```
tag_table = (('lowercase',AllIn,a2z,+1,+2),
        ('upper',AllIn,A2Z,+1),
        (None,AllIn,white+newline,+1),
        (None,AllNotIn,alpha+white+newline,+1),
        (None,EOF,Here,-4)) # EOF
```

The tuples contained in the table use a very simple format:

```
(tagobj, command+flags, command_argument
                          [,jump_no_match] [,jump_match=+1])
```

The meaning of the tuple contents is as follows:

- `tagobj` refers to the tag object that is to be associated with the match
- `command` defines the Tagging Engine command to execute
- `flags` may be provided optionally to adjust the behavior of the commands or how the tagobj is put to use
- `command_argument` is the argument to the command and defined by the command (see 2.5 Tagging Engine Commands for details)
- `jump_no_match` defines the offset to apply to the program counter (the index in the current Tag Table) in case the command did not generate a match
- `jump_match` defines the offset to apply to the program counter (the index in the current Tag Table) in case the command did generate a match; default is to continue with the next command tuple

### 2.2.1 Jump Target Support

To simplify writing Tag Table definitions, the Tag Table compiler also allows using string jump targets instead of jump offsets in the tuples:

```
tag_table = (

    # Start scanning text and mark as lower or upper case
    'start',
    ('lowercase',AllIn,a2z,+1,'skip'),
    ('uppercase',AllIn,A2Z,'skip'),

    # Skip all whitespace
    'skip',
    (None,AllIn,white+newline,+1),
    (None,AllNotIn,alpha+white+newline,+1),

    # Continue until EOF
    (None,EOF,Here,'start')) # EOF
```

The string entries `'start'` and `'skip'` in the table can be used as jump targets for `jump_no_match` and `jump_match`.

When compiling the definition using `TagTable()` or `UnicodeTagTable()` they will get replaced with the corresponding numeric relative offsets at compile time.

The strings entries themselves get replaced by a special command `JumpTarget` which was added for this purpose. It is implemented as no operation (NOP) command and only serves as placeholder, so that the indexes in the Tag Table don't change when applying the jump target replacement.

### 2.2.2 TagTable Objects

Starting with version 2.1.0 of mxTextTools, the Tagging Engine no longer uses Tag Tables and their tuple entries directly, but instead compiles the Tag Table definitions into special *TagTable* objects. These objects are then processed by the Tagging Engine.

Even though the `tag()` Tagging Engine API compiles Tag Table definitions into the TagTable object on-the-fly, you can also compile the definitions yourself and then pass the TagTable object directly to `tag()`.

### 2.2.3 Tag Table Compiler

The Tagging Engine uses compiled TagTable instances for performing the scanning. These TagTables are Python objects which can be created explicitly using a tag table definition in form of a tuple or a list (the latter are not cacheable, so it's usually better to transform the list into a tuple before passing it to the TagTable constructor).

The `TagTable()` constructor will "compile" and check the tag table definition. It then stores the table in an internal data structure which allows fast access from within the Tagging Engine. The compiler also takes care of any needed conversions such as Unicode to string or vice-versa.

There are generally two different kinds of compiled TagTables: one for scanning 8-bit strings and one for Unicode. `tag()` will complain if you try to scan strings with a UnicodeTagTable or Unicode with a string TagTable.

Note that `tag()` can take TagTables and tuples as tag table input. If given a tuple, it will automatically compile the tuple into a TagTable needed for the requested type of text (string or Unicode).

### 2.2.4 Caching of Compiled Tag Tables

The `TagTable()` constructor caches compiled TagTables if they are defined by a tuple and declared as cacheable. In that case, the compile TagTable will be stored in a dictionary addressed by the definition tuple's `id()` and be reused if the same compilation is requested again at some later point. The cache dictionary is exposed to the user as `tagtable_cache` dictionary. It has a hard limit of 100 entries, but can also be managed by user routines to lower this limit.

## 2.3 Tag Table Processing

The Tagging Engine reads the Tag Table starting at the top entry. While performing the command actions (see below for details) it moves a read-head over the characters of the text. The engine stops when a command fails to match and no alternative is given or when it reaches a non-existing entry, e.g. by jumping beyond the end of the table.

Tag Table entries are processed as follows:

If the `command` matched, say the slice `text[l:r]`, the default action is to append `(tagobj,l,r,subtags)` to the taglist (this behavior can be modified by using special `flags`; if you use `None` as `tagobj`, no tuple is appended) and to continue matching with the table entry that is reached by adding `jump_match` to the current position (think of them as relative jump offsets). The head position of the engine stays where the command left it (over index `r`), e.g. for `(None,AllIn,'A')` right after the last 'A' matched.

In case the `command` does not match, the engine either continues at the table entry reached after skipping `jump_no_match` entries, or if this value is not given, terminates matching the current table and returns *not matched*. The head position is always restored to the position it was in before the non-matching command was executed, enabling backtracking.

The format of the `command_argument` is dependent on the command. It can be a string, a set, a search object, a tuple of some other wild animal from Python land. See the command section below for details.

A table matches a string if and only if the Tagging Engine reaches a table index that lies beyond the end of the table. The engine then returns *matched ok*. Jumping beyond the start of the table (to a negative table index) causes the table to return with result *failed to match*.

## 2.4 Context Object Support

The Tagging Engine supports carrying along an optional context object. This object can be used for storing data specific to the tagging procedure, error information, etc.

You can access the context object by using a Python function as tag object which is then called with the context object as last argument if `CallTag` is used as command flag or in matching functions which are called as a result of the `Call` or `CallArg` commands.

To remain backward compatible, the context object is only provided as last argument if given to the `tag()` function (mxTextTools prior to 2.0 did not support this object).

New commands which make use of the context object at a lower level will eventually appear in the Tagging Engine in future releases.

## 2.5 Tagging Engine Commands

The commands and constants used here are integers defined in `Constants/TagTables.py` and imported into the package's root module. For the purpose of explaining the taken actions we assume that the tagging engine was called with `tag(text,table,start=0,stop=len(text))`. The current head position is indicated by `x`.

Note that for most commands, the matching string may not be empty. This is checked by the Tag Table Compiler.

| *Command* | *Matching Argument* | *Action* |
|---|---|---|
| Fail | Here | Causes the engine to fail matching at the current head position. |
| Jump | To | Causes the engine to perform a relative jump by `jump_no_match` entries. |
| AllIn | string | Matches all characters found in `text[x:stop]` up to the first that is not included in string. At least one character must match. |
| AllNotIn | string | Matches all characters found in `text[x:stop]` up to the first that is included in string. At least one character must match. |
| AllInSet | set | Matches all characters found in `text[x:stop]` up to the first that is not included in the string set. At least one character must match. Note: String sets only work with 8-bit text. Use `AllInCharSet` if you plan to use the tag table with 8-bit and Unicode text. |
| AllInCharSet | CharSet object | Matches all characters found in `text[x:stop]` up to the first that is not included in the CharSet. At least one character must match. |
| Is | character | Matches iff `text[x] == character`. |
| IsNot | character | Matches iff `text[x] != character`. |
| IsIn | string | Matches iff `text[x] is in string`. |
| IsNotIn | string | Matches iff `text[x] is not in string`. |

| *Command* | *Matching Argument* | *Action* |
|---|---|---|
| IsInSet | set | Matches iff `text[x] is in set`. Note: String sets only work with 8-bit text. Use `IsInCharSet` if you plan to use the tag table with 8-bit and Unicode text. |
| IsInCharSet | CharSet object | Matches iff `text[x]` is contained in the CharSet. |
| Word | string | Matches iff `text[x:x+len(string)] == string`. |
| WordStart | string | Matches all characters up to the first occurance of string in `text[x:stop]`.<br><br>If string is not found, the command does not match and the head position remains unchanged. Otherwise, the head stays on the first character of string in the found occurance.<br><br>At least one character must match. |
| WordEnd | string | Matches all characters up to the first occurance of string in `text[x:stop]`.<br><br>If string is not found, the command does not match and the head position remains unchanged. Otherwise, the head stays on the last character of string in the found occurance. |
| sWordStart | TextSearch object | Same as WordStart except that the TextSearch object is used to perform the necessary action (which can be much faster) and zero matching characters are allowed. |
| sWordEnd | TextSearch object | Same as WordEnd except that the TextSearch object is used to perform the necessary action (which can be much faster). |
| sFindWord | TextSearch object | Uses the TextSearch object to find the given substring.<br><br>If found, the tagobj is assigned only to the slice of the substring. The characters leading up to it are ignored.<br><br>The head position is adjusted to right after the substring -- just like for sWordEnd. |
| Call | function | Calls the matching `function(text,x,stop)` or `function(text,x,stop,context)` if a context object was provided to the `tag()` function call.<br><br>The function must return the index `y` of the character in `text[x:stop]` right after the matching substring.<br><br>The entry is considered to be matching, iff `x != y`. The engines |

| *Command* | *Matching Argument* | *Action* |
|---|---|---|
| | | head is positioned on `y` in that case. |
| CallArg | (function,[arg0,...]) | Same as Call except that `function(text,x,stop[,arg0,...])` or `function(text,x,stop,[,arg0,...],context)` (if a `context` object is used) is being called.<br><br>The command argument must be a tuple. |
| Table | tagtable or ThisTable | Matches iff tagtable matches `text[x:stop]`.<br><br>This calls the engine recursively.<br><br>In case of success the head position is adjusted to point right after the match and the returned taglist is made available in the subtags field of this table's taglist entry.<br><br>You may pass the special constant `ThisTable` instead of a Tag Table if you want to call the current table recursively. |
| SubTable | tagtable or ThisTable | Same as Table except that the subtable reuses this table's tag list for its tag list. The `subtags` entry is set to None.<br><br>You may pass the special constant `ThisTable` instead of a Tag Table if you want to call the current table recursively. |
| TableInList | (list_of_tables,index) | Same as Table except that the matching table to be used is read from the `list_of_tables` at position `index` whenever this command is executed.<br><br>This makes self-referencing tables possible which would otherwise not be possible (since Tag Tables are immutable tuples).<br><br>Note that it can also introduce circular references, so be warned ! |
| SubTableInList | (list_of_tables,index) | Same as TableInList except that the subtable reuses this table's tag list. The `subtags` entry is set to `None`. |
| EOF | Here | Matches iff the head position is beyond `stop`. The match recorded by the Tagging Engine is the `text[stop:stop]`. |
| Skip | offset | Always matches and moves the head position to `x + offset`. |
| Move | position | Always matches and moves the head position to `slice[position]`. Negative indices move the head to `slice[len(slice)+position+1]`, e.g. (None,Move,-1) |

| *Command* | *Matching Argument* | *Action* |
|---|---|---|
| | | moves to EOF. `slice` refers to the current text slice being worked on by the Tagging Engine. |
| JumpTarget | Target String | Always matches, does not move the head position.<br><br>This command is only used internally by the Tag Table compiler, but can also be used for writing Tag Table definitions, e.g. to follow the path the Tagging Engine takes through a Tag Table definition. |
| Loop | count | Remains undocumented for this release. |
| LoopControl | Break/Reset | Remains undocumented for this release. |

## 2.6 Tagging Engine Command Flags

The following flags can be added to the command integers defined in the previous section:

`CallTag`

Instead of appending `(tagobj,l,r,subtags)` to the taglist upon successful matching, call `tagobj(taglist,text,l,r,subtags)` or `tagobj(taglist,text,l,r,subtags,context)` if a context object was passed to the `tag()` function.

`AppendMatch`

Instead of appending `(tagobj,l,r,subtags)` to the taglist upon successful matching, append the match found as string.

Note that this will produce non-standard taglists ! It is useful in combination with `join()` though and can be used to implement smart split() replacements algorithms.

`AppendToTagobj`

Instead of appending `(tagobj,l,r,subtags)` to the taglist upon successful matching, call `tagobj.append((None,l,r,subtags))`.

`AppendTagobj`

Instead of appending `(tagobj,l,r,subtags)` to the taglist upon successful matching, append `tagobj` itself.

Note that this can cause the taglist to have a non-standard format, i.e. functions relying on the standard format could fail.

This flag is mainly intended to build *join-lists* usable by the `join()`-function (see below).

`LookAhead`

If this flag is set, the current position of the head will be reset to `l` (the left position of the match) after a successful match.

This is useful to implement look-ahead strategies.

Using the flag has no effect on the way the tagobj itself is treated, i.e. it will still be processed in the usual way.

Some additional constants that can be used as argument or relative jump position:

`To`

Useful as argument for 'Jump'.

`Here`

Useful as argument for 'Fail' and 'EOF'.

`MatchOk`

Jumps to a table index beyond the tables end, causing the current table to immediately return with 'matches ok'.

`MatchFail`

Jumps to a negative table index, causing the current table to immediately return with 'failed to match'.

`ToBOF,ToEOF`

Useful as arguments for 'Move': `(None,Move,ToEOF)` moves the head to the character behind the last character in the current slice, while `(None,Move,ToBOF)` moves to the first character.

`ThisTable`

Useful as argument for 'Table' and 'SubTable'. See above for more information.

Internally, the Tag Table is used as program for a state machine which is coded in C and accessible through the package as `tag()` function along with the constants used for the commands (e.g. `Allin`, `AllNotIn`, etc.).

Note that in computer science one normally differentiates between finite state machines, pushdown automata and Turing machines. The Tagging Engine offers all these levels of complexity depending on which techniques you use, yet the basic structure of the engine is best compared to a finite state machine.

> Tip: If you are getting an error **TypeError: call of a non-function** while writing a table definition, you probably have a missing comma (',') somewhere in the tuple !

## 2.7 Third Party Tools for Tag Table Writing

Writing these Tag Tables by hand is not always easy. However, since Tag Tables can easily be generated using Python code, it is possible to write tools which convert meta-languages into Tag Tables which then run on all platforms supported by mxTextTools at nearly C speeds.

- Mike C. Fletcher has written a nice tools for generating Tag Tables using an EBNF notation. You may want to check out his *SimpleParse* add-on for mxTextTools.

- Tony J. Ibbs has also started to work in this direction. His *meta-language for mxTextTools* aims at simplifying the task of writing Tag Table tuples.

More references to third party extensions or applications built on top of mxTextTools can be found in the *Add-ons Section*.

## 2.8 Debugging

The packages includes a nearly complete Python emulation of the Tagging Engine in the Examples subdirectory (pytag.py). Though it is unsupported it might still provide some use since it has a built-in debugger that will let you step through the Tag Tables as they are executed. See the source for further details.

As an alternative you can build a version of the Tagging Engine that provides lots of debugging output. The feature is only enabled if the module is compiled with debug support and output is only generated if Python is run in debugging mode (use the Python interpreter flag: `python -d script.py`).

The resulting log file is named mxTextTools.log. It will be created in the current working directory; messages are always appended to the file so no trace is lost until you explicitly erase the log file. If the log file can not be opened, the module will use `stderr` for reporting.

To have the package compiled using debug support, prepend the distutils command `mx_autoconf --enable-debugging` to the `build` or `install` command. This will then enable the define and compile a debugging version of the code, e.g.

```
cd egenix-mx-base-X.X.X
python setup.py mx_autoconf --enable-debugging install
```

installs a debugging enabled version of mxODBC on both Unix and Windows (provided you have a compiler installed).

The mxTextTools.log file should give a fairly good insight into the workings of the Tag Engine (though it still isn't as elegant as it could be).

# 3. mx.TextTools.TextSearch Object

TextSearch objects provide a very fast way of doing repeated searches for a string in one or more target texts.

The TextSearch object is immutable and usable for one search string per object only. Once created, the TextSearch objects can be applied to as many text strings as you like -- much like compiled regular expressions. Matching is done exact (doing translations on-the-fly if supported by the search algorithm).

Furthermore, the TextSearch objects can be pickled and implement the copy protocol as defined by the copy module. Comparisons and hashing are not implemented (the objects are stored by id in dictionaries).

Depending on the search algorithm, TextSearch objects can search in 8-bit strings and/or Unicode. Searching in memory buffers is currently not supported. Accordingly, the search string itself may also be an 8-bit string or Unicode.

## 3.1 TextSearch Object Constructors

In older versions of mxTextTools there were two separate constructors for search objects: `BMS()` for Boyer-Moore and `FS()` for the (unpublished) FastSearch algorithm. In mxTextTools 2.1 these two constructors were merged into one having the algorithm type as parameter.

`TextSearch(match,translate=None,algorithm=default_algorithm)`

Create a TextSearch substring search object for the string match implementing the algorithm specified in the constructor.

`algorithm` defines the algorithm to use. Possible values are:

- `BOYERMOORE`

  Enhanced Boyer-Moore-Horspool style algorithm for searching in 8-bit text. Unicode is not supported. On-the-fly translation is supported.

- `FASTSEARCH`

  Enhanced Boyer-Moore style algorithm for searching in 8-bit text. This algorithm provides better performance for match patterns

having repeating sequences, like e.g. DNA strings. Unicode is not supported. On-the-fly translation is supported.

Not included in the public release of mxTextTools.

- `TRIVIAL`

  Trivial right-to-left search algorithm. This algorithm can be used to search in 8-bit text and Unicode. On-the-fly translation is not supported.

`algorithm` defaults to `BOYERMOORE` (or `FASTSEARCH` if available) for 8-bit match strings and `TRIVIAL` for Unicode match strings.

`translate` is an optional translate-string like the one used in the module 're', i.e. a 256 character string mapping the oridnals of the base character set to new characters. It is supported by the `BOYERMOORE` and the `FASTSEARCH` algorithm only.

This function supports keyword arguments.

`BMS(match[,translate])`

**DEPRECATED:** Use TextSearch(match, translate, `BOYERMOORE`) instead.

`FS(match[,translate])`

**DEPRECATED:** Use TextSearch(match, translate, `FASTSEARCH`) instead.

## 3.2 TextSearch Object Methods

The TextSearch object has the following methods:

`.search(text,[start=0,stop=len(text)])`

Search for the substring match in text, looking only at the slice `[start:stop]` and return the slice `(l,r)` where the substring was found, or `(start,start)` if it was not found.

`.find(text,[start=0,stop=len(text)])`

Search for the substring match in text, looking only at the slice `[start:stop]` and return the index where the substring was found, or `-1` if it was not found. This interface is compatible with `string.find`.

`.findall(text,start=0,stop=len(text))`

Same as `search()`, but return a list of all non-overlapping slices `(l,r)` where the match string can be found in text.

Note that translating the text before doing the search often results in a better performance. Use `string.translate()` to do that efficiently.

## 3.3 TextSearch Object Attributes

To provide some help for reflection and pickling the TextSearch object gives (read-only) access to these attributes.

`.match`

The string that the search object will look for in the search text.

`.translate`

The translate string used by the object or None (if no translate string was passed to the constructor).

`.algorithm`

The algorithm used by the TextSearch object. For possible values, see the TextSearch() constructor documentation.

# 4. mx.TextTools.CharSet Object

The CharSet object is an immutable object which can be used for character set based string operations like text matching, searching, splitting etc.

CharSet objects can be pickled and implement the copy protocol as defined by the copy module as well as the 'in'-protocol, so that `c in charset` works as expected. Comparisons and hashing are not implemented (the objects are stored by id in dictionaries).

The objects support both 8-bit strings and UCS-2 Unicode in both the character set definition and the various methods. Mixing of the supported types is also allowed. Memory buffers are currently not supported.

## 4.1 CharSet Object Constructor

`CharSet(definition)`

Create a CharSet object for the given character set definition.

`definition` may be an 8-bit string or Unicode.

The constructor supports the re-module syntax for defining character sets: "a-e" maps to "abcde" (the backslash can be used to escape the special meaning of "-", e.g. r"a\-e" maps to "a-e") and "^a-e" maps to the set containing all but the characters "abcde".

Note that the special meaning of "^" only applies if it appears as first character in a CharSet definition. If you want to create a CharSet with the single character "^", then you'll have to use the escaped form: r"\^". The non-escape form "^" would result in a CharSet matching all characters.

To add the backslash character to a CharSet you have to escape with itself: r"\\".

Watch out for the Python quoting semantics in these explanations: the small r in front of some of these strings makes the raw Python literal strings which means that no interpretation of backslashes is applied: r"\\" == "\\\\" and r"a\-e" == "a\\-e".

## 4.2 CharSet Object Methods

The CharSet object has these methods:

`.contains(char)`

Return 1 if char is included in the character set, 0 otherwise.

`.search(text[, direction=1, start=0, stop=len(text)])`

Search `text[start:stop]` for the first character included in the character set. Returns `None` if no such character is found or the index position of the found character.

`direction` defines the search direction: a positive value searches forward starting from `text[start]`, while a negative value searches backwards from `text[stop-1]`.

`.match(text[, direction=1, start=0, stop=len(text)])`

Look for the longest match of characters in `text[start:stop]` which appear in the character set. Returns the length of this match as integer.

`direction` defines the match direction: a positive value searches forward starting from `text[start]` giving a prefix match, while a negative value searches backwards from `text[stop-1]` giving a suffix match.

`.split(text, [,start=0, stop=len(text)])`

Split `text[start:stop]` into a list of substrings using the character set definition, omitting the splitting parts and empty substrings.

`.splitx(text, [,start=0, stop=len(text)])`

Split `text[start:stop]` into a list of substrings using the character set definition, such that every second entry consists only of characters in the set.

`.strip(text[, where=0, start=0, stop=len(text)])`

Strip all characters in `text[start:stop]` appearing in the character set.

`where` indicates where to strip (<0: left; =0: left and right; >0: right).

## 4.3 CharSet Object Attributes

To provide some help for reflection and pickling the CharSet object gives (read-only) access to these attributes.

`.definition`

The definition string which was passed to the constructor.

# 5. mx.TextTools Functions

These functions are defined in the package:

`tag(text,tagtable,sliceleft=0,sliceright=len(text),taglist=[],context=None)`

This is the interface to the Tagging Engine.

`text` may be an 8-bit string or Unicode. `tagtable` must be either Tag Table definition (a tuple of tuples) or a compiled TagTable() object matching the `text` string type. Tag Table definitions are automatically compiled into TagTable() objects by this constructor.

Returns a tuple `(success, taglist, nextindex)`, where nextindex indicates the next index to be processed after the last character matched by the Tag Table.

In case of a non match (success == 0), it points to the error location in text. If you provide a tag list it will be used for the processing.

Passing `None` as taglist results in no tag list being created at all.

`context` is an optional extension to the Tagging Engine introduced in version 2.1.0 of mxTextTools. If given, it is made available to the Tagging Engine during the scan and can be used for e.g. `CallTag`.

This function supports keyword arguments.

`join(joinlist[,sep='',start=0,stop=len(joinlist)])`

This function works much like the corresponding function in module 'string'. It pastes slices from other strings together to form a new string.

The format expected as *joinlist* is similar to a tag list: it is a sequence of tuples `(string,l,r[,...])` (the resulting string will then include the slice `string[l:r]`) or strings (which are copied as a whole). Extra entries in the tuple are ignored.

The optional argument sep is a separator to be used in joining the slices together, it defaults to the empty string (unlike string.join). start and stop allow to define the slice of joinlist the function will work in.

> Important: The syntax used for negative slices is different than the Python standard: -1 corresponds to the first character *after* the string, e.g. ('Example',0,-1) gives 'Example' and not 'Exampl', like in Python. To avoid confusion, don't use negative indices.

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`cmp(a,b)`

Compare two valid taglist tuples w/r to their slice position. This is useful for sorting joinlists and not much slower than sorting integers, since the function is coded in C.

`joinlist(text,list[,start=0,stop=len(text)])`

Produces a joinlist suitable for passing to `join()` from a list of tuples `(replacement,l,r,...)` in such a way that all slices `text[l:r]` are replaced by the given replacement.

A few restrictions apply:

- the list must be sorted ascending (e.g. using the cmp() as compare function)
- it may not contain overlapping slices
- the slices may not contain negative indices
- if the taglist cannot contain overlapping slices, you can give this function the taglist produced by tag() directly (sorting is not needed, as the list will already be sorted)

If one of these conditions is not met, a ValueError is raised.

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`upper(string)`

Returns the string with all characters converted to upper case.

Note that the translation string used is generated at import time. Locale settings will only have an effect if set prior to importing the package.

This function is almost twice as fast as the one in the string module.

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`lower(string)`

Returns the string with all characters converted to lower case. Same note as for upper().

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`is_whitespace(text,start=0,stop=len(text))`

Returns 1 iff text[start:stop] only contains whitespace characters (as defined in Constants/Sets.py), 0 otherwise.

This function can handle 8-bit string or Unicode input.

`replace(text,what,with,start=0,stop=len(text))`

Works just like string.replace() -- only faster since a search object is used in the process.

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`multireplace(text,replacements,start=0,stop=len(text))`

Apply multiple replacement to a text in one processing step.

replacements must be list of tuples (replacement, left, right). The replacement string is then used to replace the slice text[left:right].

Note that the replacements do not affect one another w/r to indexing: indices always refer to the original text string.

Replacements may not overlap. Otherwise a ValueError is raised.

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`find(text,what,start=0,stop=len(text))`

Works just like string.find() -- only faster since a search object is used in the process.

This function can handle 8-bit string and Unicode input.

`findall(text,what,start=0,stop=len(text))`

Returns a list of slices representing all non-overlapping occurances of what in text[start:stop]. The slices are given as 2-tuples `(left,right)` meaning that `what` can be found at text[left:right].

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`collapse(text,separator=' ')`

Takes a string, removes all line breaks, converts all whitespace to a single separator and returns the result. Tim Peters will like this one with separator '-'.

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`charsplit(text,char,start=0,stop=len(text))`

Returns a list that results from splitting text[start:stop] at all occurances of the character given in char.

This is a special case of string.split() that has been optimized for single character splitting running 40% faster.

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`splitat(text,char,nth=1,start=0,stop=len(text))`

Returns a 2-tuple that results from splitting text[start:stop] at the nth occurance of char.

If the character is not found, the second string is empty. nth may also be negative: the search is then done from the right and the first string is empty in case the character is not found.

The splitting character itself is not included in the two substrings.

This function can handle mixed 8-bit string / Unicode input. Coercion is always towards Unicode.

`suffix(text,suffixes,start=0,stop=len(text)[,translate])`

Looks at text[start:stop] and returns the first matching suffix out of the tuple of strings given in suffixes.

If no suffix is found to be matching, None is returned. An empty suffix (") matches the end-of-string.

The optional 256 char translate string is used to translate the text prior to comparing it with the given suffixes. It uses the same format as the search object translate strings. If not given, no translation is performed and the match done exact. On-the-fly translation is not supported for Unicode input.

This function can handle either 8-bit strings or Unicode. Mixing these input types is not supported.

`prefix(text,prefixes,start=0,stop=len(text)[,translate])`

Looks at text[start:stop] and returns the first matching prefix out of the tuple of strings given in prefixes.

If no prefix is found to be matching, None is returned. An empty prefix (") matches the end-of-string.

The optional 256 char translate string is used to translate the text prior to comparing it with the given suffixes. It uses the same format as the search object translate strings. If not given, no translation is performed and the match done exact. On-the-fly translation is not supported for Unicode input.

This function can handle either 8-bit strings or Unicode. Mixing these input types is not supported.

`splitlines(text)`

Splits text into a list of single lines.

The following combinations are considered to be line-ends: '\r', '\r\n', '\n'; they may be used in any combination. The line-end indicators are removed from the strings prior to adding them to the list.

This function allows dealing with text files from Macs, PCs and Unix origins in a portable way.

This function can handle 8-bit string and Unicode input.

`countlines(text)`

Returns the number of lines in text.

Line ends are treated just like for splitlines() in a portable way.

This function can handle 8-bit string and Unicode input.

`splitwords(text)`

Splits text into a list of single words delimited by whitespace.

This function is just here for completeness. It works in the same way as string.split(text). Note that CharSet().split() gives you much more control over how splitting is performed. whitespace is defined as given below (see *Constants*).

This function can handle 8-bit string and Unicode input.

`str2hex(text)`

Returns text converted to a string consisting of two byte HEX values, e.g. ',.-' is converted to '2c2e2d'. The function uses lowercase HEX characters.

Unicode input is *not* supported.

`hex2str(hex)`

Returns the string hex interpreted as two byte HEX values converted to a string, e.g. '223344' becomes '"3D'. The function expects lowercase HEX characters per default but can also work with upper case ones.

Unicode input is *not* supported.

`isascii(text)`

Returns 1/0 depending on whether text only contains ASCII characters or not.

## 5.1 Deprecated Functions

These functions are deprecated and will be removed in a future release.

`set(string[,logic=1])`

**DEPRECATED:** Use CharSet() instead.

Returns a character set for string: a bit encoded version of the characters occurring in string.

If logic is 0, then all characters *not* in string will be in the set.

Unicode input is not supported.

`invset(string)`

**DEPRECATED:** Use CharSet("^...") instead.

Same as `set(string,0)`.

Unicode input is not supported.

`setfind(text,set[,start=0,stop=len(text)])`

**DEPRECATED:** Use CharSet().find() instead.

Find the first occurence of any character from set in `text[start:stop]`. `set` must be a string obtained from `set()`.

Unicode input is not supported.

`setstrip(text,set[,start=0,stop=len(text),mode=0])`

**DEPRECATED:** Use CharSet().strip() instead.

Strip all characters in text[start:stop] appearing in set. mode indicates where to strip (<0: left; =0: left and right; >0: right). set must be a string obtained with `set()`.

Unicode input is not supported.

`setsplit(text,set[,start=0,stop=len(text)])`

**DEPRECATED:** Use CharSet().split() instead.

Split text[start:stop] into substrings using set, omitting the splitting parts and empty substrings. `set` must be a string obtained from `set()`.

Unicode input is not supported.

`setsplitx(text,set[,start=0,stop=len(text)])`

**DEPRECATED:** Use CharSet().splitx() instead.

Split text[start:stop] into substrings using set, so that every second entry consists only of characters in set. `set` must be a string obtained from `set()`.

Unicode input is not supported.

## 5.2 Undocumented Functions

The `TextTools.py` also defines a few other functions, but these are left undocumented since they may disappear in future releases.

# 6. mx.TextTools Constants

The package exports these constants. They are defined in `Constants/Sets`.

Note that Unicode defines many more characters in the following categories. The character sets defined here are restricted to ASCII (and parts of Latin-1) only.

`a2z`

'abcdefghijklmnopqrstuvwxyz'

`A2Z`

'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

`a2z`

'abcdefghijklmnopqrstuvwxyz'

`umlaute`

'äöüß'

`Umlaute`

'ÄÖÜ'

`alpha`

A2Z + a2z

`a2z`

'abcdefghijklmnopqrstuvwxyz'

`german_alpha`

A2Z + a2z + umlaute + Umlaute

`number`

'0123456789'

`alphanumeric`

alpha + number

`white`

' \t\v'

`newline`

'\n\r'

`formfeed`

'\f'

`whitespace`

white + newline + formfeed

`any`

All characters from \000-\377

`*_charset`

All of the above as CharSet() objects.

`*_set`

All of the above as set() compatible character sets.

`tagtable_cache`

This the cache dictionary which is used by the TagTable() compiler to store compiled Tag Table definitions. It has a hard limit of 100 entries, but can also be managed by user routines to lower this limit.

`BOYERMOORE, FASTSEARCH, TRIVIAL`

`TextSearch() algorithm values.`

# 7. Examples of Use

The `Examples/` subdirectory of the package contains a few examples of how tables can be written and used.

Here's a non-trivial example for parsing HTML (well, most of it):

```
from mx.TextTools import *

# Error tag object
error = '*syntax error'

# Character sets
tagname_charset = CharSet(alpha+'\-'+number)
tagattrname_charset = CharSet(alpha+'\-'+number)
tagvalue_charset = CharSet('^"\'> ')
white_charset = CharSet(' \r\n\t')

tagname_charset = CharSet(alpha+'\-'+number)
tagattrname_charset = CharSet(alpha+'\-'+number)
tagvalue_charset = CharSet('^"\'> ')
white_charset = CharSet(' \r\n\t')

# Tag attributes
tagattr = (
       # name
       ('name',AllInCharSet,tagattrname_charset),
       # with value ?
       (None,Is,'=',MatchOk),
       # skip junk
       (None,AllInCharSet,white_charset,+1),
       # unquoted value
       ('value',AllInCharSet,tagvalue_charset,+1,MatchOk),
       # double quoted value
       (None,Is,'"',+5),
         ('value',AllNotIn,'"',+1,+2),
         ('value',Skip,0),
         (None,Is,'"'),
         (None,Jump,To,MatchOk),
       # single quoted value
       (None,Is,'\''),
         ('value',AllNotIn,'\'',+1,+2),
         ('value',Skip,0),
         (None,Is,'\'')
       )

# Tag values
valuetable = (
    # ignore whitespace + '='
    (None,AllInCharSet,CharSet(' \r\n\t='),+1),
    # unquoted value
    ('value',AllInCharSet,tagvalue_charset,+1,MatchOk),
    # double quoted value
    (None,Is,'"',+5),
     ('value',AllNotIn,'"',+1,+2),
     ('value',Skip,0),
     (None,Is,'"'),
     (None,Jump,To,MatchOk),
```

```
    # single quoted value
    (None,Is,'\''),
     ('value',AllNotIn,'\'',+1,+2),
     ('value',Skip,0),
     (None,Is,'\'')
    )

# Parse all attributes of a tag
allattrs = (
    # look for attributes
    (None,AllInCharSet,white_charset,+4),
     (None,Is,'>',+1,MatchOk),
     ('tagattr',Table,tagattr),
     (None,Jump,To,-3),
    (None,Is,'>',+1,MatchOk),
    # handle incorrect attributes
    (error,AllNotIn,'> \r\n\t'),
    (None,Jump,To,-6)
    )

# NOTE: The htmltag tag table assumes that the input text is given
#       in upper case letters (see <XMP> handling).

# Parse an HTML tag
htmltag = (
    (None,Is,'<'),
    # is this a closing tag ?
    ('closetag',Is,'/',+1),
    # a coment ?
    ('comment',Is,'!',+8),
     (None,Word,'--',+4),
     ('text',WordStart,'-->',+1),
     (None,Skip,3),
     (None,Jump,To,MatchOk),
     # a SGML-Tag ?
     ('other',AllNotIn,'>',+1),
      (None,Is,'>'),
     (None,Jump,To,MatchOk),
    # XMP-Tag ?
    ('tagname',Word,'XMP',+5),
     (None,Is,'>'),
     ('text',WordStart,'</XMP>'),
     (None,Skip,len('</XMP>')),
     (None,Jump,To,MatchOk),
    # get the tag name
    ('tagname',AllInCharSet,tagname_charset),
    # look for attributes
    (None,AllInCharSet,white_charset,+4),
     (None,Is,'>',+1,MatchOk),
     ('tagattr',Table,tagattr),
     (None,Jump,To,-3),
     (None,Is,'>',+1,MatchOk),
    # handle incorrect attributes
    (error,AllNotIn,'> \n\r\t'),
    (None,Jump,To,-6)
    )

# Parse HTML tags & text
htmltable = (# HTML-Tag
             ('htmltag',Table,htmltag,+1,+4),
             # not HTML, but still using this syntax: error or
             # inside XMP-tag !
             (error,Is,'<',+3),
              (error,AllNotIn,'>',+1),
```

```
  (error,Is,'>'),
 # normal text
 ('text',AllNotIn,'<',+1),
 # end of file
 ('eof',EOF,Here,-5),
)
```

The above may look a bit like machine code, but it's a very fast implementation of an HTML scanner and runs on all supported platforms.

# 8. Optional Add-Ons for mxTextTools

Mike C. Fletcher has written a Tag Table generator called *SimpleParse*. It works as parser generating front end to the Tagging Engine and converts a EBNF style grammar into a Tag Table directly useable with the `tag()` function.

Andrew Dalke has written a parser generator called *Martel* built upon mxTextTools which takes a regular expression grammer for a format and turns the resultant parsed tree into a set of callback events emulating the XML/SAX API.

# 9. Package Structure

```
[TextTools]
       [Constants]
              Sets.py
              TagTables.py
       Doc/
       [Examples]
              HTML.py
              Loop.py
              Python.py
              RTF.py
              RegExp.py
              Tim.py
              Words.py
              altRTF.py
              pytag.py
       [mxTextTools]
              test.py
       TextTools.py
```

Entries enclosed in brackets are packages (i.e. they are directories that include a `__init__.py` file). Ones with slashes are just ordinary subdirectories that are not accessible via `import`.

The package TextTools imports everything needed from the other components. It is sometimes also handy to do a `from mx.TextTools.Constants.TagTables import *`.

`Examples/` contains a few demos of what the Tag Tables can do.

# 10. Support

eGenix.com is providing commercial support for this package. If you are interested in receiving information about this service please see the *eGenix.com Support Conditions*.

# 11. Copyright & License

## EGENIX.COM PUBLIC LICENSE AGREEMENT

Version 1.1.0

*This license agreement is based on the Python CNRI License Agreement, a widely accepted open-source license.*

### 1. Introduction

This "License Agreement" is between eGenix.com Software, Skills and Services GmbH ("eGenix.com"), having an office at Pastor-Loeh-Str. 48, D-40764 Langenfeld, Germany, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

### 2. License

Subject to the terms and conditions of this eGenix.com Public License Agreement, eGenix.com hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the eGenix.com Public License Agreement is retained in the Software, or in any derivative version of the Software prepared by Licensee.

### 3. NO WARRANTY

eGenix.com is making the Software available to Licensee on an "AS IS" basis. SUBJECT TO ANY STATUTORY WARRANTIES WHICH CAN NOT BE EXCLUDED, EGENIX.COM MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, EGENIX.COM MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

### 4. LIMITATION OF LIABILITY

EGENIX.COM SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) AS A RESULT OF USING, MODIFYING OR

DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE EXCLUSION OR LIMITATION MAY NOT APPLY TO LICENSEE.

## 5. Termination

This License Agreement will automatically terminate upon a material breach of its terms and conditions.

## 6. Third Party Rights

Any software or documentation in source or binary form provided along with the Software that is associated with a separate license agreement is licensed to Licensee under the terms of that license agreement. This License Agreement does not apply to those portions of the Software. Copies of the third party licenses are included in the Software Distribution.

## 7. General

Nothing in this License Agreement affects any statutory rights of consumers that cannot be waived or limited by contract.

Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between eGenix.com and Licensee.

If any provision of this License Agreement shall be unlawful, void, or for any reason unenforceable, such provision shall be modified to the extent necessary to render it enforceable without losing its intent, or, if no such modification is possible, be severed from this License Agreement and shall not affect the validity and enforceability of the remaining provisions of this License Agreement.

This License Agreement shall be governed by and interpreted in all respects by the law of Germany, excluding conflict of law provisions. It shall not be governed by the United Nations Convention on Contracts for International Sale of Goods.

This License Agreement does not grant permission to use eGenix.com trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party.

The controlling language of this License Agreement is English. If Licensee has received a translation into another language, it has been provided for Licensee's convenience only.

## 8. Agreement

By downloading, copying, installing or otherwise using the Software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

For question regarding this License Agreement, please write to:

eGenix.com Software, Skills and Services GmbH

Pastor-Loeh-Str. 48

D-40764 Langenfeld

Germany