PyMaLa
Python Markup Language to Flat File Converter

This tool converts XML and, to some extend, HTML files to tab-delimited flat files. It requires a script file containing the header definition of the tab-delimited file and the path information to find the data within the XML document. In contrast to many other tools it does not parse the complete XML structure but only the necessary elements according to the path definitions in the script file saving time and memory space. PyMaLa can convert single entity document files and massive multi-entity documents. If the entity order in the flat file is of no concern, PyMala can be operated in multiprocessing mode.

How to use PyMaLa

To get an overview of the options just call PyMaLa without parameters `py pymala.py`:

```
PyMaLa - python markup-language to flat file converter
version 2024.02.08
pymala.py <script-file> [options ...]
options:
-input <input_template> : declares the document files using placeholders (* = any no of chars, ? = single
char)
                          i.e.: -inp data*\doc_*.xml
                          browse through directories starting with "data" selecting xml files starting
with "doc_"
-inp <input_template>   : shortcut for -input
-output <output_file>   : target file for the tab-delimited data
-out <output_file>      : shortcut for -output
-root <root>            : root tag definintion identifying an entity (only required for multi-entity
files)
-mp <processes>         : activates multiprocessing by assigning a number of processes to the task
                          if the no is negative or zero, it declares the CPUs not used for the task
                          file access may become a bottleneck for large numbers of assigned processes
-chunk <size>           : separates larger multi-entity files into chunks of <size> MB to enable
multiprocessing
                          every chunk is considered a separate file to be distributed to a process
                          requires a distinct root definition and should not be applied for single-entity
files
-encoding <enc>         : declares the encoding of the document files, e.g. latin1, ansi, utf-8 (default)
-info                   : concludes with some statistics (requires "true" or "false" as setting in the
script)
                          docs = number of documents or chunks, pyml = number of pymala entities,
                          rows = number of lines in output, proc = number of processes,
                          clog = congestion of output process (it cannot keep pace with parsing if close
to 100%)
                          time = run time for parsing without initialization
options override corresponding settings in the script file
a setting is not preceded by a minus and its parameter is separated by a colon, i.e. info: true
```

The script language will be described in a latter section. All **options** listed can also declared in the script file as settings. A **setting** does not have a preceding minus sign and is separated from the parameter by a colon. The command line options will always override the corresponding script settings. Usually, settings should be defined at the beginning of the script file, which is always the first parameter of a PyMaLa call. This section explains the settings:

`input: input_template` (or `inp`) defines the the path to the input files containing the XML entities. If the template contains placeholders all files matching the template will be considered. A template without placeholders always

designates only one specific file. The ***** placeholder represents any number of characters (including zero) while the **?** placeholder represents a single character. Make sure that all files retrieved by the template have the same XML format. You can also specify placeholders within the path name to browse through multiple directories in search of matching files. **The base directory for the input template is always the script directory.**

`output: output_file` (or `out`) declares the output file. It will receive the data retrieved from the XML entities. Columns will be separated by *tab* characters. The first line contains the column names (header). If omitted, the output will be redirected to standard output. **The base directory for the output file is always the script directory.** `output: output_file` (or `out`) declares the output file. It will receive the data retrieved from the XML entities. Columns will be separated by *tab* characters. The first line contains the column names (header). If omitted, the output will be redirected to standard output. **The base directory for the output file is always the script directory.**

`root: root_tags` declares the main root tag separating XML entities within a **multi-entity file**. **Do not use root for single-entity documents.** You can specify multiple root tags if different entities match the information referred in the script. Multiple tag definitions are separated by a pipe **|**. A tag definition omits the enclosing lesser-than and larger than-signs (<, >) and may contain ***?** placeholders. Roots do not have to be unique for an entity as long as they are on different hierarchical levels. A root tag only has to be unique within an entity when multiprocessing is applied because a process can jump into the middle of an entity and therefore needs a distinct start tag to find the beginning of the next entity. The **root** setting is rarely used as command line option. See the script section for more information about tag definitions.

`mp: no_of_processes` activates multiprocessing if the number of *processes* is larger than one. In multiprocessing mode the original order of the entities in the output file cannot be maintained. Multi-processing divides the work by distributing the documents retrieved by the **input** template over the number of specified processes. You can use the **chunk** setting to split up larger documents into multiple virtual files to enable multiprocessing even for those monolithic, multi-entity files. Assigning more processes than available cores (CPUs) can have detrimental effects. There may be diminishing returns of increasing this number due to file access bottlenecks. There is always only one output process to prevent file access conflicts but this may lead to a race condition between parsing and output (clogging). A virtual chunk consists of the file name, a start and a stop position within that file.

`chunk: size_in_MB` separates large multi-entity files into smaller virtual files, each having roughly the specified size in MB. This setting is only required in conjunction with the **mp** setting to enable multiprocessing for large multi-entity files. The size should be large enough to accomodate multiple entites. **Do not use chunk if every document represents only one entity.**

`endcoding: file_encoding` defines the encoding for all files retrieved by the **input** template. Typical encodings are **ansi**, **latin1** or **utf-8**, which is the default setting. The output file will have the same encoding.

`info: true_or_false` switches between showing some final statistics (*true*) or hiding them (*false* or not using the setting). As option, you only have to state `-info`. Following statistics are not shown:

- docs: the number of documents respectively virtual chunks retrieved by the input template.
- pyml: number of PyMaLa entities (encased by the highest level of the XML file or by the **root** tag(s)).
- rows: number of data rows in the **output** file (excl. the header).
- proc: number of used processes, which may be lower than the **mp** setting if there are not enough **docs**.
- clog: average clogging of the output queue. If this percentage is close to 100%, reduce the **mp** setting.
- time: run time for parsing only (without retrieval according to the **input** template and **chunk** splitting).

If you have a larger quantity of data to process, it is recommended to execute PyMaLa on an excerpt to find efficient settings that do not clog the output queue or overburden the file system with too many concurrent accesses in multiprocessing mode. Of course, you can forego **mp** altogether at the expense of processing time to maintain the original entity order.

Examples

In general, settings that are specific to the data should only be defined in the script file. This is the case for the **root** setting. If **input** or **output** should be a option or a setting depends on the task. If the script relates to a class of documents at different locations, input and output should be flexible options. If the task is location bound, fixed script settings are more appropriate. Remember, options always override settings.

```
py pymala example.mala -inp source_*\delivery_*.xml -out data.txt -mp 8
```

Uses the script file example.mala (.mala is the dispensable file extension for pymala scripts, but any other will do). The input files are in directories matching the template source_* while documents match delivery_*.xml. The data will be collected in the file data.txt. As the delivery_*.xml files are single-entity documents no root is required. The parsing will be condcuted by 8 parallel processes.

```
py pymala firms.mala -inp "data\listed-companies.xml" -out firms.txt -root "company-data" -mp 8 -chunk 64
```

The script file firma.mala is used to parse the multi-entity file listed-companies.xml into the firms.txt output file. Because the input file comprises multiple companies, we have to specify a root tag enclosing an entity, i.e. <company-data id=1234>...</company-data>. This should have been a setting declared in the script file and not as a command line option. The file seems to be huge and therefore multiprocessing is especially beneficial. Separating the monolithic file into many smaller virtual chunks of 64 MB enables the distribution of the load.

PyMaLa Script

Besides the before-mentioned options, the PyMaLa script language consists of two major sections. The header and the paths. The header declares the column header of the tab-delimited output file. A path describes the way through the tag hierarchy of the XML tree structure to an end tag enclosing actual data or containing properties. Every path has a name to be referenced in the header. A path name can represent a single value but also a vector of multiple values if multiple end tags at the same hierarchy level are detected. PyMaLa will use the path declarations to establish consistency among paths at different hierarchy levels. For a better understanding of the concepts we will refer to the following XML file named **candyshop.xml** throughout the documentation:

```
<shop>
    <name>Blueberry</name>
    <address>Candy Lane, Sugarhill</address>
    <clientlist>
        <client id=1>
            <name>Paul</name>
            <birthday year=1995 month=5 day=21/>
            <likes>sour sweets</likes>
        </client>
        <client id=2>
            <name>Peter</name>
            <misc>undecided</misc>
        </client>
        <client id=3>
            <name>Mary</name>
            <likes>licorice</likes>
        </client>
        <customer id=4>
            <name>John</name>
            <likes>candy bars</likes>
        </customer>
        <client id=5>
            <name>Frank</name>
            <birthday year=1965 month=11 day=7/>
            <likes>cotton candy</likes>
            <likes>candy cane</likes>
            <likes>chocolate</likes>
        </client>
        <name>welcome</name>
    </clientlist>
    <clientlist>
        <name>premium</name>
    </clientlist>
    <clientlist>
        <name>unwelcome</name>
        <customer id=77>
            <name>Gandalf</name>
            <likes>muffins</likes>
            <likes>donuts</likes>
            <misc>looks like slim santa</misc>
            <misc>always requests samples</misc>
            <misc>lame magic tricks</misc>
        </customer>
        <client id=79>
            <name>Saruman</name>
            <likes>smurfs</likes>
            <likes>sour bats</likes>
            <likes>sugar crystal balls</likes>
            <misc>trouble maker</misc>
            <misc>saurons buddy</misc>
        </client>
    </clientlist>
    <clientlist>
        <name>banned</name>
        <client id=666>
            <name>Sauron</name>
            <likes>jelly eyeballs</likes>
            <misc>shop lifter?</misc>
        </client>
    </clientlist>
</shop>
```

The XML structure of this file is not the most consistent one and your data will most likely be better organized. Still, it provides the necessary complexity to walk you through all the elements and features of the PyMaLa script language. It describes the clients respectively customers of a candyshop with their names, preferences and employees notes. Clients are organized into different lists assigning them a popularity level according to the shopkeeper.

Paths

We start with a simple extraction script for all the clients along with their assigned list type:

```
input: candyshop.xml
shop = shop.name
address = shop.address
type = shop.clientlist.name
id = shop.clientlist.client:id
name = shop.clientlist.client.name
```

This script file has the name candyshop.mala and resides in the same directory as the candyshop.xml file. Therefore, no further directory specifications are required. The script is evoked with `py pymala.py candyshop` (the extension ".mala" can be omitted). A path definition leading to the data has always a name followed by an equal sign. The path is depicted as the sequence of XML tags separated by dots. If the data is a property of a tag, it can be referred by a colon and its name (see "id"). The result is already shown as formatted table for the sake of clarity:

shop	address	type	id	name
Blueberry	Candy Lane, Sugarhill	welcome	1	Paul
Blueberry	Candy Lane, Sugarhill	welcome	2	Peter
Blueberry	Candy Lane, Sugarhill	welcome	3	Mary
Blueberry	Candy Lane, Sugarhill	welcome	5	Frank
Blueberry	Candy Lane, Sugarhill	premium		
Blueberry	Candy Lane, Sugarhill	unwelcome	79	Saruman
Blueberry	Candy Lane, Sugarhill	banned	666	Sauron

By default, the path names constitute the columns names in the output file in the order of appearance in the script. We will see in the next section that we can change this behaviour with the definition of a **header**. The client "John" is missing in the output table because his tag is "customer" and not "client", a mistake or some hidden signaling. We can introduce alternative tags with a pipe **|** separator. During traversing the path to the data, PyMaLa considers all alternative tags:

```
input: candyshop.xml
shop = shop.name
address = shop.address
type = shop.clientlist.name
id = shop.clientlist.client|customer:id
name = shop.clientlist.client|customer.name
```

We get the complete table including the hidden customer:

shop	address	type	id	name
Blueberry	Candy Lane, Sugarhill	welcome	1	Paul
Blueberry	Candy Lane, Sugarhill	welcome	2	Peter
Blueberry	Candy Lane, Sugarhill	welcome	3	Mary
Blueberry	Candy Lane, Sugarhill	welcome	4	John
Blueberry	Candy Lane, Sugarhill	welcome	5	Frank
Blueberry	Candy Lane, Sugarhill	premium		
Blueberry	Candy Lane, Sugarhill	unwelcome	79	Saruman
Blueberry	Candy Lane, Sugarhill	banned	666	Sauron

We do not have to repeat the full path for every definition. A path without a name is considered a root. All following path declarations start directly after the last root definition including the differentiation between tag data or properties (see ":id"). Roots can be extended if they are preceded with a dot. A root without a preceding dot resets the root and starts a new one. Even though the equal sign is optional for a root definition, a single equal sign always resets the root. Alternatively, you can use a single dot **.** or the colon **:** sign. Later, we will learn that PyMaLa has a complete set of alternate symbols to cater for different tastes. This, for the sake of completeness, quite convoluted script shows all root variants:

```
input: candyshop.xml
shop
shop = name
address = address
.clientlist
type = name
= .client|customer
id = :id
.
name = shop.clientlist.client|customer.name
```

Besides alternative tags declared with pipe **|** characters, the tag definitions also supports the placeholder ***** for any number of characters (incl. zero) and **?** for exactly one character. These placeholders can be used if tags are ambiguous but share identical parts, i.e. `name = shop.*list.client|customer.name` would accept any kind of tag that ends with the word "list" to also capture "customerlist". If a tag definition consists of only an asteryx ***** any number of tags will be skipped until the next tag in the path is found. This proves useful when the XML structure is very deep or variable and the tag in question is distinct. If only one tag level should be skipped, regardless of the tag definition, use the combination ***?**. Placeholders and pipes can also be used for property definitions. If multiple properties match a definition, they will be separated by a pipe **|**. The following example shows the pitfalls of the too liberal use of placeholders:

```
input: candyshop.xml
shop = shop.name
address = shop.address
type = *.*list.name
id = *.customer|client:id
name = *.customer|client.name
birthday = *.customer|client.birthday:*
```

Because the hierarchical structure of the XML document is not reflected in the path descriptors, PyMaLa is not able to associate the clients with their assigned client list:

shop	address	type	id	name	birthday
Blueberry	Candy Lane, Sugarhill	welcome	1	Paul	1995|5|21
Blueberry	Candy Lane, Sugarhill	premium	2	Peter	
Blueberry	Candy Lane, Sugarhill	unwelcome	3	Mary	
Blueberry	Candy Lane, Sugarhill	banned	4	John	
Blueberry	Candy Lane, Sugarhill	banned	5	Frank	1965|11|7
Blueberry	Candy Lane, Sugarhill	banned	77	Gandalf	
Blueberry	Candy Lane, Sugarhill	banned	79	Saruman	
Blueberry	Candy Lane, Sugarhill	banned	666	Sauron	

This output shows that PyMaLa always tries to generate a rectangular table by copying the values to fill the gaps. PyMaLa creates a tree structure on the base of the path descriptors. During parsing, it traverses the branches to the data leafs where it collects the data. To reach all leafs, it has to return to branches not traversed yet. While doing so, it always maintains a rectangular table structure of the data by copying the last entries of columns that are shorter than the longest column. If this tree is flat and has no hierarchical structure this procedure falls flat.

A more sensible way of using placeholders would be to use them to navigate to a root like "clientlist" and specify the complete path structure from this starting point for all subordinate elements:

```
input: candyshop.xml
shop = shop.name
address = shop.address
*.*list
type = name
id = customer|client:id
name = customer|client.name
```

This script maintains the hierarchical structure of the XML document and generates the correct output. You do not have to use roots but they help you to structure the script. The use of a placeholder in the root is only for demonstration purposes. We know that there are no other lists than "clientlist". Keep in mind that placeholder will slow down the parsing, albeit only slightly.

Finally, in the unlikely case that a dot "." is part of a tag definition, which would collide with the usage as path separator, the greater-than > sign can be used as a substitute. This also enforces the replacement of the : sign by the lesser-than symbol < to declare a property definition. This also extends to root declarations. Accordingly, a root can be reset by a single > or <. Within a path declaration you cannot mix the two styles:

```
input: candyshop.xml
shop = shop>name
address = shop>address
*>*list
type = name
id = customer>client<id
# resetting the root
<
name = *.clientlist.customer|client.name
```

Which separator you use in general is just a matter of taste. Finally, you can make comments by preceding them with a hash # sign at the beginning of the line.

Headers

By default, path names become columns names of the header line in the output file in the order of appearance in the script. To have more control over the final table, you can specify an explicit header to rearrange the order of the columns, rename columns and even to combine multiple path names into one column. Another purpose of the header is to differentiate between key fields and data fields. The header also defines if an entity can have multiple rows or if only one row per entity will be reported.

A header declaration always starts with the keyword **header:** followed by column declarations separated by comma. You can split the header over multiple declarations starting with this keyword. The header has to be defined before the first path respectively root declaration. Any path name not used in the header will be appended as new column. PyMaLa automatically avoids name conflicts.

Now we can tackle the additional data in the example XML structure in a proper way. First, let us extract the birthday dates that are stored as properties for some clients:

```
input: candyshop.xml
header: id, client = name, type
header: birthday = year "." month '.' day
shop = shop.name
address = shop.address
type = *.clientlist.name
id = *.clientlist.customer|client:id
year = *.clientlist.customer|client.birthday:year
month = *.clientlist.customer|client.birthday:month
day = *.clientlist.customer|client.birthday:day
name = *.clientlist.customer|client.name
```

Take note that you can split the header over multiple lines, always starting with the keyword **header:**, to prevent unwieldy header lines. In the header declaration of this example, we put the clients first by assigning the column name "client" for the client name. The birthday is assembled from the properties of the tag "birthday". A column definition may have a column name separated by an equal sign from the field template. A field template can be composed of any combination of path names and string literals, which can be enclosed by pairs of single of double quotes. Columns may even comprise of constant literals only. This table shows the output with the rearranged header:

id	client	type	birthday	shop	address
1	Paul	welcome	1995.5.21	Blueberry	Candy Lane, Sugarhill
2	Peter	welcome		Blueberry	Candy Lane, Sugarhill
3	Mary	welcome		Blueberry	Candy Lane, Sugarhill
4	John	welcome		Blueberry	Candy Lane, Sugarhill
5	Frank	welcome	1965.11.7	Blueberry	Candy Lane, Sugarhill
		premium		Blueberry	Candy Lane, Sugarhill
77	Gandalf	unwelcome		Blueberry	Candy Lane, Sugarhill
79	Saruman	unwelcome		Blueberry	Candy Lane, Sugarhill
666	Sauron	banned		Blueberry	Candy Lane, Sugarhill

The path names not mentioned in the header have been appended to the right side. This may lead to unexpected name conflicts. The following example shows how PyMaLa resolves those conflicts:

```
input: candyshop.xml
header: id, name = client, type, birthday = year "." month '.' day
name = shop.name
address = shop.address
type = *.clientlist.name
id = *.clientlist.customer|client:id
year = *.clientlist.customer|client.birthday:year
month = *.clientlist.customer|client.birthday:month
day = *.clientlist.customer|client.birthday:day
client = *.clientlist.customer|client.name
```

Because we have changed the path names and column names to create a conflict between the shop and the client name, PyMaLa automatically changes column names to create an unambiguous header:

id	name	type	birthday	name_1	address
1	Paul	welcome	1995.5.21	Blueberry	Candy Lane, Sugarhill
2	Peter	welcome		Blueberry	Candy Lane, Sugarhill
...	...	...	...	...	...

In many cases, we encounter incomplete data in the XML structure, for example the client list with the name "premium" that does not have any associated clients. To supress the report of this rows with insufficient information, the header allows declare specific path names as key fields. If a key field is missing, even as part of an composed column, the whole row will be suppressed as an essential part is missing. This is also the case, if only the key field(s) exist(s) and all other non-key fields are empty. It is quite common in XML data deliveries that the general identfiers are always available while the specific information is missing. It is a preferable behaviour to not report those. A key field is declared by putting an exclamation mark **!** infront of the path name.

There are different ways to suppress client lists without clients. First we have to make sure that there is no information retrieved that is not directly associated with a client like the shop name and address. Then, we can report only rows where we have a client id by declaring it as key field:

```
input: candyshop.xml
header: !id, client, type, birthday = year "." month "." day
type = *.clientlist.name
id = *.clientlist.customer|client:id
year = *.clientlist.customer|client.birthday:year
month = *.clientlist.customer|client.birthday:month
day = *.clientlist.customer|client.birthday:day
client = *.clientlist.customer|client.name
```

Lo and behold, the irritatingly empty client list "premium" has disappeared because the field "id" is empty.

id	client	type	birthday
1	Paul	welcome	1995.5.21
2	Peter	welcome	
3	Mary	welcome	
4	John	welcome	
5	Frank	welcome	1965.11.7
77	Gandalf	unwelcome	
79	Saruman	unwelcome	
666	Sauron	banned	

We would have achieved the same by declaring the "type" field as key field. Because all other fields are empty for the "premium" type, its row will be suppressed. This only works if the shop fields are not part of the data because they constitute information associated with the client list.

By declaring the path names "year", "month" and "day" as keys in the header, we can filter all clients with a complete birthday date: `header: id, client, type, birthday = !year "." !month "." !day`

id	client	type	birthday
1	Paul	welcome	1995.5.21
5	Frank	welcome	1965.11.7

If we add the preferences and miscellaneous remarks to the data, the clients with multiple mentions in those fields have multiple rows to accommodate the information. For a change, we use roots to shorten the script:

```
input: candyshop.xml
header: !id, client
*.clientlist
type = name
.customer|client
id = :id
client = name
likes = likes
misc = misc
```

Values are repeated to maintain a rectangular table shape, which leads to false associations between the "like" fields and the "misc" fields for "Gandalf" and "Saruman":

id	client	type	likes	misc
1	Paul	welcome	sour sweets	
2	Peter	welcome		undecided
3	Mary	welcome	licorice	
4	John	welcome	candy bars	
5	Frank	welcome	cotton candy	
5	Frank	welcome	candy cane	
5	Frank	welcome	chocolate	
77	Gandalf	unwelcome	muffins	looks like slim santa
77	Gandalf	unwelcome	donuts	always requests samples
77	Gandalf	unwelcome	donuts	lame magic tricks
79	Saruman	unwelcome	smurfs	trouble maker
79	Saruman	unwelcome	sour bats	saurons buddy
79	Saruman	unwelcome	sugar crystal balls	saurons buddy
666	Sauron	banned	jelly eyeballs	shop lifter?

The issue is that Pymala cannot differentiate between the necessary repetition of the "id", "client" name and "type" fields and the inconsistent repetition of the "likes" and "misc" fields. Given the XML structure, there is no explicit indication for the auxiliary nature of those fields. We can mitigate this issue by declaring the respective fields as lists that may contain multiple uncorrelated values. This can be done by attaching the list indicator **.0** to the tag names:

```
input: candyshop.xml
header: !id, client, type, likes.0, misc.0
*.clientlist
type = name
.customer|client
id = :id
client = name
likes = likes
misc = misc
```

This solves the redundancy among those fields but not the inconsistent association of the data, i.e. "smurfs" and "trouble maker" are associated with the client but not with each other:

id	client	type	likes	misc
1	Paul	welcome	sour sweets	
2	Peter	welcome		undecided
3	Mary	welcome	licorice	
4	John	welcome	candy bars	
5	Frank	welcome	cotton candy	
5	Frank	welcome	candy cane	
5	Frank	welcome	chocolate	
77	Gandalf	unwelcome	muffins	looks like slim santa
77	Gandalf	unwelcome	donuts	always requests samples
77	Gandalf	unwelcome		lame magic tricks
79	Saruman	unwelcome	smurfs	trouble maker
79	Saruman	unwelcome	sour bats	saurons buddy
79	Saruman	unwelcome	sugar crystal balls	
666	Sauron	banned	jelly eyeballs	shop lifter?

You can represent a single 1:n relationship within one file, which will cause redundancies (repeated values) but at least no inconsistencies. It is impossible to achieve that with more than one of those relationships within one output file. It would be better to report each of those fields in a separate file using specific scripts. You can prevent the reporting of multiple lines for an entity by enforcing single-line output by attaching at least one list indicator greater than zero to a tag name, i.e. **likes.1**. An entity can be defined by the input document, the **root** setting/option in case of multi-entity documents or by the declaration of key fields (path names with an exclamation mark prefix). The latter is required if a physical document or a root extraction still represents multiple entities as it is the case for our example. We can enforce a single-line per entity output by simply attaching a **.1** list indicator to any path name in the header, i.e. **!id.1**:

```
input: candyshop.xml
header: !id.1, client, type, likes, misc
*.clientlist
type = name
.customer|client
id = :id
client = name
likes = likes
misc = misc
```

Every client occurs only once in the output file no matter to which path name the list indicator was attached. Only the first element of lists will be reported:

id	client	type	likes	misc
1	Paul	welcome	sour sweets	
2	Peter	welcome		undecided
3	Mary	welcome	licorice	
4	John	welcome	candy bars	
5	Frank	welcome	cotton candy	
77	Gandalf	unwelcome	muffins	looks like slim santa
79	Saruman	unwelcome	smurfs	trouble maker
666	Sauron	banned	jelly eyeballs	shop lifter?

If we want to report more than the first element, we can just declare more output columns with list indicators. PyMaLa will automatically enumerate the column names accordingly:

```
input: candyshop.xml
header: !id, client, type, likes.1, likes.2, likes.3, misc.1, misc.2, misc.3
*.clientlist
type = name
.customer|client
id = :id
client = name
likes = likes
misc = misc
```

Of course, you can specify specific names for the different columns when you are not content with the way PyMaLa does it. This format can report a fixed number of list elements, which is sufficient for limited lists.

id	client	type	likes_1	likes_2	likes_3	misc_1	misc_2	misc_3
1	Paul	welcome	sour sweets					
2	Peter	welcome				undecided		
3	Mary	welcome	licorice					
4	John	welcome	candy bars					
5	Frank	welcome	cotton candy	candy cane	chocolate			
77	Gandalf	unwelcome	muffins	donuts		looks like slim santa	always requests samples	lame magic tricks
79	Saruman	unwelcome	smurfs	sour bats	sugar crystal balls	trouble maker	saurons buddy	
666	Sauron	banned	jelly eyeballs			shop lifter?		

Even though the redundancy seems averted it still exists in the shape of a sparsely populated table. Keep that in mind if you are using list indicators instead of a relational setup of multiple tables based on specific PyMaLa scripts. A horizontal structure is more convenient for single use tables while multiple relational tables are better suited for database environments.

Syntax

```
header: [column_name = ]field_template[,[column_name = ]field_template ...]
field_template: {string | [!]path_name[.pos]}[string | [!]path_name[.pos] ...]
string: {"text_without_double_quotes" | 'text_without_single_quotes'}

{path_name = path_declaration[{:|<}template] | [=] path_decaration |=|.|:|<|>|#}
path_declaration: tag_descriptor[{.|>}tag_descriptor ...]
tag_descriptor: {template[|template ...] |*|*?}
template: any_char|*|?[any_char|*|? ...]
```