# CONTENTS:

1 Available formats .......................................................... 3
2 Conversion stages ............................................................ 5
3 Converters: Parsers + Builders ........................................... 7
4 Usage ........................................................................... 9
   4.1 Benker .................................................................. 10
   4.2 Tutorials ................................................................. 12
   4.3 API ...................................................................... 33
   4.4 Changelog ............................................................... 62
5 Indices and tables ............................................................. 67

Python Module Index ............................................................... 69
Index .................................................................................. 71
The Benker library can be used to convert tables from one format to another.

Yes, it only converts the tables, not the whole document, but it tries to do it well. The document itself is not changed, and the paragraphs inside the cells, neither. It’s your responsibility to do this part of the work.
# AVAILABLE FORMATS

The Benker library works on XML documents. Currently, it can handle:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Official website</th>
<th>Wikipedia page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OOXML (Office Open XML)</strong></td>
<td>An XML-based format for office documents, including word processing documents, spreadsheets, presentations, as well as charts, diagrams, shapes, and other graphical material. This is the XML format used by Microsoft Word documents: <code>.docx</code>.</td>
<td><a href="http://officeopenxml.com/">http://officeopenxml.com/</a></td>
<td><a href="https://en.wikipedia.org/wiki/Office_Open_XML">https://en.wikipedia.org/wiki/Office_Open_XML</a></td>
</tr>
<tr>
<td><strong>CALS (Continuous Acquisition and Life-cycle Support)</strong></td>
<td>A table model standard for representing tables in SGML/XML. Developed as part of the CALS Department of Defence initiative. The DTD of the CALS table model is available in the OASIS (Organization for the Advancement of Structured Information Standards) web site.</td>
<td><a href="https://www.oasis-open.org/specs/tablemodels.php">https://www.oasis-open.org/specs/tablemodels.php</a></td>
<td><a href="https://en.wikipedia.org/wiki/CALS_Table_Model">https://en.wikipedia.org/wiki/CALS_Table_Model</a></td>
</tr>
<tr>
<td><strong>FORMEX (Formalized Exchange of Electronic Publications)</strong></td>
<td>Describes the format for the exchange of data between the Publication Office and its contractors. In particular, it defines the logical markup for documents which are published in the different series of the Official Journal of the European Union. Formex v4 is based on the international standard XML.</td>
<td><a href="https://op.europa.eu/en/web/eu-vocabularies/formex/">https://op.europa.eu/en/web/eu-vocabularies/formex/</a></td>
<td></td>
</tr>
</tbody>
</table>
To convert a document, Benker uses several stages:

1. Parse the source document and construct a nodes tree,
2. Search for table elements and construct the table objects,
3. Build the target nodes tree by replacing table nodes,
4. Serialise the target document.
The decoupling between parsing, building and final serialization allows a simplified and modular implementation. This decoupling also allows to multiply the combinations: it is easy to change a builder to another one, and to develop its own parser...

The advantage of this approach is that we avoid having a specific document conversion for each format pair (input, output). Instead, you can build a converter by choosing a parser and a builder, as you assemble the pieces of a puzzle.

The following table show you the available converters which groups parser and builders by pairs.
Table 1: Available converters

<table>
<thead>
<tr>
<th></th>
<th>OOXXML</th>
<th>HTML</th>
<th>CALS</th>
<th>Formex4</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOXXML</td>
<td>–</td>
<td>(unavailable)</td>
<td>convert_ooxml2cals()</td>
<td>convert_ooxml2formex4()</td>
</tr>
<tr>
<td>HTML</td>
<td>(unavailable)</td>
<td>–</td>
<td>(unavailable)</td>
<td>(unavailable)</td>
</tr>
<tr>
<td>CALS</td>
<td>(unavailable)</td>
<td>(unavailable)</td>
<td>–</td>
<td>(unavailable)</td>
</tr>
<tr>
<td>Formex4</td>
<td>(unavailable)</td>
<td>(unavailable)</td>
<td>(unavailable)</td>
<td>–</td>
</tr>
</tbody>
</table>

You can create your own converter by inheriting the available base classes:

- **BaseConverter**: inherit this class to create your own converter. Set your own parser class to the `parser_cls` class attribute, and your own builder class to the `builder_cls` class attribute.

- **BaseParser**: inherit this class to create your own parser. The method `transform_tables()` is an abstract method, so you need to implement it in your subclass: it must call the method `generate_table_tree()` each time a table node is found and converted to a `Table` object.

- **BaseBuilder**: inherit this class to create your own builder. The method `generate_table_tree()` is an abstract method, so you need to implement it in your subclass: it must convert the `Table` object into a target XML node (the resulting table format). You can also implement the method `finalize_tree()` to do any post-processing to the resulting XML tree.

**Hint**: Contribution is welcome!
For example, to convert the tables of a .docx document to Formex4 format, you can process as follow:

```python
import os
import zipfile

from benker.converters.ooxml2formex4 import convert_ooxml2formex4

# - Unzip the `.docx` in a temporary directory
src_zip = "/path/to/demo.docx"
tmp_dir = "/path/to/tmp/dir/"
with zipfile.ZipFile(src_zip) as zf:
    zf.extractall(tmp_dir)

# - Source paths
src_xml = os.path.join(tmp_dir, "word/document.xml")
styless_xml = os.path.join(tmp_dir, "word/styles.xml")

# - Destination path
dst_xml = "/path/to/demo.xml"

# - Create some options and convert tables
options = {
    'encoding': 'utf-8',
    'styles_path': styles_xml,
}
convert_ooxml2formex4(src_xml, dst_xml, **options)
```

This code produces a table like that:

```xml
<TBL COLS="7" NO.SEQ="0001">
<CORPUS>
<ROW>
   <CELL COL="1" ROWSPAN="2">
      <w:p><w:t>A</w:t></w:p>
   </CELL>
   <CELL COL="2" COLSPAN="2">
      <w:p><w:t>B</w:t></w:p>
   </CELL>
   <CELL COL="4">
      <IE/>
   </CELL>
</ROW>
</CORPUS>
```

(continues on next page)
The content of the cells still contains OOXML fragments. It’s your own responsibility to convert them to the target format.

### 4.1 Benker

Organize your data in a spatial grid system for CALS, HTML, Formex4, Office Open XML tables conversion

#### 4.1.1 Overview

To convert the tables of a `.docx` document to **CALS** format, you can process as follow:

```python
import os
import zipfile

from benker.converters.ooxml2cals import convert_ooxml2cals

# - Unzip the `.docx` in a temporary directory
src_zip = "/path/to/demo.docx"
tmp_dir = "/path/to/tmp/dir/
with zipfile.ZipFile(src_zip) as zf:
  zf.extractall(tmp_dir)

# - Source paths
src_xml = os.path.join(tmp_dir, "word/document.xml")
styles_xml = os.path.join(tmp_dir, "word/styles.xml")

# - Destination path
dst_xml = "/path/to/demo.xml"
```

(continues on next page)
# - Create some options and convert tables
options = {
    'encoding': 'utf-8',
    'styles_path': styles_xml,
    'width_unit': 'mm',
    'table_in_tgroup': True,
}
convert_ooxml2cals(src_xml, dst_xml, **options)

## 4.1.2 Installation

To install this library, you can create and activate a virtualenv, and run:

```
pip install benker
```

### Requirements

This library uses lxml library and is tested with the versions 3.8 and 4.*x*.

The following table shows the compatibility between different combinations of Python and lxml versions:

<table>
<thead>
<tr>
<th>Py</th>
<th>lxml</th>
<th>3.8</th>
<th>4.0</th>
<th>4.1</th>
<th>4.2</th>
<th>4.3</th>
<th>4.4</th>
<th>4.5</th>
<th>4.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.4</td>
<td>✓!</td>
<td>✓!</td>
<td>✓!</td>
<td>✓!</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3.5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.7</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.8</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.9</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- ✓ lxml is available for this version and unit tests succeed.
- ✓! installation succeed using “attrs < 21.1”.
- × lxml is not available for this version of Python.

### Usage in your library/application

You can use this library in your own library/application.

To do so, add this library in your `setup.py` in your project requirements:

```python
setup(
    name="YourApp",
    install_requires=['benker'],
    ...
)
```

To install the dependencies, activate your virtualenv and run:
pip install -e .

And enjoy!

4.1.3 Licence

This library is distributed according to the MIT licence.
Users have legal right to download, modify, or distribute the library.

4.1.4 Authors

Benker was written by Laurent LAPORTE.

4.2 Tutorials

4.2.1 Size

Description

A Size is a tuple with width, height coordinates. It represents the width and the height of a cell in a grid.

```python
>>> from benker.size import Size
>>> size = Size(4, 5)
```

The representation of a size is “(width x height)”:

```python
>>> print(size)
(4 x 5)
```

Operations

You can change the size of a Size by adding, subtracting or multiplying values.

You can add two sizes, a size and a tuple (x, y), a size and a single quantity (integer):

```python
>>> Size(2, 3) + Size(3, 4)
Size(width=5, height=7)
>>> Size(2, 3) + (3, 4)
Size(width=5, height=7)
>>> Size(2, 3) + 1
Size(width=3, height=4)
```

You can subtract two sizes, a size and a tuple (x, y), a size and a single quantity (integer):
You can multiply a size by an integer. This last ability is useful to reverse a size:

```python
>>> Size(3, 4) * 3
Size(width=9, height=12)

>>> Size(3, 4) * -1
Size(width=-3, height=-4)
```

You can also negate a size:

```python
>>> -Size(3, 5)
Size(width=-3, height=-5)

>>> +Size(3, 5)
Size(width=3, height=5)
```

All this operations are useful to do mathematical transformation with `Coord — Operations`, for instance, a translation of a coord is done by adding a coord and a size.

### 4.2.2 Coord

**Description**

A `Coord` is a tuple with `x`, `y` coordinates. It represents the top-left origin of a cell in a grid.

```python
>>> from benker.coord import Coord

>>> coord = Coord(4, 5)
```

We use the Excel convention to represent a `Coord`: columns are represented by letters, rows are represented by numbers.

```python
>>> print(Coord(2, 5))
B5
```
Operations

Mathematical operations are an easy way to translate a Coord to another locations.

You can use a Size to move a coord to another position. You can also use a tuple \((x, y)\) or a single quantity (integer):

```python
>>> from benker.size import Size

>>> Coord(2, 5) + Size(1, 2)
Coord(x=3, y=7)

>>> Coord(2, 5) + (1, 2)
Coord(x=3, y=7)

>>> Coord(2, 5) + 1
Coord(x=3, y=6)
```

The translation can be positive or negative:

```python
>>> Coord(2, 5) - Size(1, 2)
Coord(x=1, y=3)

>>> Coord(2, 5) - (1, 2)
Coord(x=1, y=3)

>>> Coord(2, 5) - 1
Coord(x=1, y=4)
```

You cannot add or subtract two coordinates:

```python
>>> Coord(2, 5) + Coord(2, 1)
Traceback (most recent call last):
  ...
TypeError: <class 'benker.coord.Coord'>

>>> Coord(2, 5) - Coord(1, 2)
Traceback (most recent call last):
  ...
TypeError: <class 'benker.coord.Coord'>
```

Again, you cannot add a size and a coord:

```python
>>> Size(2, 5) + Coord(2, 1)
Traceback (most recent call last):
  ...
TypeError: <class 'benker.coord.Coord'>

>>> Size(2, 5) - Coord(1, 2)
Traceback (most recent call last):
  ...
TypeError: <class 'benker.coord.Coord'>
```

**Warning:** This constraint must be respected in order to help diagnosing conceptual errors.
4.2.3 Box

Description

A Box is a rectangular area defined by two coordinates:
- the top-left corner of the rectangle: the min coord,
- the bottom-right corner of the rectangle: the max coord.

The default size of a Box is (1, 1), so you can create a box by only specifying the top-left corner of the rectangle.

```python
>>> from benker.box import Box

>>> Box(1, 2, 2, 3)
Box(min=Coord(x=1, y=2), max=Coord(x=2, y=3))

>>> Box(1, 2)
Box(min=Coord(x=1, y=2), max=Coord(x=1, y=2))
```

You can use two coordinates to define a box:

```python
>>> from benker.coord import Coord

>>> Box(Coord(5, 6), Coord(7, 8))
Box(min=Coord(x=5, y=6), max=Coord(x=7, y=8))

>>> Box(Coord(5, 6))
Box(min=Coord(x=5, y=6), max=Coord(x=5, y=6))
```

You can specify the size of a box:

```python
>>> from benker.size import Size

>>> Box(Coord(5, 6), Size(3, 2))
Box(min=Coord(x=5, y=6), max=Coord(x=7, y=7))
```

We use the Excel convention to represent a Box: columns are represented by letters, rows are represented by numbers.

```python
>>> print(Box(Coord(2, 5)))
B5
>>> print(Box(Coord(2, 5), Size(3, 2)))
B5:D6
```

Properties

You can use the following properties to extract information from a box:
- use min to get the top-left corner coordinates,
- use max to get the bottom-right corner coordinates,
- use width to get the width of the box (number of columns),
- use height to get the height of the box (number of rows),
- use size to get the size (width and height) of the box.
Benker Documentation, Release 0.4.5

```python
>>> b1 = Box(Coord(5, 6), Size(3, 2))

>>> b1.min
Coord(x=5, y=6)
>>> b1.max
Coord(x=7, y=7)
>>> b1.width
3
>>> b1.height
2
>>> b1.size
Size(width=3, height=2)
```

**Warning:** All properties are non-mutable:

```python
>>> b1.width = 9
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
```

**Operations**

**Contains**

You can check if a point, defined by its coordinates (tuple (x, y) or Coord instance), is contained in a box:

```python
>>> top_left = Coord(5, 6)
>>> top_right = Coord(6, 6)
>>> bottom_left = Coord(5, 8)
>>> bottom_right = Coord(6, 8)

>>> b1 = Box(top_left, bottom_right)

>>> top_left in b1
True
>>> top_right in b1
True
>>> bottom_left in b1
True
>>> bottom_right in b1
True
>>> Coord(7, 6) in b1
False
>>> (5, 7) in b1
True
```
Warning: Even if a Size object is a subtype of tuple, such an object cannot be “contained” in a Box.

```python
>>> b1 = Box(Coord(x=5, y=6), Coord(x=6, y=8))
>>> Size(5, 7) in b1
Traceback (most recent call last):
...
TypeError: <class 'benker.size.Size'>
```

You can check if a Box is contained in another box:

```python
>>> b1 = Box(Coord(x=5, y=6), Coord(x=6, y=8))
>>> b2 = Box(Coord(x=5, y=7), Coord(x=6, y=7))
>>> b3 = Box(Coord(x=6, y=6), Coord(x=7, y=6))

>>> b1 in b1
True
>>> b2 in b1
True
>>> b3 in b1
False
```

**Intersection and Union**

You can find if a Box intersects another Box:

```python
>>> b1 = Box(Coord(x=1, y=1), Coord(x=3, y=3))
>>> b2 = Box(Coord(x=2, y=2), Coord(x=4, y=4))
>>> b3 = Box(Coord(x=4, y=1), Coord(x=5, y=1))

>>> b1.intersect(b2)
True
>>> b1.intersect(b3)
False
```

Two boxes are disjoint if they don’t intersect each other:

```python
>>> b1.isdisjoint(b2)
False
>>> b1.isdisjoint(b3)
True
```

You can calculate the intersection of two boxes. You can use the “&” operator to do that:

```python
>>> b1.intersection(b2)
Box(min=Coord(x=2, y=2), max=Coord(x=3, y=3))
>>> b1 & b2
Box(min=Coord(x=2, y=2), max=Coord(x=3, y=3))
```

Warning: If the two boxes are disjoint, there is no intersection:
You can calculate the union of two boxes. The union of two boxes is the bounding box: You can use the “|” operator to do that:

```python
gb1 = Box(Coord(3, 2), Coord(6, 4))
>>> gb1 | gb1
Box(min=Coord(x=1, y=1), max=Coord(x=4, y=4))
```

### Total ordering

A total ordering is defined for the boxes. The aim is to order the cells in a grid sorted from left to right and from top to bottom. This order is useful to group the cells by rows.

You can compare boxes:

```python
gb1 = Box(Coord(3, 2), Coord(6, 4))
>>> gb1 < gb1
False
>>> gb1 < Box(Coord(3, 2), Coord(6, 5))
True
>>> gb1 < Box(Coord(3, 2), Coord(7, 4))
True
>>> gb1 < Box(Coord(4, 2), Coord(6, 4))
True
>>> gb1 < Box(Coord(3, 3), Coord(6, 4))
True
```

You can sort boxes. The sort order can be defined as below:

- top cells are sorted before bottom cells,
- top-left cells are sorted before top-right cells,
- smaller cells are sorted before bigger.

```python
gfrom random import shuffle

gboxes = [Box(x, y) for x in range(1, 4) for y in range(1, 3)]
>>> [str(box) for box in sorted(bboxes)]
```
4.2.4 Styled

Description

A *Styled* object contains a dictionary of styles.

It is mainly used for *Table, RowView, ColView,* and *Cell.*

```python
>>> from benker.styled import Styled
>>> styled = Styled({'text-align': 'justify'}, "body")
```

The representation of a styled is the representation of its dictionary of styles:

```python
>>> print(styled)
{'text-align': 'justify'}
```

Attributes

A *Styled* object has the following attribute:

- *styles* is the user-defined styles: a dictionary of key-value pairs. This values are useful to store some HTML-like styles (border-style, border-width, border-color, vertical-align, text-align, etc.). Of course, we are not tied to the HTML-like styles, you can use your own styles list.

**Note:** The style dictionary is always copied: in other words, key-value pairs are copied but a shallow copy is done for the values (in general, it is not a problem if you use non-mutable values like *str*).

- *nature*: a way to distinguish the body cells, from the header and the footer. The default value is “body”, but you can use “header”, “footer” or whatever is suitable for your needs. This kind of information is in general not stored in the styles, even if it is similar.

Tables can also have a *nature*, similar to HTML *@class* attribute, you can use it do identify the styles to apply to your table. For tables, the default value is *None*.

**Note:** In a *Grid*, the merging of two natures is done by keeping the first nature and dropping the second one. In other words, the resulting nature is the group of the most top-left nature of the merged cells.

Example of *styles* initialisation and shallow copy:

```python
>>> css = { 'border-style': 'solid', 'border-width': '5px' }
>>> one = Styled(css, "body")
>>> one.styles['border-width'] = '2px 10px 4px 20px'
>>> two = Styled(one.styles, "body")
>>> two.styles['border-width'] = 'medium'

>>> css
{'border-style': 'solid', 'border-width': '5px'}

>>> one.styles
{'border-style': 'solid', 'border-width': '2px 10px 4px 20px'}
```

(continues on next page)
4.2.5 Cell

Description

A **Cell** object stores the *content* of a *Grid* cell.

A cell can have *styles*, a dictionary of key-value properties attached to the cell.

A cell has a *type* to distinguish between header, body and footer cells. The default type is “body”, but you can also use “header”, “footer” or whatever...

A cell has top-left coordinates: *x* and *y*. The default coordinates is (1, 1): this is the top-left coordinate of the cell box. The coordinates *x* and *y* cannot be null: grid coordinates are 1-indexed.

A cell has a size: *width* and *height*. The default size is (1, 1), you can increase them to represent horizontal or vertical spanning. The *width* and the *height* cannot be null.

To instantiate a *Cell*, you can do:

```python
>>> from benker.cell import Cell
>>> c1 = Cell("c1")
>>> c2 = Cell("c2", styles={'color': 'red'})
>>> c3 = Cell("c3", x=2, y=3, nature="footer")
>>> c4 = Cell("c4", width=2)
>>> c5 = Cell("c5", height=2)
```

The string representation of a cell is the string representation of it’s content:

```python
>>> for cell in c1, c2, c3, c4, c5:
...     print(cell)
```

**Attributes**

A cell has the following attributes:

- *content* is the user-defined cell content. It can be of any type: *None*, *str*, *int*, *float*, a container (*list*), a *XML* element, etc. The same content can be shared by several cells, it’s your own responsibility to handle the copy (or deep copy) of the *content* reference when needed.

**Note:** In a *Grid*, the *merging* of two cell contents is done with the “*+*” operator (__add__()). You can override this by using a *content_appender*, a two-arguments function which will perform the concatenation of the two contents.
• **styles** is the user-defined cell styles: a dictionary of key-value pairs. This values are useful to store some HTML-like styles (border-style, border-width, border-color, vertical-align, text-align, etc.). Of course, we are not tied to the HTML-like styles, you can use your own styles list.

**Note:** The style dictionary is always copied: in other words, key-value pairs are copied but a shallow copy is done for the values (in general, it is not a problem if you use non-mutable values like `str`).

• **type** is a way to distinguish the body cells, from the header and the footer. The default value is “body”, but you can use “header”, “footer” or whatever is suitable for your needs.

**Note:** In a Grid, the **merging** of two cell types is done by keeping the first cell type and dropping the second one. In other words, the resulting cell type is the type of the most top-left cell type of the merged cells.

Using the cell attributes:

```python
>>> paragraphs = ['Hello', 'How are you?']
>>> css = {'text-align': 'justify', 'vertical-align': 'top'}

>>> c1 = Cell(paragraphs, styles=css, nature="normal")
>>> c2 = Cell(paragraphs, styles=css, nature="normal")

# this will mutate the referenced `paragraphs` list:
>>> c1.content.append("I am well.")
>>> c2.content
['Hello', 'How are you?', 'I am well.']

# this will change only the cell styles:
>>> c1.styles["vertical-align"] = 'middle'
>>> c2.styles == {'text-align': 'justify', 'vertical-align': 'top'}
True
```

### Properties

You can use the following properties to extract information from a cell:

- use **min** to get the top-left corner coordinates,
- use **max** to get the bottom-right corner coordinates,
- use **width** to get the width of the box (number of columns),
- use **height** to get the height of the box (number of rows),
- use **size** to get the size (**width** and **height**) of the box.

```python
>>> c1 = Cell("Hi", x=5, y=6, width=3, height=2)

>>> c1.min
Coord(x=5, y=6)
>>> c1.max
Coord(x=7, y=7)
>>> c1.width
3
```

(continues on next page)
Warning: All properties are non-mutable:

```python
>>> c1.width = 9
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
```

Operations

Contains

You can check if a point, defined by its coordinates (tuple \((x, y)\) or \(\text{Coord} \) instance), is contained in a \(\text{Cell} \).

A cell contains a point if it is in its \(\text{Box} \). This rule is explained in detail in the section \(\text{Box – Contains} \).

```python
>>> c1 = Cell("A", x=2, y=3, width=2, height=1)

>>> (3, 3) in c1
True

>>> (7, 9) in c1
False
```

Total ordering

A total ordering is defined for the cells. The aim is to order the cells in a grid sorted from left to right and from top to bottom. This order is useful to group the cells by rows.

The total ordering is base on the \(\text{Box Total ordering} \).

```python
>>> c1 = Cell("one")
>>> c2 = Cell("two", x=2)
>>> c3 = Cell("three", y=2)

>>> c1 < c2 < c3
True
```

This total ordering allow us to sort the cells:

```python
>>> from random import shuffle

>>> cells = [c1, c2, c3]

>>> shuffle(cells)

>>> [str(cell) for cell in sorted(cells)]
['one', 'two', 'three']
```
Transformations

It is possible to change the cell position and size by using two kind of transformations:

- Move a cell to a different coordinates,
- Resize a cell.

```python
>>> from benker.coord import Coord
>>> from benker.size import Size

>>> c1 = Cell("A")
>>> c1
<Cell('A', styles={}, nature=None, x=1, y=1, width=1, height=1)>

>>> c1.move_to(Coord(2, 3))
<Cell('A', styles={}, nature=None, x=2, y=3, width=1, height=1)>

>>> c1.resize(Size(3, 4))
<Cell('A', styles={}, nature=None, x=1, y=1, width=3, height=4)>

>>> c1.transform(Coord(2, 3), Size(3, 4))
<Cell('A', styles={}, nature=None, x=2, y=3, width=3, height=4)>
```

The transformation functions don’t change the current cell but produce a new one with new coordinates/size.

### 4.2.6 Grid

**Description**

A Grid is a collection of Cell objects ordered in a grid of rows and columns.

You can define a empty grid like this:

```python
>>> from benker.grid import Grid

>>> Grid()
Grid([])
```

You can also define a grid from a collection (list, set...) of cells. Cells are ordered according to the total ordering of the cell boxes:

```python
>>> from benker.cell import Cell

>>> red = Cell('red', x=1, y=1, height=2)
>>> pink = Cell('pink', x=2, y=1, width=2)
>>> blue = Cell('blue', x=2, y=2)

>>> grid = Grid([red, blue, pink])
>>> for cell in grid:
...     print(cell)
red
pink
blue
```
**Warning:** If at least one cell intersect another one, an exception is raised:

```python
>>> Grid([Cell("one"), Cell("two")])
Traceback (most recent call last):
  ...
  KeyError: Coord(x=1, y=1)
```

So, it is important to define the coordinates of the cells.

It’s easy to copy the cells of another grid.

Remember that:

- cells are copied (not shared between grids),
- cell contents are shared: two different cells share the same content,
- cell styles are copied (but not deeply).

```python
>>> grid1 = Grid([red, blue, pink])
>>> grid2 = Grid(grid1)

>>> tuple(id(cell) for cell in grid1) != tuple(id(cell) for cell in grid2)
True

>>> tuple(id(cell.content) for cell in grid1) == tuple(id(cell.content) for cell in grid2)
True

>>> tuple(id(cell.styles) for cell in grid1) != tuple(id(cell.styles) for cell in grid2)
True
```

You can pretty print a grid:

```python
>>> grid = Grid([red, blue, pink])
>>> print(grid)
+-----------+-----------------------+
| red | pink |
| +-----------+-----------+
| | blue |
>+-----------+-----------+-----------+
```

**Properties**

The bounding box of a grid is the bounding box of all cells:

```python
>>> grid = Grid()
>>> grid[1, 1] = Cell("red", height=2)
>>> grid[2, 1] = Cell("pink", width=2)  
>>> grid[3, 2] = Cell("gray")
>>> print(grid)
+-----------+-----------------------+
| red | pink |
| +-----------+-----------+
| | blue |
>+-----------+-----------+-----------+
```

(continues on next page)
grid.bounding_box
Box(min=Coord(x=1, y=1), max=Coord(x=3, y=2))

Important: The bounding box is not defined for an empty grid, so None is returned in that case (this behavior is preferable to raising an exception, in order to simplify interactive debugging).

grid = Grid()
grid.bounding_box is None
True

Operations

Contains

You can check if a point, defined by its coordinates (tuple (x, y) or Coord instance), is contained in a Grid.

The rule is simple: a grid contains a point if it exists a Cell of the grid which contains that point. In other words, a point may be contained in the bounding box of a grid but not in any cell if there are some gaps in the grid.

from benker.coord import Coord
red = Cell('red', x=1, y=1, height=2)
pink = Cell('pink', x=2, y=1, width=2)
blue = Cell('blue', x=2, y=2)
grid = Grid([red, blue, pink])

(1, 1) in grid
True
(3, 1) in grid
True
(4, 1) in grid
False
(3, 2) in grid
False
Coord(1, 2) in grid
True

Set, Get, Delete cells

A grid is a MutableMapping, it works like a dictionary of cells. Keys of the dictionary are coordinates (tuple (x, y) or Coord instance). The coordinates are the top-left coordinates of the cells.

grid = Grid()
grid[1, 1] = Cell("red", height=2)
grid[2, 1] = Cell("pink", width=2)
grid[2, 2] = Cell("blue")
>>> grid[3, 2] = Cell("gray")

>>> print(grid)
+-----------+-----------------------+
| red       | pink                  |
|           | +-----------------------+
|           | | blue       | gray       |
+-----------+-----------------------+

**Warning:** Unlike a `dict`, you cannot set a cell to a given location if a cell already exist in that location, an exception is raised in that case.

>>> grid[3, 1] = Cell("purple")
Traceback (most recent call last):
...
KeyError: Coord(x=3, y=1)

You can get a cell at a given location:

```python
>>> grid[1, 1]
<Cell('red', styles={}, nature=None, x=1, y=1, width=1, height=2)>
>>> grid[3, 1]
<Cell('pink', styles={}, nature=None, x=2, y=1, width=2, height=1)>
```

You can delete a cell at a given location:

```python
>>> del grid[3, 1]
>>> print(grid)
+-----------+-----------+-----------+
| red       | |          |
|           | +-----------+-----------+
|           | | blue       | gray       |
+-----------+-----------+-----------+
```

**Merging/expanding**

It is possible to merge several cells in the grid. The merging takes the `start` coordinates and the `end` coordinates of the cells to merge.

We can define a `content_appender` to give the content merging operation to use to merge several cell contents.

```python
>>> grid = Grid()
>>> grid[1, 1] = Cell("red", height=2)
>>> grid[2, 1] = Cell("pink")
>>> grid[3, 1] = Cell("blue")
>>> print(grid)
+-----------+-----------+-----------+
| red       | pink      | blue      |
|           |           |           |
|           | +-----------+-----------+
|           |           |           |
+-----------+-----------+-----------+
```
>>> grid.merge((2, 1), (3, 1), content_appender=lambda a, b: "/".join([a, b]))
<Cell('pink/blue', styles={}, nature=None, x=2, y=1, width=2, height=1)>

```python
>>> print(grid)
+-----------+-----------+-----------+
| red | pink/blue |
| +-----------+-----------+
```

**Warning:** All cells in the bounding box of the merging must be inside of the bounding box. In other words, the bounding box of the merging must not intersect any cell in the grid.

```python
>>> grid.merge((1, 2), (2, 2))
Traceback (most recent call last):
  ... 
ValueError: ((1, 2), (2, 2))
```

Similar to the merging, you can expand the size of a cell:

```python
>>> grid = Grid()
>>> grid[1, 1] = Cell("red", height=2)
>>> grid[2, 1] = Cell("pink")
>>> grid[3, 1] = Cell("blue")
>>> print(grid)
+-----------+-----------+-----------+
| red | pink | blue |
| +-----------+-----------+
```

```python
>>> grid.expand((2, 1), height=1)
<Cell('pink', styles={}, nature=None, x=2, y=1, width=1, height=2)>

>>> print(grid)
+-----------+-----------+-----------+
| red | pink | blue |
| +-----------+
```

### 4.2. Tutorials
Iterators

You can iterate the cells of a grid:

```python
>>> grid = Grid()
>>> grid[1, 1] = Cell("red", height=2)
>>> grid[2, 1] = Cell("hot", width=2)
>>> grid[2, 2] = Cell("chili")
>>> grid[3, 2] = Cell("peppers")
>>> grid[1, 3] = Cell("Californication", width=3)
```

```python
>>> print(grid)
+-----------+-----------------------+
| red       | hot                   |
| +-----------+-----------+                  |
|           | chili      | peppers               |
+-----------------------------------+                  |
|                                | Californi  |
+-----------------------------------+                  |
```

```python
>>> for cell in grid:
...     print(cell)
red
hot
chili
peppers
Californication
```

You can iterate over the grid rows with the method `iter_rows()`. Each row is a tuple of cells:

```python
>>> for row in grid.iter_rows():
...     print(" / ".join(cell.content for cell in row))
red / hot
chili / peppers
Californication
```

4.2.7 Table

Description

A `Table` is a data structure used to represent Office Open XML tables, CALS tables or HTML tables.

A `Table` is a `Styled` object, so you can attach a dictionary of styles and a nature (“body” by default). The nature is used to give a default value to the the row/column views.

```python
>>> from benker.table import Table
>>> Table(styles={'frame': 'all'})
<Table({'frame': 'all'}, None)>
```

A table can be initialize with a collection of cells. Make sure all cells are disjoints.
```python
>>> from benker.cell import Cell

>>> red = Cell('red', x=1, y=1, height=2)
>>> pink = Cell('pink', x=2, y=1, width=2)
>>> blue = Cell('blue', x=2, y=2)

>>> table = Table([red, pink, blue], nature='header')
>>> table
<Table({}, 'header')>

>>> print(table)
+-----------+-----------------------+
| red | pink |
| +-----------+-----------+
| | blue | |
| +-----------+-----------+-----------+

```

**Warning:** Make sure all cells are disjoints:

```python
>>> red = Cell('overlap', x=1, y=1, width=2)
>>> pink = Cell('oops!', x=2, y=1)
>>> Table([red, pink])
Traceback (most recent call last):
  ...
KeyError: Coord(x=2, y=1)
```

**Properties**

You can use the following properties to extract information from a *table*:

The bounding box of a table is the bounding box of all cells in the grid:

```python
>>> red = Cell('red', x=1, y=1, height=2)
>>> pink = Cell('pink', x=2, y=1, width=2)
>>> blue = Cell('blue', x=2, y=2)
>>> table = Table([red, pink, blue])

>>> table.bounding_box
Box(min=Coord(x=1, y=1), max=Coord(x=3, y=2))
```

**Important:** The bounding box is not defined for an empty table, so *None* is returned in that case (this behavior is preferable to raising an exception, in order to simplify interactive debugging).

```python
>>> table = Table()
>>> table.bounding_box is None
True
```
Operations

Cells Insertion

You can insert a row to a table. This row is then used to insert cells.

```python
>>> table = Table()

>>> row = table.rows[1]
>>> row.nature = "header"
>>> row.insert_cell("Astronomer", width=2)
>>> row.insert_cell("Year")
>>> row.insert_cell("Country")

>>> row = table.rows[2]
>>> row.insert_cell("Nicolaus")
>>> row.insert_cell("Copernicus")
>>> row.insert_cell("1473-1543")
>>> row.insert_cell("Royal Prussia")

>>> row = table.rows[3]
>>> row.insert_cell("Charles")
>>> row.insert_cell("Messier")
>>> row.insert_cell("1730-1817")
>>> row.insert_cell("France", height=2)

>>> row = table.rows[4]
>>> row.insert_cell("Jean-Baptiste")
>>> row.insert_cell("Delambre")
>>> row.insert_cell("1749-1822")

>>> print(table)
+-----------------------+-----------+-----------+
| Astronome | Year | Country |
+-----------+-----------+-----------+
| Nicolaus | Copernicus | 1473-1543 | Royal Pru |
+-----------+-----------+-----------+-----------+
| Charles | Messier | 1730-1817 | France |
+-----------------------+-----------+-----------+-----------+
| Jean-Bapt | Delambre | 1749-1822 | |
+-----------------------+-----------+-----------+-----------+
```

The `nature` of a cell is inherited from its parent’s row. The first row contains the header, so the cell nature is “header”:

```python
>>> table.rows[1].nature
'header'

>>> [cell.nature for cell in table.rows[1].owned_cells]
['header', 'header', 'header']
```

The other rows have no `nature`, so the cell nature is None

```python
>>> table.rows[2].nature is None
True
```
Cells Merging

You can merge cells by giving the coordinates of the cells to merge or by extending the size of a given cell.

```python
>>> table = Table()
>>> letters = "abcdEFGHijklMNOP"
>>> for index, letter in enumerate(letters):
...     table[(1 + index % 4, 1 + index // 4)] = Cell(letter)
>>> print(table)
+-----------+-----------+-----------+-----------+
| a | b | c | d |
+-----------+-----------+-----------+-----------+
| E | F | G | H |
+-----------+-----------+-----------+-----------+
| i | j | k | l |
+-----------+-----------+-----------+-----------+
| M | N | O | P |
+-----------+-----------+-----------+-----------+

>>> table.merge((2, 2), (3, 3))
>>> print(table)
+-----------+-----------+-----------+-----------+
| a | b | c | d |
+-----------------------+-----------+
| E | FGjk | H |
+-----------------------+-----------+
| i | | l |
+-----------+-----------+-----------+-----------+
| M | N | O | P |
+-----------------------+-----------+

>>> table.expand((2, 3), height=1)
>>> print(table)
+-----------+-----------+-----------+-----------+
| a | b | c | d |
+-----------------------+-----------+
| E | | H |
+-----------------------+-----------+
| i | FGjkNO | l |
+-----------------------+-----------+
| M | | P |
+-----------------------+-----------+
```

4.2. Tutorials 31
Owned and caught cells

When a cell is merged into a row group, it is always bound to the top row of this group (the first row). In that case, we say that the first row owns the cell and the other rows catch the cell.

```python
>>> table = Table()
>>> row = table.rows[1]
>>> row.insert_cell("merged", height=2)
>>> row.insert_cell("A")

>>> row = table.rows[2]
>>> row.insert_cell("B")

>>> row = table.rows[3]
>>> row.insert_cell("C")
>>> row.insert_cell("D")

>>> print(table)
+-----------+-----------+
| merged    | A |
| +-----------+
|           | B |
| +-----------+-----------+
| C | D |
| +-----------+-----------+
```

Here are the owned_cells of this table:

```python
>>> for pos, row in enumerate(table.rows, 1):
...     cells = "", ".join("{}").format(cell) for cell in row.owned_cells
...     print("row #pos: {cells}".format(pos=pos, cells=cells))

table owned cells:
row #1: merged, A
row #2: B
row #3: C, D
```

Here are the caught_cells of this table:

```python
>>> for pos, row in enumerate(table.rows, 1):
...     cells = "", ".join("{}").format(cell) for cell in row.caught_cells
...     print("row #pos: {cells}".format(pos=pos, cells=cells))

table caught cells:
row #1: merged, A
row #2: merged, B
row #3: C, D
```

The same applies to columns: if a cell is merged into several columns then it belongs to the first column (left) of the merged column group.

```python
>>> table = Table()
>>> row = table.rows[1]
>>> row.insert_cell("merged", width=2)
>>> row.insert_cell("A")

>>> row = table.rows[2]
```
Benker Documentation, Release 0.4.5

>>> row.insert_cell("B")
>>> row.insert_cell("C")
>>> row.insert_cell("D")
>>> print(table)
+-----------------------+-----------+
| merged | A |
+-----------+-----------+-----------+
| B | C | D |
+-----------+-----------+-----------+

Here are the **owned_cells** of this table:

```python
>>> for pos, col in enumerate(table.cols, 1):
...     cells = ', '.join("{}".format(cell) for cell in col.owned_cells)
...     print("col #{}: {}".format(pos, cells))
col #1: merged, merged, B
col #2: C
col #3: A, D
```

Here are the **caught_cells** of this table:

```python
>>> for pos, col in enumerate(table.cols, 1):
...     cells = ', '.join("{}".format(cell) for cell in col.caught_cells)
...     print("col #{}: {}".format(pos, cells))
col #1: merged, merged, B
col #2: merged, merged, C
col #3: A, D
```

## 4.3 API

Organize your data in a spatial grid system for CALS, HTML, Formex4, Office Open XML tables conversion

### 4.3.1 Cell Size

A `Size` object is used to represent the *width* and *height* of a *Cell*. The *width* is the number of spanned columns and the *height* is the number of spanned rows. The default cell size is (1, 1).

This module defines the `Size` tuple and give some classic use cases.

#### class benker.size.Size(width, height)

**Bases:** `benker.size.SizeTuple`

Size of a cell: *width* is the number of columns and *height* is the number of row.

**Usage:**

```python
>>> from benker.size import Size

>>> size = Size(2, 1)
>>> size
Size(width=2, height=1)
```
You can use the “+” or “-” operators to increase or decrease the size:

```python
>>> Size(2, 1) + Size(3, 3)
Size(width=5, height=4)
>>> Size(5, 4) - Size(3, 3)
Size(width=2, height=1)
```

You can expand the width and height to a given factor using the “*” operator:

```python
>>> Size(2, 1) * 2
Size(width=4, height=2)
```

You can have negative or positive sizes using the unary operators “-” and “+”:

```python
>>> +Size(3, 2)
Size(width=3, height=2)
>>> -Size(3, 2)
Size(width=-3, height=-2)
```

Note: A Cell object cannot have a negative or null sizes, but you can need this values for calculation, for instance when you want to reduce the cell size.

```python
classmethod from_value(value)
    Convert a value of type tuple to a Size tuple.

    Parameters value -- tuple of two integers or Size tuple.

    Returns Newly created object.

    Raises TypeError -- if the value is not a tuple of integers nor a Size tuple.
```

```python
class benker.size.SizeTuple(width, height)
Bases: tuple

height
    Alias for field number 1

width
    Alias for field number 0
```
4.3.2 Cell Coordinates

A Coord object is used to represent the x and y positions of a Cell. The x is the left position (column number) and the y is the top position (row number). The default cell coordinates is (1, 1).

This module defines the Coord tuple and give some classic use cases.

class benker.coord.Coord(x, y)

    Bases: benker.coord.CoordTuple

    Coordinates of a cell in a grid: x is the left column, y if the top row.

Usage:

```python
>>> from benker.coord import Coord

>>> coord = Coord(5, 3)
>>> coord
Coord(x=5, y=3)
>>> coord.x
5
>>> coord.y
3
>>> str(coord)
'E3'
```

You can use the “+” or “-” operators to move the coordinates:

```python
>>> from benker.size import Size

>>> Coord(2, 1) + Size(3, 3)
Coord(x=5, y=4)
>>> Coord(5, 4) - Size(3, 3)
Coord(x=2, y=1)
```

**Warning:** You cannot add or subtract two coordinates, only a coordinate and a size.

```python
>>> from benker.coord import Coord

>>> Coord(2, 1) + Coord(3, 3)
Traceback (most recent call last):
...
TypeError: <class 'benker.coord.Coord'>
```

classmethod from_value(value)

    Convert a value of type tuple to a Coord tuple.

    Parameters value – tuple of two integers or Coord tuple.

    Returns Newly created object.

    Raises TypeError – if the value is not a tuple of integers nor a Coord tuple.

class benker.coord.CoordTuple(x, y)

    Bases: tuple
4.3.3 Box

A Box is a rectangular area defined by two coordinates:

- the top-left corner of the rectangle: the min coord,
- the bottom-right corner of the rectangle: the max coord.

To instantiate a Box, you can do:

```python
>>> b1 = Box(Coord(5, 6), Coord(7, 8))
>>> b2 = Box(Coord(5, 6))
>>> b3 = Box(1, 2, 2, 3)
>>> b4 = Box(1, 2)
>>> b5 = Box(b1)
```

Box objects have a string representation à la Excel:

```python
>>> for box in b1, b2, b3, b4, b5:
...     print(box)
E6:G8
E6
A2:B3
A2
E6:G8
```

You can calculate the width and height of boxes:

```python
>>> b1 = Box(Coord(5, 6), Coord(7, 8))
>>> b1.width, b1.height
(2, 3)
```

You can determine if a Coord is included in a Box:

```python
>>> top_left = Coord(5, 6)
>>> top_right = Coord(6, 6)
>>> bottom_left = Coord(5, 8)
>>> bottom_right = Coord(6, 8)
>>> b1 = Box(top_left, bottom_right)

>>> top_left in b1
True
>>> top_right in b1
True
```

(continues on next page)
You can determine if two boxes intersect each other, or are disjoints:

```python
>>> b1 = Box(Coord(5, 6), Coord(6, 8))
>>> b2 = Box(Coord(6, 6), Coord(6, 7))
>>> b3 = Box(Coord(7, 6), Coord(7, 8))
>>> b2.intersect(b3)
False
>>> b1.isdisjoint(b2)
False
>>> b2.isdisjoint(b1)
False
>>> b1.isdisjoint(b3)
True
>>> b3.isdisjoint(b1)
True
```

```python
class benker.box.Box(*args)
    Bases: benker.box.BoxTuple

    A Box is a rectangular area defined by two coordinates:
    • the top-left corner of the rectangle: the min coord,
    • the bottom-right corner of the rectangle: the max coord.

Usage:

```
```
>>> b1 = Box(Coord(3, 2), Coord(6, 4))
>>> b2 = Box(Coord(4, 3), Coord(5, 7))
>>> b1.intersection(b2)
Box(min=Coord(x=4, y=3), max=Coord(x=5, y=4))

>>> b1 & b2
Box(min=Coord(x=4, y=3), max=Coord(x=5, y=4))
```

**Parameters** `others`  – collections of boxes

**Returns** The inner box of all the boxes.

**Raises** `ValueError` – if the two boxes are disjoint.

```
isdisjoint(that: benker.box.Box) → bool
move_to(coord)
resize(size)
property size
transform(coord=None, size=None)
union(*others)
```

**Parameters** `others`  – collections of boxes

**Returns** The bounding box of all the boxes.

```
>>> from benker.box import Box
>>> from benker.coord import Coord

>>> b1 = Box(Coord(3, 2), Coord(6, 4))
>>> b2 = Box(Coord(4, 3), Coord(5, 7))
>>> b1.union(b2)
Box(min=Coord(x=3, y=2), max=Coord(x=6, y=7))

>>> b1 | b2
Box(min=Coord(x=3, y=2), max=Coord(x=6, y=7))
```

**Parameters** `others`  – collections of boxes

**Returns** The union of `self` and all the boxes.

**property width**

```
class benker.box.BoxTuple(min, max)
    Bases: tuple
    max
        Alias for field number 1
    min
        Alias for field number 0
```
4.3.4 Styled object

A *Styled* object contains a dictionary of styles. It is mainly used for *Table, RowView, ColView, and Cell*.

```python
class benker.styled.Styled(styles, nature):
    Bases: object

    Styled object, like Table, Row, Column, or Cell objects.

    A styled object stores user-defined styles: a dictionary of key-value pairs. This values are useful to store some HTML-like styles (border-style, border-width, border-color, vertical-align, text-align, etc.). Of course, we are not tied to the HTML-like styles, you can use your own list of styles.

    **Note:** The style dictionary is always copied: in other words, key-value pairs are copied but a shallow copy is done for the values (in general, it is not a problem if you use non-mutable values like `str`).
```

A styled object stores a nature: a way to distinguish the body cells, from the header and the footer. The default value is `None`, but you can use “body”, “header”, “footer” or whatever is suitable for your needs. This kind of information is in general not stored in the styles, even if it is similar.

Tables can also have a `nature`, similar to HTML `@class` attribute, you can use it do identify the styles to apply to your table.

**Note:** In a *Grid*, the *merging* of two natures is done by keeping the first nature and dropping the second one. In other words, the resulting nature is the group of the most top-left nature of the merged cells.

```python
definitions:
    nature
      Cell `nature` used to distinguish the body cells, from the header and the footer.

    property styles
      Dictionary of key-value pairs, where keys are the style names.
```

4.3.5 Grid Cell

A *Cell* object stores the *content* of a *Grid* cell.

A cell can have `styles`, a dictionary of key-value properties attached to the cell.

A cell has a `nature` to distinguish between header, body and footer cells. The default `nature` is `None`, but you can also use “body”, “header”, “footer” or whatever...

A cell has top-left coordinates: `x` and `y`. The default coordinates is `(1, 1): this is the top-left coordinate of the cell box. The coordinates `x` and `y` cannot be null: grid coordinates are 1-indexed.

A cell has a size: `width` and `height`. The default size is `(1, 1), you can increase them to represent horizontal or vertical spanning. The `width` and the `height` cannot be null.

To instantiate a *Cell*, you can do:

```python
>>> c1 = Cell("c1")
>>> c2 = Cell("c2", styles={'color': 'red'})
>>> c3 = Cell("c3", nature="footer")
>>> c4 = Cell("c4", width=2)
>>> c5 = Cell("c5", height=2)
```
The string representation of a cell is the string representation of it’s content:

```python
>>> for cell in c1, c2, c3, c4, c5:
...     print(cell)
c1
c2
c3
c4
c5
```

On initialization, the cell min position is always (1, 1), a.k.a. the top-left.

```python
>>> c1 = Cell("c1")
>>> c1.min
Coord(x=1, y=1)
>>> c1.size
Size(width=1, height=1)
>>> c1.box
Box(min=Coord(x=1, y=1), max=Coord(x=1, y=1))
```

A cell can be moved to another position:

```python
>>> c1 = Cell("c1", width=3, height=2)
>>> c2 = c1.move_to(Coord(5, 3))
>>> c2.min
Coord(x=5, y=3)
>>> c2.size
Size(width=3, height=2)
>>> c2.box
Box(min=Coord(x=5, y=3), max=Coord(x=7, y=4))
```

You can check if a coord is inside the box:

```python
>>> c1 = Cell("c1", width=3, height=2)
>>> c2 = c1.move_to(Coord(5, 3))
>>> (7, 4) in c2
True
>>> Coord(6, 3) in c2
True
>>> Box(6, 3, 7, 4) in c2
True
```

```python
class benker.cell.Cell(content, styles=None, nature=None, x=1, y=1, width=1, height=1)
    Bases: benker.styled.Styled
    Cell of a grid.

    Variables
    
    • **content** – user-defined cell content. It can be of any type: None, str, int, float, a container (list), a XML element, etc. The same content can be shared by several cells, it’s your own responsibility to handle the copy (or deep copy) of the content reference when needed.

    Note: In a Grid, the merging of two cell contents is done with the “+” operator (__add__). You can override this by using a content_appender, a two-arguments function
```
which will perform the concatenation of the two contents.

- **styles** – user-defined cell styles: a dictionary of key-value pairs. This values are useful to store some HTML-like styles (border-style, border-width, border-color, vertical-align, text-align, etc.). Of course, we are not tied to the HTML-like styles, you can use your own list of styles.

  **Note:** The style dictionary is always copied: in other words, key-value pairs are copied but a shallow copy is done for the values (in general, it is not a problem if you use non-mutable values like `str`).

- **nature** – nature: a way to distinguish the body cells, from the header and the footer. The default value is `None`, but you can use “body”, “header”, “footer” or whatever is suitable for your needs.

  **Note:** In a `Grid`, the merging of two natures is done by keeping the first nature and dropping the second one. In other words, the resulting nature is the group of the most top-left nature of the merged cells.

Changed in version 0.4.2: The default value of `nature` is `None` (instead of “body”).

**property box**
  Bounding box of the cell.

**property height**
  Height of the cell – rows spanning.

**property max**
  Maximum coordinates of the cell – bottom-right coordinates.

**property min**
  Minimum coordinates of the cell – top-left coordinates.

**move_to(coord)**
  Move the cell to the given coordinates.
  
  See: `transform()`.
  
  **Parameters**
  
  `coord` (`tuple[int, int]` or `benker.coord.Coord`) – new top-left coordinates of the cell, by default it is unchanged.

  **Return type** `benker.cell.Cell`

  **Returns** a copy of this cell after transformation.

**resize(size)**
  Resize the cell to the given size.
  
  See: `transform()`.

  **Parameters**
  
  `size` (`tuple[int, int]` or `benker.size.Size`) – new size of the cell, by default it is unchanged.

  **Return type** `benker.cell.Cell`

  **Returns** a copy of this cell after transformation.
property size
Size of the cell – (width, height).

transform(coord=None, size=None)
Transform the bounding box of the cell by making a move and a resize.

Parameters
• coord (tuple[int, int] or benker.coord.Coord) – new top-left coordinates of the cell, by default it is unchanged.
• size (tuple[int, int] or benker.size.Size) – new size of the cell, by default it is unchanged.

Return type benker.cell.Cell
Returns a copy of this cell after transformation.

property width
Width of the cell – columns spanning.

benker.cell.get_content_text(content)
Try hard to extract a good string representation of the cell content.

```python
>>> from benker.cell import get_content_text

>>> get_content_text(None) == ''
True
>>> get_content_text('') == ''
True
>>> print(get_content_text('Hi'))
Hi
>>> print(get_content_text(True))
True
>>> print(get_content_text(123))
123
>>> print(get_content_text(3.14))
3.14
>>> get_content_text([None, None]) == ''
True
>>> print(get_content_text(['hello', '', 'world!']))
hello world!
```

Parameters content – Cell content.

Returns
the cell text:
• if the content is None: returns ‘’,
• if the content is a string: return the string unchanged,
• if the content is a number: return the string representation of the number,
• if the content is a list of strings, return the concatenated strings (None is ignored),
• if the content is a list of XML nodes, return the concatenated strings of the elements (the processing-instruction and the comments are ignored),
• else: return a concatenation of the string representation of the content items.
New in version 0.4.1.

### 4.3.6 Grid

A grid of cells.

Example of grid:

```
.. doctest:: grid_demo

``` ```
```from benker.grid import Grid
from benker.cell import Cell
``` ```
```grid = Grid()
grid[1, 1] = Cell("red", height=2)
grid[2, 1] = Cell("pink", width=2)
grid[2, 2] = Cell("blue")
``` ```
```print(grid)
+-----------+-----------------------+
| red | pink |
| +-----------+-----------+
| | blue | |
| +-----------+-----------+-----------+
``` ```

You can retrieve the grid cells as follow:

```
``` ```
```from benker.grid import Grid
from benker.cell import Cell
``` ```
```grid = Grid()
grid[1, 1] = Cell("red", height=2)
grid[2, 1] = Cell("pink", width=2)
grid[2, 2] = Cell("blue")
``` ```
```grid[1, 1]
CELL('red', styles={}, nature=None, x=1, y=1, width=1, height=2)
``` ```
```grid[2, 1]
CELL('pink', styles={}, nature=None, x=2, y=1, width=2, height=1)
``` ```
```grid[2, 2]
CELL('blue', styles={}, nature=None, x=2, y=2, width=1, height=1)
``` ```
```grid[3, 3]
Traceback (most recent call last):
...
KeyError: Coord(x=3, y=3)
``` ```

A grid has a bounding box, useful to get the grid sizes:

```
``` ```
```from benker.grid import Grid
from benker.cell import Cell
``` ```
```grid = Grid()
grid[1, 1] = Cell("red", height=2)
``` ```
(continues on next page)
grid[2, 1] = Cell("pink", width=2)
grid[2, 2] = Cell("blue")

grid.bounding_box
Box(min=Coord(x=1, y=1), max=Coord(x=3, y=2))

grid.bounding_box.size
Size(width=3, height=2)

You can expand the cell size horizontally or vertically:

cell = benker.grid.Grid()
cell[1, 1] = Cell("red", height=2)
cell[2, 1] = Cell("pink", width=2)
cell[2, 2] = Cell("blue")
cell.expand((2, 2), width=1)
<Cell('blue', styles={}, nature=None, x=2, y=2, width=2, height=1)>

print(cell)
+-----------+-----------------------+
| red | pink |
| +-----------------------+
| | blue |
+-----------+-----------------------+

The content of the merged cells is merged too:

cell = benker.grid.Grid()
cell[1, 1] = Cell("red", height=2)
cell[2, 1] = Cell("pink", width=2)
cell[2, 2] = Cell("blue", width=2)
cell.merge((2, 1), (3, 2), content_appender=lambda a, b: ""/".join([a, b]))
<Cell('pink/blue', styles={}, nature=None, x=2, y=1, width=2, height=2)>

print(cell)
+-----------+-----------------------+
| red | pink/blue |
| +-----------------------+

class benker.grid.Grid(cells=None)

Base classes: collections.abc.MutableMapping

Collection of Cell objects ordered in a grid of rows and columns.

property bounding_box

Bounding box of the grid (None if the grid is empty).
expand(coord, width=0, height=0, content_appender=None)

Expand (or shrink) the width and/or height of a cell, using the content_appender to append cell contents.

See also the method merge() to merge a group of cells contained in a bounding box.

Parameters

- **coord** – Coordinates of the cell to expand (or shrink).
- **width** – Number of columns to add to the cell width.
- **height** – Number of rows to add to the cell height.
- **content_appender** – Function to use to append the cell contents. The function must have the following signature: \( f(a, b) \rightarrow c \), where \( a, b \) and \( c \) must be of the same type than the cell content. If not provided, the default function is `operator.__add__()`.

Returns The merged cell.

Raises `ValueError` – If the group of cells is empty or if cells cannot be merged.

iter_rows()

Iterate the cells grouped by rows.

merge(start, end, content_appender=None)

Merge a group of cells contained in a bounding box, using the content_appender to append cell contents.

The coordinates start and end delimit a group of cells to merge.

**Warning:** All the cells of the group must be included in the group bounding box, no intersection is allowed. If not, `ValueError` is raised.

See also the method expand() to expand (or shrink) the width and/or height of a cell.

Parameters

- **start** (Coord or tuple[int, int]) – Top-left coordinates of the group of cells to merge.
- **end** – Bottom-right coordinates of the group of cells to merge (inclusive).
- **content_appender** – Function to use to append the cell contents. The function must have the following signature: \( f(a, b) \rightarrow c \), where \( a, b \) and \( c \) must be of the same type than the cell content. If not provided, the default function is `operator.__add__()`.

Returns The merged cell.

Raises `ValueError` – If the group of cells is empty or if cells cannot be merged.

### 4.3.7 Table

Generic table structure which simplify the conversion from docx table format to CALS or HTML tables.

class benker.table.ColView(table, pos, styles=None, nature=None)

Bases: benker.table TableView

A view on the table cells for a given column.

can_catch(cell)

Check if a cell can be caught by that view.

Parameters cell (benker.cell.Cell) – table cell
Returns True if the cell intercept to this view.

can_own(cell)
Check if a cell can be stored it that view.

Parameters cell (benker.cell.Cell) – table cell

Returns True if the cell belong to this view.

property col_pos
Column position in the table (1-based).

insert_cell(content, styles=None, nature=None, width=1, height=1)
Insert a new cell in the column at the next free position, or at the end.

Parameters

• content – User-defined cell content. It can be of any type: None, str, int, float, a container (list), a XML element, etc. The same content can be shared by several cells, it’s your own responsibility to handle the copy (or deep copy) of the content reference when needed.

• styles (typing.Dict[str, str]) – User-defined cell styles: a dictionary of key-value pairs. This values are useful to store some HTML-like styles (border-style, border-width, border-color, vertical-align, text-align, etc.). Of course, we are not tied to the HTML-like styles, you can use your own list of styles.

• width (int) – Width of the cell (columns spanning), default to 1.

• height (int) – Height of the cell (rows spanning), default to 1.

Variables nature – a way to distinguish the body cells, from the header and the footer. The default value is None, but you can use “body”, “header”, “footer” or whatever is suitable for your needs. If set to None, the cell nature is inherited from the column nature.

Changed in version 0.4.2: The nature of a cell is inherited from its parent’s column.

class benker.table.RowView(table, pos, styles=None, nature=None)
Bases: benker.table TableView

A view on the table cells for a given row.

can_catch(cell)
Check if a cell can be caught by that view.

Parameters cell (benker.cell.Cell) – table cell

Returns True if the cell intercept to this view.

can_own(cell)
Check if a cell can be stored it that view.

Parameters cell (benker.cell.Cell) – table cell

Returns True if the cell belong to this view.

insert_cell(content, styles=None, nature=None, width=1, height=1)
Insert a new cell in the row at the next free position, or at the end.

Parameters

• content – User-defined cell content. It can be of any type: None, str, int, float, a container (list), a XML element, etc. The same content can be shared by several cells, it’s your own responsibility to handle the copy (or deep copy) of the content reference when needed.
• **styles** (*typing.Dict[str, str]*) – User-defined cell styles: a dictionary of key-value pairs. This values are useful to store some HTML-like styles (border-style, border-width, border-color, vertical-align, text-align, etc.). Of course, we are not tied to the HTML-like styles, you can use your own list of styles.

• **width** (*int*) – Width of the cell (columns spanning), default to 1.

• **height** (*int*) – Height of the cell (rows spanning), default to 1.

**Variables nature** – a way to distinguish the body cells, from the header and the footer. The default value is `None`, but you can use “body”, “header”, “footer” or whatever is suitable for your needs. If set to `None`, the cell nature is inherited from the row nature.

Changed in version 0.4.2: The *nature* of a cell is inherited from its parent’s row.

**property row_pos**
Row position in the table (1-based).

**class benker.table.Table** (*cells=None, styles=None, nature=None*)
Bases: `benker.styled.Styled`, `collections.abc.MutableMapping`
Table data structure used to simplify conversion to CALS or HTML.

Short demonstration:

```python
>>> from benker.cell import Cell
>>> from benker.table import Table

>>> table = Table(styles={'frame': 'all'})

>>> table[(1, 1)] = Cell("one")
>>> table.rows[1].insert_cell("two")

>>> table[(2, 1)]
<Cell('two', styles={}, nature=None, x=2, y=1, width=1, height=1)>

>>> table.cols[1].insert_cell("alpha")
>>> table.cols[2].insert_cell("beta")

>>> (1, 2) in table
True

>>> del table[(1, 2)]

>>> (1, 2) in table
False

>>> len(table)
3

>>> for cell in table:
...     print(cell)
one
two
beta

>>> for row in table.rows:
...     print(row)
[one, two]
```

(continues on next page)
[beta]

```python
>>> table.merge((1, 2), (2, 2))
>>> print(table)
+-----------+-----------+
| one | two |
+-----------------------+
| beta |
+-----------------------+

>>> table.expand((1, 1), width=3)
>>> print(table)
+-----------------------------------------------+
| onetwo | |
+-----------------------+-----------+-----------+
| beta | | |
+-----------------------+-----------+-----------+
```

### Variables

- **styles** – User-defined table styles: a dictionary of key-value pairs. This values are useful to store some HTML-like styles (border-style, border-width, border-color, vertical-align, text-align, etc.). Of course, we are not tied to the HTML-like styles, you can use your own list of styles.

**Note:** The style dictionary is always copied: in other words, key-value pairs are copied but a shallow copy is done for the values (in general, it is not a problem if you use non-mutable values like `str`).

- **nature** – A table can have a nature: a way to distinguish the body cells, from the header and the footer. The default value is `None`, but you can use “body”, “header”, “footer” or whatever is suitable for your needs. This kind of information is in general not stored in the styles, even if it is similar.

**Note:** In a Grid, the merging of two natures is done by keeping the first nature and dropping the second one. In other words, the resulting nature is the group of the most top-left nature of the merged cells.

### property bounding_box

Bounding box of the table (`None` if the table is empty).

**Return type** `benker.box.Box`

**Returns** The bounding box.

### cols

List of views of type `ColView`

### expand(coord, width=0, height=0, content_appender=None)

### merge(start, end, content_appender=None)
nature

    Cell *nature* used to distinguish the body cells, from the header and the footer.

rows

    List of views of type *RowView*

class benker.table TableView

    **Base class of** *RowView* and *ColView* **used to create a view on the table cells for a given row or column.**

    **See also:** *TableViewList*

    **adopt_cell**(cell)

        Event handler called by the system when a cell is about to be inserted in the table.

    **can_catch**(cell)

        Check if a cell can be caught by that view.

        **Parameters**

            *cell* ([benker.cell.Cell]) – table cell

        **Returns**

            True if the cell intercept to this view.

    **can_own**(cell)

        Check if a cell can be stored it that view.

        **Parameters**

            *cell* ([benker.cell.Cell]) – table cell

        **Returns**

            True if the cell belong to this view.

    **property caught_cells**

        List of cells caught (intercepted) by this view.

    **property owned_cells**

        List of cells owned by this view.

    **property table**

        Non-mutable reference to the table (instance of *Table*).

class benker.table TableViewList

    **Bases:** object

    **This class defined a (simplified) list of views.**

    **Short demonstration:**

```python
>>> from benker.cell import Cell
>>> from benker.table import Table
>>> from benker.table import ColView
>>> from benker.table import RowView
>>> from benker.table import TableViewList

>>> red = Cell('red', x=1, y=1, height=2)
>>> pink = Cell('pink', x=2, y=1, width=2)
>>> blue = Cell('blue', x=2, y=2)
>>> table = Table([red, pink, blue])
>>> print(table)
+-----------+-----------------------+
| red | pink |
| +-----------+-----------------------+
| | blue |
| +-----------------------+-----------+
```

(continues on next page)
>>> cols = TableViewList(table, ColView)
>>> len(cols)
3
>>> rows = TableViewList(table, RowView)
>>> len(rows)
2

>>> for pos, col in enumerate(cols, 1):
...     print("col #\{pos\}: \{col\}".format(pos=pos, col=str(col)))
...     col #1: [red]
...     col #2: [pink, blue]
...     col #3: []

>>> cols[3].insert_cell("yellow")
>>> print(table)
+-----------+-----------------------+
| red | pink |
| +-----------+-----------+
| | blue | yellow |
+-----------+-----------+-----------+

adopt_cell(cell)
Adopt a new cell in the views.

Parameters cell (benker.cell.Cell) – New cell to adopt

refresh_all()
Cleanup and refresh all the views, taking into account the cells which are in the table grid.

class benker.table.ViewsProperty(view_cls)
Bases: object
Descriptor used to define the rows/cols properties in the class Table.

4.3.8 Available Parsers

This package contains a collection of parsers which you can use in conjunction with builders to convert tables from one format to another.

You can pick a builder in the Available Builders.

4.3.9 Base Parser

Base class of parsers.

class benker.parsers.base_parser.BaseParser(builder, encoding='utf-8', **options)
Bases: object
Abstract base class of the parsers classes.

parse_file(src_xml, dst_xml)
Parse and convert the tables from one format to another.

Parameters
• `src_xml` (str) – Source path of the XML file to convert.
• `dst_xml` (str) – Destination path of the XML file to produce.

`transform_tables(tree)`
benker.parsers.base_parser.value_of(element, xpath, namespaces=None, default=None)
Take the first value of a xpath evaluation.

Parameters
• `element` (etree._Element) – Root element used to evaluate the xpath expression.
• `xpath` (str) – xpath expression. This expression will be evaluated using the `namespaces`.
• `namespaces` (dict[str, str]) – Namespace map to use for the xpath evaluation.
• `default` – default value used if the xpath evaluation returns no result.

Returns the first result or the `default` value.

### 4.3.10 Office Open XML parser

This module can parse Office Open XML (OOXML) tables.

Specifications:
• The documentation about OOXML Table is available online at Word Processing - Table Grid/Column Definition.

benker.parsers.ooxml.NS = {'w': 'http://schemas.openxmlformats.org/wordprocessingml/2006/main'}
Namespace map used for xpath evaluation in Office Open XML documents

class benker.parsers.ooxml.OoxmlParser(builder, styles_path=None, **options)
    Bases: benker.parsers.base_parser.BaseParser
    Office Open XML parser.

    parse_grid_col(w_grid_col)
    Parse a `<w:gridCol>` element.

    See: Table Grid/Column Definition.

    Parameters `w_grid_col` (etree._Element) – Table element.

    parse_table(w_tbl)
    Convert a Office Open XML `<w:tbl>` into CALS `<table>`

    Parameters `w_tbl` (etree._Element) – Office Open XML element.

    Return type etree._Element
    Returns CALS element.

    parse_tbl(w_tbl)
    Parse a `<w:tbl>` element.

    See: Table Properties.

    Parameters `w_tbl` (etree._Element) – Table element.

    Changed in version 0.4.0: The section width and height are now stored in the `x-sect-size` table style (units in `pt`).
parse_tc(w_tc)
Parse a <w:tc> element.

See: Table Cell Properties.

Parameters w_tc(etree._Element) – Table element.

Changed in version 0.4.4: Improved empty cells detection for Formex4 conversion (<IE/> tag management).

parse_tr(w_tr)
Parse a <w:tr> element.

See: Table Row Properties.

Parameters w_tr(etree._Element) – Table element.

transform_tables(tree)
benker.parsers.ooxml.ns_name(ns, name)
benker.parsers.ooxml.w(name)

4.3.11 lxml Iterators

Python alternative to lxml.etree.iterwalk for lxml < 4.2.1

4.3.12 Available Builders

This package contains a collection of builders which you can use in conjunction with parsers to convert tables from one format to another.

You can pick a parser in the Available Parsers.

4.3.13 Base Builder

Base class of Builders.

class benker.builders.base_builder.BaseBuilder(**options)
Bases: object

Base class of Builders.

finalize_tree(tree)
Give the opportunity to finalize the resulting tree structure.

Parameters tree(etree._ElementTree) – The resulting tree.

New in version 0.4.0.

generate_table_tree(table)
Build the XML table from the Table instance.

Parameters table(benker.table.Table) – Table

Returns Table tree
4.3.14 CALS Builder

This module can construct a CALS table from an instance of type `Table`.

Specifications and examples:

- The CALS DTD is available online in the OASIS web site: CALS Table Model Document Type Definition.
- An example of CALS table is available in Wikipedia: CALS Table Model

```python
class benker.builders.cals.CalsBuilder(width_unit='mm', table_in_tgroup=False, **options)
    Bases: benker.builders.base_builder.BaseBuilder

    CALS table builder.

    build_cell(row_elem, cell)
        Build the CALS <entry> element.

        CALS attributes:
        - @colsep is built from the “border-right” style. Default value is “1” (displayed), so, it is better to always define it. This value is only set if different from the table @colsep value.
        - @rowsep is built from the “border-bottom” style. Default value is “1” (displayed), so, it is better to always define it. This value is only set if different from the table @rowsep value.
        - @valign is built from the “valign” style. Values can be “top”, “middle”, “bottom” (note: “baseline” is not supported). Default value is “bottom”.
        - @align is built from the “align” style. Values can be “left”, “center”, “right”, or “justify”. Default value is “left”. note: paragraphs alignment should be preferred to cells alignment.
        - @namest/@nameend are set when the cell is spanned horizontally.
        - @morerows is set when the cell is spanned vertically.

        Parameters
        - cell (benker.cell.Cell) – The cell.

    build_colspec(group_elem, col)
        Build the CALS <colspec> element.

        CALS attributes:
        - @colname is the column name. Its format is “c{col_pos}”.
        - @colwidth width of the column (with its unit). The unit is defined by the width_unit options.

    Note: The @colnum attribute (number of column) is not generated because this value is usually implied, and can be deduce from the @colname attribute.

    Parameters
    - group_elem (etree._Element) – Parent element: <tgroup>.
    - col (benker.table.ColView) – Columns
```
**build_row**(tbody_elem, row)

Build the CALS `<row>` element.

CALS attributes:

- @valign is built from the “valign” style. Values can be “top”, “middle”, “bottom” (note: “baseline” is not supported). Default value is “bottom”.

**Note:** A row can be marked as inserted if “x-ins” is defined in the row styles. Revision marks are inserted before and after a `<row>` using a couple of processing-instructions. We use the `<?change-start?>` PI to mark the start of the inserted row, and the `<?change-end?>` PI to mark the end.

**Parameters**

- tbody_elem (etree._Element) – Parent element: `<tbody>`, `<thead>` or `<tfoot>`.

**build_table**(table)

Build the CALS `<table>` element.

CALS attributes:

- @colsep is built from the “x-cell-border-right” style. Default value is “0” (not displayed).
- @rowsep is built from the “x-cell-border-bottom” style. Default value is “0” (not displayed).
- @tabstyle is built from the table nature.
- @orient is built from the “x-sect-orient” style (orientation of the current section). Possible values are “port” (portrait, the default) or “land” (landscape).
- @pgwide is built from the “x-sect-cols” style (column number of the current section). Default value is “0” (width of the current column).

**Note:** @colsep, @rowsep and @tabstyle attributes are generated only if the `table_in_tgroup` options is False.

**Attention:** According to the CALS specification, the default value for @colsep and @rowsep should be “1”. But, having this value as a default is really problematic for conversions: most of nowadays formats, like Office Open XML and CSS, consider that the default value is “no border” (a.k.a: `border: none`). So, setting “0” as a default value is a better choice.

**Parameters**

- table (benker.table.Table) – Table

**Returns** The newly-created `<table>` element.

**build_tbody**(group_elem, row_list, nature_tag)

Build the CALS `<tbody>`, `<thead>` or `<tfoot>` element.

**Parameters**

- group_elem (etree._Element) – Parent element: `<tgroup>`.
- row_list – List of rows
- nature_tag – name of the tag: ‘tbody’, ‘thead’ or ‘tfoot’.
**build_tgroup**(table_elem, table)

Build the CALS `<tgroup>` element.

CALS attributes:

- `@cols` is the total number of columns.
- `@colsep` is built from the “x-cell-border-right” style. Default value is “0” (not displayed).
- `@rowsep` is built from the “x-cell-border-bottom” style. Default value is “0” (not displayed).
- `@tgroupstyle` is built from the table nature.

**Note:** `@colsep`, `@rowsep` and `@tgroupstyle` attributes are generated only if the `table_in_tgroup` options is True.

---

**Parameters**

- `table` *(benker.table.Table)* – Table

**Returns** The newly-created `<tgroup>` element.

**generate_table_tree**(table)

Build the XML table from the Table instance.

**Parameters**

- `table` *(benker.table.Table)* – Table

**Returns** Table tree

---

**benker.builders.cals.revision_mark**(name, attrs)

---

### 4.3.15 Formex4 Builder

This module can construct a Formex4 table from an instance of type `Table`.

Formex describes the format for the exchange of data between the Publication Office and its contractors. In particular, it defines the logical markup for documents which are published in the different series of the Official Journal of the European Union.

This builder allow you to convert Word document tables into Formex4 tables using the Formex4 schema (formex-05.59-20170418.xd).

Specifications and examples:

- The Formex4 documentation and schema is available online in the Publication Office: Formex Version 4.
- An example of Formex4 table is available in the Schema documentation: TBL

**class** benker.builders.formex4.Formex4Builder(*detect_titles=False, **options*)

**Bases:** benker.builders.base_builder.BaseBuilder

Formex4 builder used to convert tables into TBL elements according to the TBL Schema

**build_cell**(row_elem, cell, row)

Build the Formex4 `<CELL>` element.

Formex4 attributes:

- `@COL` The mandatory COL attribute is used to specify in which column the cell is located.
• **@COLSPAN** When a cell in a row ‘A’ must be linked to a group of cells in the same row, the first CELL element of this group has to provide the COLSPAN attribute. The value of the COLSPAN attribute is the number of cells in the group. The COL attribute of the first cell indicates the number of the first column in the group.

The use of the COLSPAN attribute is only allowed to relate the value of a cell in several columns within the same row. Its value must be at least equal to ‘2’.

• **@ROWSPAN** When a cell in column ‘A’ is linked to a cell in row ‘B’ located just below row ‘A’, the CELL element of this single cell must provide the ROWSPAN attribute. The value of the ROWSPAN attribute is equal to the number of cells in the group. The CELL element relating to the single cell must be placed within the first ROW element in the group. The ROW elements corresponding to the other rows in the group may not contain any CELL elements for the column containing the single cell ‘A’.

The use of the ROWSPAN attribute is only authorised to relate the value of a cell in several rows. Its value must be at least equal to ‘2’.

• **@ACCH** If the group of related cells is physically delimited by a horizontal brace, this symbol must be marked up using the ACCH attribute.

• **@ACCV** If the group of related cells is physically delimited by a vertical brace, this symbol must be marked up using the ACCV attribute.

• **@TYPE** The TYPE attribute of the CELL element is used to indicate locally the type of contents of the cells. It overrides the value of the TYPE attribute defined for the row (ROW) which contains the cell.

Parameters

- **row_elem (etree._Element)** – Parent element: `<ROW>`.
- **cell (benker.cell.Cell)** – The cell.

Changed in version 0.4.4: Modification of the Formex4 builder to better deal with empty cells (management of `<IE/>` tags).

**build_corpus(tbl_elem, table)**

Build the Formex4 `<CORPUS>` element.

Parameters

- **tbl_elem (etree._Element)** – Parent element: `<TBL>`.
- **table (benker.table.Table)** – Table

Changed in version 0.4.3: If this option `detect_titles` is enable, a title will be generated if the first row contains an unique cell with centered text.

**build_row(corpus_elem, row)**

Build the Formex4 `<ROW>` element.

Formex4 attributes:

• **@TYPE** The TYPE attribute indicates the specific role of the row in the table. The allowed values are:
  - **ALIAS**: if the row contains aliases. Such references may be used when the table is included on several pages of a publication. The references are associated to column headers on the first page and are repeated on subsequent pages.
  - **HEADER**: if the row contains cells which may be considered as a column header. This generally occurs for the first row of a table.
– NORMAL: if most of the cells of the row contain ‘simple’ or ‘normal’ data. This is the default value.
– NOTCOL: if the cells of the row contain units of measure relating to subsequent rows.
– TOTAL: if the row contains data which could be considered as ‘totals’.

Note that this TYPE attribute is also provided for the cells (CELL), which could be used to override the value defined for the row. On the other hand, ‘NORMAL’ is the default value, so it is necessary to specify the TYPE attribute value in each cell of a row which has a specific type in order to avoid the default overriding (see the first row of the example below).

**Parameters**


**build_tbl**(*table*)

Build the Formex4 <TBL> element.

Formex4 attributes:

- @NO.SEQ This mandatory attribute provides a sequence number to the table. This number represents the order in which the table appears in the document.
- @CLASS The CLASS attribute is mandatory and is used to specify the type of data contained in the table. The allowed values are:
  - GEN: if the table contains general data (default value),
  - SCHEDULE: if it is a schedule,
  - RECAP: if it is a synoptic table.

These two last values are only used for documents related to the general budget.
- @COLS This mandatory attribute provides the actual number of columns of the table.
- @PAGE.SIZE The PAGE.SIZE attribute takes one of these values:
  - DOUBLE.LANDSCAPE: table on two A4 pages forming an A3 landscape page,
  - DOUBLE.PORTRAIT: table on two A4 pages forming an A3 portrait page,
  - SINGLE.LANDSCAPE: table on a single A4 page in landscape,
  - SINGLE.PORTRAIT: table on a single A4 page in portrait (default).

**Parameters**

- table (benker.table.Table) – Table

**Returns**

The newly-created <TBL> element.

**build_title**(tbl_elem, row)

Build the table title using the <TITLE> element.

For instance:

```
<TITLE>
<TI>
  <P>Table IV</P>
</TI>
</TITLE>
```
Parameters

• **tbl_elem** (*etree._Element*) – Parent element: `<TBL>`.
• **row** (*benker.table.RowView*) – The row which contains the title.

Changed in version 0.4.4: Modification of the Formex4 builder to better deal with empty cells (management of `<IE/>` tags).

**finalize_tree** (*tree*)

Finalize the resulting tree structure: calculate the @NO_SEQ values: sequence number of each table.

Parameters

**tree** (*etree._ElementTree*) – The resulting tree.

**generate_table_tree** (*table*)

Build the XML table from the Table instance.

Parameters

**table** (*benker.table.Table*) – Table

Returns Table tree

benker.builders.formex4 revision_mark (*name, attrs*)

4.3.16 Available Converters

This package contains a collection of converters which you can use to convert tables from one format to another.

To do that, you need to select a **parser** and a **builder**:

• You can pick a parser in the Available Parsers.
• You can pick a builder in the Available Builders.

4.3.17 Base Converter

Bas class of all converters.

```python
class benker.converters.base_converter.BaseConverter
    Bases: object
    Bas class of all converters.

    builder_cls
        alias of benker.builders.base_builder.BaseBuilder

    convert_file(src_xml, dst_xml, **options)
        Convert the tables from one format to another.
```

Parameters

• **src_xml** (*str*) – Source path of the XML file to convert.
• **dst_xml** (*str*) – Destination path of the XML file to produce.
• **options** – Dictionary of parsing/building options.

Common parsing options:

• **encoding** (default: “utf-8”): XML encoding of the destination file.
• **OOXML parser options**:
  - **styles_path** (default: `None`): Path to the stylesheet to use to resolve table styles. In an uncompressed .docx tree structure, the stylesheet path is `word/styles.xml`.
Common builder options: (none)

CALS builder options:


- **table_in_tgroup** (default: False): Where should we put the table properties:
  - False to insert the attributes @colsep, @rowsep, and @tabstyle in the <table> element.
  - True to insert the attributes @colsep, @rowsep, and @tgroupstyle in the <tgroup> element.

Formex4 builder options: (none)

```
parser_cls
alias of benker.parsers.base_parser.BaseParser
```

### 4.3.18 Office Open XML to CALS converter

class benker.converters.ooxml2cals.Ooxml2CalsConverter
    Bases: benker.converters.base_converter.BaseConverter
    Office Open XML to CALS converter

    builder_cls
    alias of benker.builders.cals.CalsBuilder

    parser_cls
    alias of benker.parsers.ooxml.OoxmlParser

    benker.converters.ooxml2cals.convert_ooxml2cals(src_xml, dst_xml, **options)
    Convert Office Open XML (OOXML) tables to CALS tables.

    Parameters
    
    - **src_xml** (str) – Source path of the XML file to convert.
      This must be an XML file, for instance, if you want to convert a Word document (.docx), you first need to unzip the .docx file, and then, run this function on the file word/document.xml. You can also use the styles_path option to parse and use the table styles defined in the file word/styles.xml.
    
    - **dst_xml** (str) – Destination path of the XML file to produce.
    
    - **options** – Extra conversion options. See convert_file() to have a list of all possible options.

### 4.3.19 Office Open XML to Formex4 converter

class benker.converters.ooxml2formex4.Ooxml2Formex4Converter
    Bases: benker.converters.base_converter.BaseConverter
    Office Open XML to Formex4 converter

    builder_cls
    alias of benker.builders.formex4.Formex4Builder
parser_cls
   alias of benker.parsers.ooxml.OoxmlParser

benker.converters.ooxml2formex4.convert_ooxml2formex4(src_xml, dst_xml, **options)
   Convert Office Open XML (OOXML) tables to Formex4 tables.

   Parameters
   • src_xml (str) – Source path of the XML file to convert.
     This must be an XML file, for instance, if you want to convert a Word document (.docx),
     you first need to unzip the .docx file, and then, run this function on the file word/document.xml. You can also use the styles_path option to parse and use the table styles
defined in the file word/styles.xml.
   • dst_xml (str) – Destination path of the XML file to produce.
   • options – Extra conversion options. See convert_file() to have a list of all possible
     options.

4.3.20 Alphabet

Utility functions to convert integer into a base-26 “number”, and vice versa.

benker.alphabet.alphabet_to_int(letters, alphabet='ABCDEFGHIJKLMNOPQRSTUVWXYZ')
   Convert a base-26 “number” using uppercase ASCII letters into an integer.

   >>> from benker.alphabet import alphabet_to_int
   >>> alphabet_to_int("A")
   1
   >>> alphabet_to_int("B")
   2
   >>> alphabet_to_int("AA")
   27
   >>> alphabet_to_int("AB")
   28
   >>> alphabet_to_int("ZZZ")
   18278
   >>> alphabet_to_int(""")
   0
   >>> alphabet_to_int("AA@")
   Traceback (most recent call last):
   ... ValueError: AA@

   Parameters
   • letters – string representing a “number” in the base-26.
   • alphabet – alphabet to use for the conversion.

   Returns Integer value of the “number”.

benker.alphabet.int_to_alphabet(value, alphabet='ABCDEFGHIJKLMNOPQRSTUVWXYZ')
   Convert a non-nul integer into a base-26 “number” using uppercase ASCII letters.

   Usage:
```python
>>> from benker.alphabet import int_to_alphabet

>>> int_to_alphabet(1)
'A'

>>> int_to_alphabet(2)
'B'

>>> int_to_alphabet(26)
'Z'

>>> int_to_alphabet(27)
'AA'

>>> int_to_alphabet(28)
'AB'

>>> int_to_alphabet(52)
'AZ'

>>> int_to_alphabet(53)
'BA'

>>> int_to_alphabet(18278)
'ZZZ'

>>> int_to_alphabet(-5)
Traceback (most recent call last):
...
ValueError: -5
```

Parameters

- **value** (int) – value to convert
- **alphabet** – alphabet to use for the conversion.

Returns

string representing this “number” in the base-26.

### 4.3.21 Drawing

Utility functions used to draw a Grid.

**Benker.drawing.TILES** = {
    (False, False, False, False): ' \n XXXXXXXXX \n',
    (False, False, False, True): ' \n XXXXXXXXX \n----------------\n',
    (False, False, True, False): ' \n XXXXXXXXX |
',
    (False, False, True, True): ' \n XXXXXXXXX |
----------------+
',
    (False, True, False, False): ' \n----------------
 XXXXXXXXX \n',
    (False, True, False, True): ' \n----------------
 XXXXXXXXX \n------------
',
    (False, True, True, False): ' \n------------
 XXXXXXXXX |
',
    (False, True, True, True): ' \n------------
 XXXXXXXXX |
------------
',
    (True, False, False, False): ' \n+-----------
 XXXXXXXXX 
',
    (True, False, False, True): ' \n+-----------
 XXXXXXXXX 
------------
',
    (True, False, True, False): ' \n+-----------
 XXXXXXXXX |
',
    (True, False, True, True): ' \n+-----------
 XXXXXXXXX |
+-----------
',
    (True, True, False, False): ' \n+-----------
 XXXXXXXXX |
',
    (True, True, False, True): ' \n+-----------
 XXXXXXXXX |
+-----------+
',
    (True, True, True, False): ' \n+-----------+
 XXXXXXXXX |
',
    (True, True, True, True): ' \n+-----------+
 XXXXXXXXX |
+-----------+
'
}

Default tiles used to draw a Grid.

Keys are tuples (left, top, right, bottom) : which represent the presence (if True) or absence (if False) : of the border. Values are the string representation of the tiles, “XXXXXXXXX” will be replaced by the cell content.

**Benker.drawing.draw(grid, tiles=None)**

Draw a grid using a collection of tiles.

### 4.3. API

61
Parameters

- **grid** *(benker.grid.Grid)* – Grid to draw.
- **tiles** – Collection of tiles, use `TILES` if not provided.

**Returns** String representation of the grid.

```python
tenker.drawing.iter_lines(grid, tiles=None)
tenker.drawing.iter_tiles(grid, tiles=None)
```

### 4.3.22 Units

Utility functions to convert values from one unit to another.

```python
Benker.units.UNITS = { 'cm': 0.01, 'dm': 0.1, 'ft': 0.3048, 'in': 0.0254, 'm': 1.0, 'mm': 0.001, 'pc': 0.0021166666666666664, 'pt': 0.00035277777777777776, 'px': 0.00035277777777777776}
```

**Usual units** Lengths in meter

```python
Benker.units.convert_value(value, unit_in, unit_out)
```

Convert a value from one unit to another.

To convert ‘1pt’ to ‘mm’, you can do:

```python
>>> from benker.units import convert_value

>>> convert_value(1, 'pt', 'mm')
0.353
```

**Parameters**

- **value** *(int or float)* – Value to convert
- **unit_in** – Current unit of the value.
- **unit_out** – Expected unit of the value.

**Return type** float

**Returns** The converted value

### 4.4 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.
4.4.1 v0.4.5 (unreleased)

Bug fix release

Fixed

Other

Change in the documentation: fix a broken and redirected links in the documentation.

4.4.2 v0.4.4 (2021-11-10)

Bug fix release

Fixed

Fix #13: ooxml2formex4 conversion – Loss of image calls in table conversion.
  • Modification of the OOXXML parser to improve empty cells detection for Formex4 conversion (<IE/> tags management).
  • Modification of the Formex4 builder to better deal with empty cells (management of <IE/> tags).

4.4.3 v0.4.3 (2019-10-15)

Bug fix release

Fixed

Fix #5: The title generation should be optional.
  • Change in the Formex4Builder class: Add the detect_titles option: if this option is enable, a title will be generated if the first row contains an unique cell with centered text. The detect_titles options is disable by default.

Other

Change in the documentation: update the URL of the Formex documentation, use: https://op.europa.eu/en/web/eu-vocabularies/formex/.

4.4.4 v0.4.2 (2019-06-06)

Bug fix release
Fixed

Fix #1: Cell nature should inherit row nature by default.

- Change in the class *Styled*: The default value of the `nature` parameter is `None` (instead of “body”).
- Change in the methods `insert_cell()` and `insert_cell()` The `nature` of a cell is inherited from its parent’s row (or column).

Other

- Change the requirements for Sphinx: add ‘requests[security]’ for Python 2.7.
- Fix an issue with the AppVeyor build: change the Tox configuration: set `py27,py34,py35: pip >= 9.0.3`.

4.4.5 v0.4.1 (2019-04-24)

Bug fix release

Fixed

- Change in the parser *OoxmlParser*: fix the ‘x-sect-cols’ value extraction when the `w:sectPr` is missing (use “1” by default).
- Fix the Formex4 builder *Formex4Builder*: Generate a `<IE/>` element if the cell content (the string representation) is empty.

4.4.6 v0.4.0 (2019-04-23)

Feature release

Added

- New converter: `convert_ooxml2formex4()`: Convert Office Open XML (OOXML) tables to Formex4 tables.
- New builder: *Formex4Builder*: Formex4 builder used to convert tables into TBL elements.
- Change in the parser *OoxmlParser*:
  - The section width and height are now stored in the ‘x-sect-size’ table style (units in ‘pt’).
- Change in the builder *BaseBuilder*: Add the method `finalize_tree()`: Give the opportunity to finalize the resulting tree structure.
4.4.7 v0.3.0 (2019-02-16)

Feature release

**Added**

- Change in the parser `OoxmlParser`
  - Parse cell `w:tcPr/w:vAlign` values.
  - Parse paragraph alignments to calculate cell horizontal alignments.
  - Parse cell `w:tcPr/w:tcBorders` values to extract border styles.
- Change in the builder `benker.builders.cals.CalsBuilder`
  - Generate `entry/@valign` attributes.
  - Generate `entry/@align` attributes.
  - Generate `entry/@colsep` and `entry/@rowsep` attributes.

**Changed**

- Change in the parser `OoxmlParser`
  - Add more supported border styles

4.4.8 v0.2.2 (2018-12-15)

Bug fix release

**Added**

- Add a Python alternative to `lxml.etree.iterwalk` if using `lxml < 4.2.1`. See `lxml changelog v4.2.1`.

**Fixed**

- Fix the implementation of `parse_table()`: use a new implementation of `lxml.etree.iterwalk` if using `lxml < 4.2.1`.

**Other**

- Change Tox configuration file to test the library with `lxml v3` and `v4`.
- Add a changelog in the documentation.
4.4.9 v0.2.1 (2018-11-27)

Fixed

- Fix Coverage configuration file.
- Fix and improve configuration for Tox.
- Fix docstring in ooxml2cals.
- Fix calculation of the @frame attribute in the method benker.builders.cals.CalsBuilder.build_table().

Other

- Change link to PyPi project to “https://pypi.org/project/Benker/”.
- Add the README to the documentation.
- Add configuration files for TravisCI and AppVeyor.

4.4.10 v0.2.0 (2018-11-26)

Changed

- Update project configuration
- Add missing __init__.py file in tests directory: it is required for test modules import.

Fixed

- Fix unit tests (Python 2.7).
- Fix flakes8 problems.
- Fix implementation of the Grid class for Python 2.7 (remove annotation). And minor fixes.
- Remove pipenv configuration files.
- Fix project configuration.

4.4.11 v0.1.0 (2018-11-26)

- First version of Benker.
CHAPTER

FIVE

INDICES AND TABLES

• genindex
• modindex
• search
b

benker, 33
benker.alphabet, 60
benker.box, 36
benker.builders, 52
benker.builders.base_builder, 52
benker.builders.cals, 52
benker.builders.formex4, 55
benker.cell, 39
benker.converters, 58
benker.converters.base_converter, 58
benker.converters.ooxml2cals, 59
benker.converters.ooxml2formex4, 59
benker.coord, 34
benker.drawing, 61
benker.grid, 43
benker.parsers, 50
benker.parsers.base_parser, 50
benker.parsers.lxml_iterwalk, 52
benker.parsers.ooxml, 51
benker.size, 33
benkerstyled, 38
benker.table, 45
benker.units, 62
INDEX

A
adopter_cell() (Benker tableView method), 49
adopter_cell() (Benker tableViewList method), 50
alphabet_to_int() (in module Benker alphabet), 60

B
BaseBuilder (class in Benker builders base builder), 52
BaseConverter (class in Benker converters base converter), 58
BaseParser (class in Benker parsers base parser), 50
Benker
   module, 33
Benker alphabet
      module, 60
Benker box
      module, 36
Benker builders
      module, 52
Benker builders base builder
      module, 52
Benker builders cals
      module, 52
Benker builders formex4
      module, 55
Benker cell
      module, 39
Benker converters
      module, 58
Benker converters base converter
      module, 58
Benker converters ooxml2cals
      module, 59
Benker converters ooxml2formex4
      module, 59
Benker coord
      module, 34
Benker drawing
      module, 61
Benker grid
      module, 43
Benker parsers
      module, 50
Benker parsers base parser
      module, 50
Benker parsers lxml_iterwalk
      module, 52
Benker parsers ooxml
      module, 51
Benker size
      module, 33
Benker styled
      module, 38
Benker table
      module, 45
Benker units
      module, 62
bounding_box (Benker grid Grid property), 44
bounding_box (Benker tableView Table property), 48
box (Benker cell Cell property), 41
Box (class in Benker box), 37
BoxTuple (class in Benker box), 38
build_cell() (Benker builders cals CalsBuilder method), 53
build_cell() (Benker builders formex4 Formex4Builder method), 55
build_colspec() (Benker builders cals CalsBuilder method), 53
build_corpus() (Benker builders formex4 Formex4Builder method), 56
build_row() (Benker builders cals CalsBuilder method), 53
build_row() (Benker builders formex4 Formex4Builder method), 55
build_table() (Benker builders cals CalsBuilder method), 56
build_tbl() (Benker builders formex4 Formex4Builder method), 54
build_tbody() (Benker builders cals CalsBuilder method), 54
build_tgroup() (Benker builders cals CalsBuilder method), 54
build_title() (Benker builders formex4 Formex4Builder method), 57
builder_cls (Benker converters base converter BaseConverter method), 57
move_to() (benker.box.Box method), 38
move_to() (benker.cell.Cell method), 41

N
nature (benker.styled.Styled attribute), 39
nature (benker.table.Table attribute), 48
NS (in module benker.parsers.ooxml), 51
ns_name() (in module benker.parsers.ooxml), 52

O
Oxml2CalsConverter (class in benker.converters.ooxml2cals), 59
Oxml2Formex4Converter (class in benker.converters.ooxml2formex4), 59
OxmlParser (class in benker.parsers.ooxml), 51
owned_cells (class in benker.tableTableView property), 49

P
parse_file() (benker.parsers.base_parser.BaseParser method), 50
parse_grid_col() (benker.parsers.ooxml.OxmlParser method), 51
parse_table() (benker.parsers.ooxml.OxmlParser method), 51
parse_tbl() (benker.parsers.ooxml.OxmlParser method), 51
parser_cls (benker.converters.base_converter.BaseConverter attribute), 59
parser_cls (benker.converters.ooxml2cals.Oxml2CalsConverter attribute), 59
parser_cls (benker.converters.ooxml2formex4.Oxml2Formex4Converter attribute), 59

R
refresh_all() (benker.table TableViewList method), 50
resize() (benker.box.Box method), 38
resize() (benker.cell.Cell method), 41
revision_mark() (in module benker.builders.cals), 55
revision_mark() (in module benker.builders.formex4), 58
row_pos (benker.table.RowView property), 47
rows (class in benker.tableTableView attribute), 49
RowView (class in benker.table), 46

S
size (benker.box.Box property), 38
size (benker.cell.Cell property), 41
Size (class in benker.size), 33
SizeTuple (class in benker.size), 34

Stylist (class in benker.styled), 39
styles (benker.styled.Styled property), 39

T
table (class in benker.table), 47
TableView (class in benker.table), 49
TableViewList (class in benker.table), 49
TILES (in module benker.drawing), 61
transform() (benker.box.Box method), 38
transform() (benker.cell.Cell method), 42
transform_tables() (benker.parsers.base_parser.BaseParser method), 51
transform_tables() (benker.parsers.ooxml.OxmlParser method), 52

U
union() (benker.box.Box method), 38
UNITS (in module benker.units), 62

V
value_of() (in module benker.parsers.base_parser), 51
ViewsProperty (class in benker.table), 50

W
w() (in module benker.parsers.ooxml), 52
width (benker.box.Box property), 38
width (benker.cell.Cell property), 42
width (benker.size.SizeTuple attribute), 34

X
x (benker.coord.CoordTuple attribute), 35

Y
y (benker.coord.CoordTuple attribute), 36