xhtml2pdf Documentation

Release 0.1b3

xhtml2pdf

Oct 03, 2017

Contents

xhtml2pdf enables users to generate PDF documents from HTML content easily and with automated flow control such as pagination and keeping text together. The **Python module** can be used in any Python environment, including Django. The **Command line tool** is a stand-alone program that can be executed from the command line.

Contents:

CHAPTER 1

Installation

This is a typical Python library and is installed using pip

```
pip install xhtml2pdf
```

To obtain the latest experimental version that has **Python 3 support**, please use a prerelease

```
pip install --pre xhtml2pdf
```

Requirements

Python 2.7+. Only Python 3.4+ is tested and guaranteed to work.

All additional requirements are listed in `requirements.txt` file and are installed automatically using the `pip install xhtml2pdf` method.

Development environment

1. If you don't have it, install `pip`, the python package installer

   ```
   sudo easy_install pip
   ```

   For more information about `pip` refer to http://www.pip-installer.org/.

2. I will recommend using `virtualenv` for development. This is great to have separate environment for each project, keeping the dependencies for multiple projects separated

   ```
   sudo pip install virtualenv
   ```

   For more information about `virtualenv` refer to http://www.virtualenv.org/

3. Create virtualenv for the project. This can be inside the project directory, but cannot be under version control

   ```
   virtualenv --distribute xhtml2pdfenv --python=python2
   ```

4. Activate your virtualenv

   ```
   source xhtml2pdfenv/bin/activate
   ```

   Later to deactivate use

```
deactivate
```

5. Next step will be to install/upgrade dependencies from `requirements.txt` file

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

6. Run tests to check your configuration

```
nosetests --with-coverage
```

You should have a log with success status:

```
Ran 36 tests in 0.322s

OK
```

CHAPTER 2

Usage

Using with Python standalone

```
from xhtml2pdf import pisa             # import python module

# Define your data
sourceHtml = "<html><body><p>To PDF or not to PDF</p></body></html>"
outputFilename = "test.pdf"

# Utility function
def convertHtmlToPdf(sourceHtml, outputFilename):
    # open output file for writing (truncated binary)
    resultFile = open(outputFilename, "w+b")

    # convert HTML to PDF
    pisaStatus = pisa.CreatePDF(
            sourceHtml,                # the HTML to convert
            dest=resultFile)           # file handle to recieve result

    # close output file
    resultFile.close()                 # close output file

    # return True on success and False on errors
    return pisaStatus.err

# Main program
if __name__ == "__main__":
    pisa.showLogging()
    convertHtmlToPdf(sourceHtml, outputFilename)
```

This basic Python example will generate a test.pdf file with the text 'To PDF or not to PDF' in the top left of the page. In-memory files can be generated by using StringIO or cStringIO instead of the file open. Advanced options will be discussed later in this document.

Using xhtml2pdf in Django

To allow URL references to be resolved using Django's STATIC_URL and MEDIA_URL settings, xhtml2pdf allows users to specify a `link_callback` paramter to point to a function that converts relative URLs to absolute system paths.

```
import os
from django.conf import settings
from django.http import HttpResponse
from django.template import Context
from django.template.loader import get_template
from xhtml2pdf import pisa

def link_callback(uri, rel):
    """
    Convert HTML URIs to absolute system paths so xhtml2pdf can access those
    resources
    """
    # use short variable names
    sUrl = settings.STATIC_URL      # Typically /static/
    sRoot = settings.STATIC_ROOT    # Typically /home/userX/project_static/
    mUrl = settings.MEDIA_URL       # Typically /static/media/
    mRoot = settings.MEDIA_ROOT     # Typically /home/userX/project_static/media/

    # convert URIs to absolute system paths
    if uri.startswith(mUrl):
        path = os.path.join(mRoot, uri.replace(mUrl, ""))
    elif uri.startswith(sUrl):
        path = os.path.join(sRoot, uri.replace(sUrl, ""))
    else:
        return uri  # handle absolute uri (ie: http://some.tld/foo.png)

    # make sure that file exists
    if not os.path.isfile(path):
            raise Exception(
                'media URI must start with %s or %s' % (sUrl, mUrl)
            )
    return path
```

```
def render_pdf_view(request):
    template_path = 'user_printer.html'
    context = {'myvar': 'this is your template context'}
    # Create a Django response object, and specify content_type as pdf
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment; filename="report.pdf"'
    # find the template and render it.
    template = get_template(template_path)
    html = template.render(Context(context))

    # create a pdf
    pisaStatus = pisa.CreatePDF(
       html, dest=response, link_callback=link_callback)
    # if error then show some funy view
    if pisaStatus.err:
       return HttpResponse('We had some errors <pre>' + html + '</pre>')
    return response
```

You can see in action in `demo/djangoproject` folder

Using in Command line

xhtml2pdf also provides a convenient command line tool which you can use to convert HTML files to PDF documents using the command line.

```
$ xhtml2pdf test.html
```

This basic command will convert the content of test.html to PDF and save it to test.pdf. Advanced options will be described later in this document.

The `-s` option can be used to start the default PDF viewer after the conversion:

```
$ xhtml2pdf -s test.html
```

Advanced Command line tool options

Use `xhtml2pdf --help` to get started.

Converting HTML data

To generate a PDF from an HTML file called `test.html` call:

```
$ xhtml2pdf -s test.html
```

The resulting PDF will be called `test.pdf` (if this file is locked e.g. by the Adobe Reader it will be called `test-0.pdf` and so on). The `-s` option takes care that the PDF will be opened directly in the Operating Systems default viewer.

To convert more than one file you may use wildcard patterns like `*` and `?`:

```
$ xhtml2pdf "test/test-*.html"
```

You may also directly access pages from the internet:

```
$ xhtml2pdf -s http://www.xhtml2pdf.com/
```

Using special properties

If the conversion doesn't work as expected some more informations may be usefull. You may turn on the output of warnings adding `-w` or even the debugging output by using `-d`.

Another reason could be, that the parsing failed. Consider trying the `-xhtml` and `-html` options. `xhtml2pdf` uses the HTMLT5lib parser that offers two internal parsing modes: one for HTML and one for XHTML.

When generating the HTML output `xhtml2pdf` uses an internal default CSS definition (otherwise all tags would appear with no diffences). To get an impression of how this one looks like start `xhtml2pdf` like this:

```
$ xhtml2pdf --css-dump > xhtml2pdf-default.css
```

The CSS will be dumped into the file `xhtml2pdf-default.css`. You may modify this or even take a totaly self defined one and hand it in by using the `-css` option, e.g.:

```
$ xhtml2pdf --css=xhtml2pdf-default.css test.html
```

CHAPTER 3

Working with HTML

PDF vs. HTML

Before we discuss how to define page layouts with xhtml2pdf style sheets, it helps to understand some of the inherent differences between PDF and HTML.

PDF is specifically designed around pages of a specific width and height. PDF page elements (such as paragraphs, tables and images) are positioned at absolute (X,Y) coordinates.

Note: While true PDF files use (0,0) to denote the bottom left of a page, xhtml2pdf uses (0,0) to denote the top left of a page, partly to maintain similarity with the HTML coordinate system.

HTML, by itself, does not have the concept of pagination or of pages with a specific width and height. An HTML page can be as wide as your browser width (and even wider), and it can be as long as the page content itself. HTML page elements are positioned in relationship to each other and may change when the browser window is resized.

In order to bridge the differences between HTML and PDFs, xhtml2pdf makes use of the concept of **Pages** and **Frames**. Pages define the size, orientation and margins of pages. Frames are rectangular regions with in each page.

The **Frame location is specified in absolute (X,Y) coordinates**, while the **Frame content is used to flow HTML content using the relative positioning rules of HTML**. This is the essence from which the power of xhtml2pdf stems.

HTML content will start printing in the first available content frame. Once the first frame is filled up, the content will continue to print in the next frame, and the next, and so on, until the entire HTML content has been printed.

Defining Page Layouts

xhtml2pdf facilitates the conversion of HTML content into a PDF document by flowing the continuous HTML content through one or more pages using Pages and Frames. A page represents a page layout within a PDF document. A Frame represents a rectangular region within a page through which the HTML content will flow.

Pages

The **@page** object defines a **Page template** by defining the properties of a page, such as page type (letter or A4), page orientation (portrait or landscape), and page margins. The @page definition follows the style sheet convention of ordinary CSS style sheets:

```
<style>
    @page {
        size: letter landscape;
        margin: 2cm;
    }
</style>
```

Frames

The **@frame** object defines the position and size of rectangular region within a page. A @page object can hold one or more @frame objects. The @frame definition follows the style sheet convention of ordinary CSS style sheets:

Here's a definition of a page template with one Content Frame. It makes use of the Letter page size of 612pt x 792pt.

```
<style>
    @page {
        size: letter portrait;
        @frame content_frame {
            left: 50pt;
            width: 512pt;
            top: 50pt;
            height: 692pt;
            -pdf-frame-border: 1;    /* for debugging the layout */
        }
    }
</style>
```

This will result in the following page layout:

```
+-page----------------+
|                     |
|  +-content_frame-+  |
|  |               |  |
|  |               |  |
|  |               |  |
|  |               |  |
|  +---------------+  |
|                     |
+---------------------+
```

Note: To visualize the page layout you are defining, add the `-pdf-frame-border: 1;` property to your each of your @frame objects.

Static frames vs Content frames

xhtml2pdf uses the concept of **Static Frames** to define content that remains the same across different pages (like headers and footers), and uses **Content Frames** to position the to-be-converted HTML content.

Static Frames are defined through use of the @frame property `-pdf-frame-content`. Regular HTML content will not flow through Static Frames.

Content Frames are @frame objects without this property defined. Regular HTML content will flow through Content Frames.

Example with 2 Static Frames and 1 Content Frame

```
<html>
<head>
<style>
    @page {
        size: a4 portrait;
        @frame header_frame {           /* Static Frame */
            -pdf-frame-content: header_content;
            left: 50pt; width: 512pt; top: 50pt; height: 40pt;
        }
        @frame content_frame {          /* Content Frame */
            left: 50pt; width: 512pt; top: 90pt; height: 632pt;
        }
        @frame footer_frame {           /* Another static Frame */
            -pdf-frame-content: footer_content;
            left: 50pt; width: 512pt; top: 772pt; height: 20pt;
        }
    }
</style>
</head>

<body>
    <!-- Content for Static Frame 'header_frame' -->
    <div id="header_content">Lyrics-R-Us</div>

    <!-- Content for Static Frame 'footer_frame' -->
    <div id="footer_content">(c) - page <pdf:pagenumber>
        of <pdf:pagecount>
    </div>

    <!-- HTML Content -->
    To PDF or not to PDF
</body>
</html>
```

In the example above, the vendor-specific tags `<pdf:pagenumber>` and `<pdf:pagecount>` are used to display page numbers and the total page count. This example will produce the following PDF Document:

```
+-page------------------+
| +-header_frame------+ |
| | Lyrics-R-Us       | |
| +-------------------+ |
| +-content_frame-----+ |
| | To PDF or not to  | |
| | PDF               | |
| |                   | |
| |                   | |
| +-------------------+ |
| +-footer_frame------+ |
| | (c) - page 1 of 1 | |
| +-------------------+ |
+-----------------------+
```

```
# Developer's note:
# To avoid a problem where duplicate numbers are printed,
# make sure that these tags are immediately followed by a newline.
```

Flowing HTML content through Content Frames

Content frames are used to position the HTML content across multiple pages. HTML content will start printing in the first available Content Frame. Once the first frame is filled up, the content will continue to print in the next frame, and the next, and so on, until the entire HTML content has been printed. This concept is illustrated by the example below.

Example page template with a header, two columns, and a footer

```
<html>
<head>
<style>
    @page {
        size: letter portrait;
        @frame header_frame {           /* Static frame */
            -pdf-frame-content: header_content;
            left: 50pt; width: 512pt; top: 50pt; height: 40pt;
        }
        @frame col1_frame {             /* Content frame 1 */
            left: 44pt; width: 245pt; top: 90pt; height: 632pt;
        }
        @frame col2_frame {             /* Content frame 2 */
            left: 323pt; width: 245pt; top: 90pt; height: 632pt;
        }
        @frame footer_frame {           /* Static frame */
            -pdf-frame-content: footer_content;
            left: 50pt; width: 512pt; top: 772pt; height: 20pt;
        }
    }
</style>
<head>
<body>
    <div id="header_content">Lyrics-R-Us</div>
    <div id="footer_content">(c) - page <pdf:pagenumber>
        of <pdf:pagecount>
    </div>
    <p>Old MacDonald had a farm. EIEIO.</p>
    <p>And on that farm he had a cow. EIEIO.</p>
    <p>With a moo-moo here, and a moo-moo there.</p>
    <p>Here a moo, there a moo, everywhere a moo-moo.</p>
</body>
</html>
```

The HTML content will flow from Page1.Col1 to Page1.Col2 to Page2.Col1, etc. Here's what the resulting PDF document could look like:

```
+------------------------------+    +------------------------------+
| Lyrics-R-Us                  |    | Lyrics-R-Us                  |
|                              |    |                              |
| Old MacDonald   farm he had a |    | a moo-moo       everywhere a  |
| had a farm.     cow. EIEIO.  |    | there.          moo-moo.     |
| EIEIO.          With a moo-  |    | Here a moo,                  |
| and on that     moo here, and |    | there a moo,                 |
|                              |    |                              |
| (c) - page 1 of 2            |    | (c) - page 2 of 2            |
+------------------------------+    +------------------------------+
```

CHAPTER 4

Advanced concepts

Keeping text and tables together

You can prevent a block of text from being split across separate frames through the use of the vendor-specific `-pdf-keep-with-next` property.

Here's an example where paragraphs and tables are kept together until a 'separator paragraph' appears in the HTML content flow.

```
<style>
    table { -pdf-keep-with-next: true; }
    p { margin: 0; -pdf-keep-with-next: true; }
    p.separator { -pdf-keep-with-next: false; font-size: 6pt; }
</style>
  ...
<body>
    <p>Keep these lines</p>
    <table><tr><td>And this table</td></tr></table>
    <p>together in one frame</p>

    <p class="separator"> <p>

    <p>Keep these sets of lines</p>
    <p>may appear in a different frame</p>
    <p class="separator"> <p>
</body>
```

Named Page templates

Page templates can be named by providing the name after the @page keyword

```
@page my_page {
    margin: 40pt;
}
```

Switching between multiple Page templates

PDF documents sometimes requires a different page layout across different sections of the document. xhtml2pdf allows you to define multiple @page templates and a way to switch between them using the vendor-specific tag `<pdf:nexttemplate>`.

As an illustration, consider the following example for a title page with large 5cm margins and regular pages with regular 2cm margins.

```
<html>
<head>
<style>
    @page title_template { margin: 5cm; }
    @page regular_template { margin: 2cm; }
</style>
</head>

<body>
    <h1>Title Page</h1>
    This is a title page with a large 5cm margin.

    <!-- switch page templates -->
    <pdf:nexttemplate name="regular_template" />

    <h1>Chapter 1</h1>
    This is a regular page with a regular 2cm margin.
</body>
</html>
```

CHAPTER 5

How to run tests

This file describes how people should run the various test suites included with xhtml2pdf.

Unit tests

Running unit tests should be pretty intuitive to most python developers. xhtml2pdf uses the standard library's unittest library to write tests. As such, the following command should "just work":

```
nosetests
```

A few extra bells and whistles are available. Specifically, a .coveragerc file is included with this project, therefore running coverage reports should give you immediately useful information:

```
pip install coverage
nosetests --with-coverage
coverage html
x-www-browser htmlcov/index.html
```

The coverage percentage is currently pretty low - feel free to add extra tests!

Functional tests

xhtml2pdf ships with a functional tests suite. To see it in action, run the following commands from the xhtml2pdf directory:

```
python testrender/testrender.py
x-www-browser testrender/output/index.html
```

The suite renders a set of templates to pdf, then uses imagemagic (available on most unix-like systems) to convert the pdfs to png images, and finally creates a image of differences between the generated image and a reference image.

Image sets with a "difference score" of more than 0 are highlighted in red - this means the rendering library produced a bad result.

Caveats

Font rendering is a very tricky business. As such, the functional suite often creates "ghost differences" for some font renderings (the images look perfect for a human eye, but the computer gives them a bad score anyway).

To solve theses, you should try regenerating reference images on your particular system, so that the exact mechanism used on your platform are used in both cases:

```
python testrender/testrender.py --create-reference local_reference
python testrender/testrender.py --ref-dir local_reference
x-www-browser testrender/output/index.html
```

You can now happily hack away at the library, without any ghost images.

CHAPTER 6

Reference

Supported @page properties and values

Valid @page properties:

```
background-image
size
margin, margin-bottom, margin-left, margin-right, margin-top
```

Valid size syntax and values:

```
Syntax: @page { size: <type> <orientation>; }

Where <type> is one of:
a0 .. a6
b0 .. b6
elevenseventeen
legal
letter

And <orientation> is one of:
landscape
portrait

Defaults to:
size: a4 portrait;
```

Supported @frame properties:

Valid @frame properties.

```
bottom, top, height
left, right, width
margin, margin-bottom, margin-left, margin-right, margin-top
```

To avoid unexpected results, please only specify two out of three bottom/top/height properties, and two out of three left/right/width properties per @frame object.

Supported CSS properties

xhtml2pdf supports the following standard CSS properties

```
background-color
border-bottom-color, border-bottom-style, border-bottom-width
border-left-color, border-left-style, border-left-width
border-right-color, border-right-style, border-right-width
border-top-color, border-top-style, border-top-width
colordisplay
font-family, font-size, font-style, font-weight
height
line-height, list-style-type
margin-bottom, margin-left, margin-right, margin-top
padding-bottom, padding-left, padding-right, padding-top
page-break-after, page-break-before
size
text-align, text-decoration, text-indent
vertical-align
white-space
width
zoom
```

xhtml2pdf adds the following vendor-specific properties:

```
-pdf-frame-border
-pdf-frame-break
-pdf-frame-content
-pdf-keep-with-next
-pdf-next-page
-pdf-outline
-pdf-outline-level
-pdf-outline-open
-pdf-page-break
```

@page background-image

To add a watermark to the PDF, use the `background-image` property to specify a background image

```
@page {
    background-image: url('/path/to/pdf-background.jpg');
}
```

Create PDF

The main function of xhtml2pdf is called CreatePDF(). It offers the following arguments in this order:

- **src**: The source to be parsed. This can be a file handle or a `String` - or even better - a `Unicode` object.
- **dest**: The destination for the resulting PDF. This has to be a file object wich will not be closed by `CreatePDF`. (XXX allow file name?)
- **path**: The original file path or URL. This is needed to calculate relative paths of images and style sheets. (XXX calculate automatically from src?)

- **link_callback**: Handler for special file paths (see below).
- **debug**: ** DEPRECATED **
- **show_error_as_pdf**: Boolean that indicates that the errors will be dumped into a PDF. This is usefull if that is the only way to show the errors like in simple web applications.
- **default_css**: Here you can pass a default CSS definition in as a `String`. If set to `None` the predefined CSS of xhtml2pdf is used.
- **xhtml**: Boolean to force parsing the source as XHTML. By default the HTML5 parser tries to guess this.
- **encoding**: The encoding name of the source. By default this is guessed by the HTML5 parser. But HTML with no meta information this may not work an then this argument is helpfull.

Link callback

Images, backgrounds and stylesheets are loaded form an HTML document. Normaly `xhtml2pdf` expects these files to be found on the local drive. They may also be referenced relative to the original document. But the programmer might want to load form different kind of sources like the Internet via HTTP requests or from a database or anything else. Therefore you may define a `link_callback` that handles these reuests.

XXX

Web applications

XXX

Defaults

- The name of the first layout template is `body`, but you better leave the name empty for defining the default template (XXX May be changed in the future!)

Fonts

By default there is just a certain set of fonts available for PDF. Here is the complete list - and their repective alias names - `xhtml2pdf` knows by default (the names are not case sensitive):

- **Times-Roman**: Times New Roman, Times, Georgia, serif
- **Helvetica**: Arial, Verdana, Geneva, sansserif, sans
- **Courier**: Courier New, monospace, monospaced, mono
- **ZapfDingbats**
- **Symbol**

But you may also embed new font faces by using the `@font-face` keyword in CSS like this:

```
@font-face {
  font-family: Example, "Example Font";
  src: url(example.ttf);
}
```

The font-family property defines the names under which the embedded font will be known. src defines the place of the fonts source file. This can be a TrueType font or a Postscript font. The file name of the first has to end with .ttf the latter with one of .pfb or .afm. For Postscript font pass just one filename like <name>.afm or <name>.pfb, the missing one will be calculated automatically.

To define other shapes you may do like this:

```
/* Normal */
@font-face {
   font-family: DejaMono;
   src: url(font/DejaVuSansMono.ttf);
}

/* Bold */
@font-face {
   font-family: DejaMono;
   src: url(font/DejaVuSansMono-Bold.ttf);
   font-weight: bold;
}

/* Italic */
@font-face {
   font-family: DejaMono;
   src: url(font/DejaVuSansMono-Oblique.ttf);
   font-style: italic;
}

/* Bold and italic */
@font-face {
   font-family: DejaMono;
   src: url(font/DejaVuSansMono-BoldOblique.ttf);
   font-weight: bold;
   font-style: italic;
}
```

Outlines/ Bookmarks

PDF supports outlines (Adobe calls them "bookmarks"). By default xhtml2pdf defines the <h1> to <h6> tags to be shown in the outline. But you can specify exactly for every tag which outline behaviour it should have. Therefore you may want to use the following vendor specific styles:

- **-pdf-outline** set it to "true" if the block element should appear in the outline
- **-pdf-outline-level** set the value starting with "0" for the level on which the outline should appear. Missing predecessors are inserted automatically with the same name as the current outline
- -pdf-outline-open set to "true" if the outline should be shown uncollapsed

Example:

```
h1 {
  -pdf-outline: true;  -pdf-level: 0;
  -pdf-open: false;
}
```

Table of Contents

It is possible to automatically generate a Table of Contents (TOC) with `xhtml2pdf`. By default all headings from `<h1>` to `<h6>` will be inserted into that TOC. But you may change that behaviour by setting the CSS property `-pdf-outline` to `true` or `false`. To generate the TOC simply insert `<pdf:toc />` into your document. You then may modify the look of it by defining styles for the `pdf:toc` tag and the classes `pdftoc.pdftoclevel0` to `pdftoc.pdftoclevel5`. Here is a simple example for a nice looking CSS:

```
pdftoc {
    color: #666;
}
pdftoc.pdftoclevel0 {
    font-weight: bold;
    margin-top: 0.5em;
}
pdftoc.pdftoclevel1 {
    margin-left: 1em;
}
pdftoc.pdftoclevel2 {
    margin-left: 2em;
    font-style: italic;
}
```

Tables

Tables are supported but may behave a little different to the way you might expect them to do. These restriction are due to the underlying table mechanism of ReportLab.

- The main restriction is that table cells that are longer than one page lead to an error
- Tables can not float left or right and can not be inlined

Long cells

`xhtml2pdf` is not able to split table cells that are larger than the available space. To work around it you may define what should happen in this case. The `-pdf-keep-in-frame-mode` can be one of: “error”, “overflow”, “shrink”, “truncate”, where “shrink” is the default value.

```
table {    -pdf-keep-in-frame-mode: shrink;}
```

Cell widths

The table renderer is not able to adjust the width of the table automatically. Therefore you should explicitly set the width of the table and to the table rows or cells.

Headers

It is possible to repeat table rows if a page break occurs within a table. The number of repeated rows is passed in the property `repeat`. Example:

```
<table repeat="1">
  <tr><th>Column 1</th><th>...</th></tr>
  ...
</table>
```

Borders

Borders are supported. Use corresponding CSS styles.

Images

Size

By default JPG images are supported. If the Python Imaging Library (PIL) is installed the file types supported by it are available too. As mapping pixels to points is not trivial the images may appear bigger in the PDF as in the browser. To adjust this you may want to use the `zoom` style. Here is a small example:

```
img { zoom: 80%; }
```

Position/ floating

Since Reportlab Toolkit does not yet support the use of images within paragraphs, images are always rendered in a seperate paragraph. Therefore floating is not available yet.

Barcodes

You can embed barcodes automatically in a document. Various barcode formats are supported through the `type` property. If you want the original barcode text to be appeared on the document, simply add `humanreadable="1"`, otherwise simply omit this property. Some barcode formats have a checksum as an option and it will be on by default, set `checksum="0"` to override. Alignment is achieved through `align` property and available values are any of `"baseline","top","middle","bottom"` whereas default is `baseline`. Finally, bar width and height can be controlled through `barwidth` and `barheight` properties respectively.

```
<pdf:barcode value="BARCODE TEXT COMES HERE" type="code128" humanreadable="1" align=
↪"right" />
```

Custom Tags

`xhtml2pdf` provides some custom tags. They are all prefixed by the namespace identifier `pdf:`. As the HTML5 parser used by xhtml2pdf does not know about these specific tags it may be confused if they are without a block. To avoid problems you may condsider sourrounding them by `<div>` tags, like this:

```
<div>
   <pdf:toc />
</div>
```

Tag-Definitions

pdf:barcode

Creates a barcode.

pdf:pagenumber

Prints current page number. The argument "example" defines the space the page number will require e.g. "00".

pdf:pagecount

Prints total page count.

pdf:nexttemplate

Defines the template to be used on the next page. The name of the template is passed via the `name` property and refers to a `@page templateName` style definition:

```
<pdf:nexttemplate name="templateName">
```

pdf:nextpage

Create a new page after this position.

pdf:nextframe

Jump to next unused frame on the same page or to the first on a new page. You may not jump to a named frame.

pdf:spacer

Creates an object of a specific size.

pdf:toc

Creates a Table of Contents.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search