Tutorial

As a standalone program

Once you have WeasyPrint installed, you should have a `weasyprint` executable. Using it can be as simple as this:

```
weasyprint http://weasyprint.org /tmp/weasyprint-website.pdf
```

You may see warnings on *stderr* about unsupported CSS properties. See Command-line API for the details of all available options.

In particular, the `-s` option can add a filename for a user stylesheet. For quick experimentation however, you may not want to create a file. In bash or zsh, you can use the shell’s redirection instead:

```
weasyprint http://weasyprint.org /tmp/weasyprint-website.pdf \
    -s <(echo 'body { font-family: serif !important }')
```

If you have many documents to convert you may prefer using the Python API in long-lived processes to avoid paying the start-up costs every time.

Adjusting Document Dimensions

Currently, WeasyPrint does not provide support for adjusting page size or document margins via command-line flags. This is best accomplished with the CSS `@page` at-rule. Consider the following example:

```
@page {
  size: Letter; /* Change from the default size of A4 */
  margin: 2.5cm; /* Set margin on each page */
}
```

There is much more which can be achieved with the `@page` at-rule, such as page numbers, headers, etc. Read more about the page at-rule, and find an example here.

As a Python library

 Attention

Using WeasyPrint with untrusted HTML or untrusted CSS may lead to various security problems.

Quickstart

The Python version of the above example goes like this:

```
from weasyprint import HTML
HTML('http://weasyprint.org/').write_pdf('/tmp/weasyprint-website.pdf')
```

& or with the inline stylesheet:

```
from weasyprint import HTML, CSS
HTML('http://weasyprint.org/').write_pdf('/tmp/weasyprint-website.pdf',
    stylesheets=[CSS(string='body { font-family: serif !important }')])
```

Instantiating HTML and CSS objects

If you have a file name, an absolute URL or a readable file object, you can just pass it to `HTML` or `CSS` to create an instance. Alternatively, use a named argument so that no guessing is involved:

```
from weasyprint import HTML

HTML('../foo.html')  # Same as …
HTML(filename='../foo.html')

HTML('http://weasyprint.org')  # Same as …
HTML(url='http://weasyprint.org')

HTML(sys.stdin)  # Same as …
HTML(file_obj=sys.stdin)
```

If you have a byte string or Unicode string already in memory you can also pass that, although the argument must be named:

```
from weasyprint import HTML, CSS

# HTML('<h1>foo') would be filename
HTML(string='''
    <h1>The title</h1>
    <p>Content goes here
''')
CSS(string='@page { size: A3; margin: 1cm }')
```

If you have `@font-face` rules in your CSS, you have to create a `FontConfiguration` object:

```
from weasyprint import HTML, CSS
from weasyprint.fonts import FontConfiguration

font_config = FontConfiguration()
html = HTML(string='<h1>The title</h1>')
css = CSS(string='''
    @font-face {
        font-family: Gentium;
        src: url(http://example.com/fonts/Gentium.otf);
    }
    h1 { font-family: Gentium }''', font_config=font_config)
html.write_pdf(
    '/tmp/example.pdf', stylesheets=[css],
    font_config=font_config)
```

Rendering to a single file

Once you have a `HTML` object, call its `write_pdf()` or `write_png()` method to get the rendered document in a single PDF or PNG file.

Without arguments, these methods return a byte string in memory. If you pass a file name or a writable file object, they will write there directly instead. (**Warning:** with a filename, these methods will overwrite existing files silently.)

Individual pages, meta-data, other output formats, …

If you want more than a single PDF, the `render()` method gives you a `Document` object with access to individual `Page` objects. Thus you can get the number of pages, their size[1], the details of hyperlinks and bookmarks, etc. Documents also have `write_pdf()` and `write_png()` methods, and you can get a subset of the pages with `copy()`. Finally, for ultimate control, `paint()` individual pages anywhere on any type of cairo surface.

[1] Pages in the same document do not always have the same size.

See the Python API for details. A few random examples:

```
# Write odd and even pages separately:
#   Lists count from 0 but page numbers usually from 1
#   [::2] is a slice of even list indexes but odd-numbered pages.
document.copy(document.pages[::2]).write_pdf('odd_pages.pdf')
document.copy(document.pages[1::2]).write_pdf('even_pages.pdf')
```

```
# Write one PNG image per page:
for i, page in enumerate(document.pages):
    document.copy([page]).write_png('page_%s.png' % i)
```

```
# Some previous versions of WeasyPrint had a method like this:
def get_png_pages(document):
    """Yield (png_bytes, width, height) tuples."""
    for page in document.pages:
        yield document.copy([page]).write_png()
```

```
# Print the outline of the document.
# Output on http://www.w3.org/TR/CSS21/intro.html
#     1. Introduction to CSS 2.1 (page 2)
#       1. A brief CSS 2.1 tutorial for HTML (page 2)
#       2. A brief CSS 2.1 tutorial for XML (page 5)
#       3. The CSS 2.1 processing model (page 6)
#         1. The canvas (page 7)
#         2. CSS 2.1 addressing model (page 7)
#       4. CSS design principles (page 8)
def print_outline(bookmarks, indent=0):
    for i, bookmark in enumerate(bookmarks, 1):
        page = bookmark.destination[0]
        print('%s%d. %s (page %d)' % (
            ' ' * indent, i, bookmark.label.lstrip('0123456789. '), page))
        print_outline(bookmark.children, indent + 2)
print_outline(document.make_bookmark_tree())
```

```
# PostScript on standard output:
surface = cairo.PSSurface(sys.stdout, 1, 1)
context = cairo.Context(surface)
for page in document.pages:
    # 0.75 = 72 PostScript point per inch / 96 CSS pixel per inch
    surface.set_size(page.width * 0.75, page.height * 0.75)
    page.paint(context, scale=0.75)
    surface.show_page()
surface.finish()
```

URL fetchers

WeasyPrint goes through a *URL fetcher* to fetch external resources such as images or CSS stylesheets. The default fetcher can natively open file and HTTP URLs, but the HTTP client does not support advanced features like cookies or authentication. This can be worked-around by passing a custom `url_fetcher` callable to the `HTML` or `CSS` classes. It must have the same signature as `default_url_fetcher()` .

Custom fetchers can choose to handle some URLs and defer others to the default fetcher:

```
from weasyprint import default_url_fetcher, HTML

def my_fetcher(url):
    if url.startswith('graph:'):
        graph_data = map(float, url[6:].split(','))
        return dict(string=generate_graph(graph_data),
                    mime_type='image/png')
    return weasyprint.default_url_fetcher(url)

source = '<img src="graph:42,10.3,87">'
HTML(string=source, url_fetcher=my_fetcher).write_pdf('out.pdf')
```

Flask-WeasyPrint for Flask and Django-Weasyprint for Django both make use of a custom URL fetcher to integrate WeasyPrint and use the filesystem instead of a network call for static and media files.

A custom fetcher should be returning a `dict` with

- One of `string` (a `bytestring`) or `file_obj` (a file object).
- Optionally: `mime_type`, a MIME type extracted e.g. from a *Content-Type* header. If not provided, the type is guessed from the file extension in the URL.
- Optionally: `encoding`, a character encoding extracted e.g. from a *charset* parameter in a *Content-Type* header
- Optionally: `redirected_url`, the actual URL of the resource if there were e.g. HTTP redirects.
- Optionally: `filename`, the filename of the resource. Usually derived from the *filename* parameter in a *Content-Disposition* header

If a `file_obj` is given, the resource will be closed automatically by the function internally used by WeasyPrint to retrieve data.

Logging

Most errors (unsupported CSS property, missing image, ...) are not fatal and will not prevent a document from being rendered.

WeasyPrint uses the `logging` module from the Python standard library to log these errors and let you know about them. When WeasyPrint is launched in a terminal, logged messages will go to *stderr* by default. You can change that by configuring the `weasyprint` logger object:

```
import logging
logger = logging.getLogger('weasyprint')
logger.addHandler(logging.FileHandler('/path/to/weasyprint.log'))
```

The `weasyprint.progress` logger is used to report the rendering progress. It is useful to get feedback when WeasyPrint is launched in a terminal (using the `--verbose` or `--debug` option), or to give this feedback to end users when used as a library.

See the documentation of the `logging` module for details.

WeasyPrint Tools

WeasyPrint provides two very limited tools, helping users to play with WeasyPrint, test it, and understand how to use it as a library.

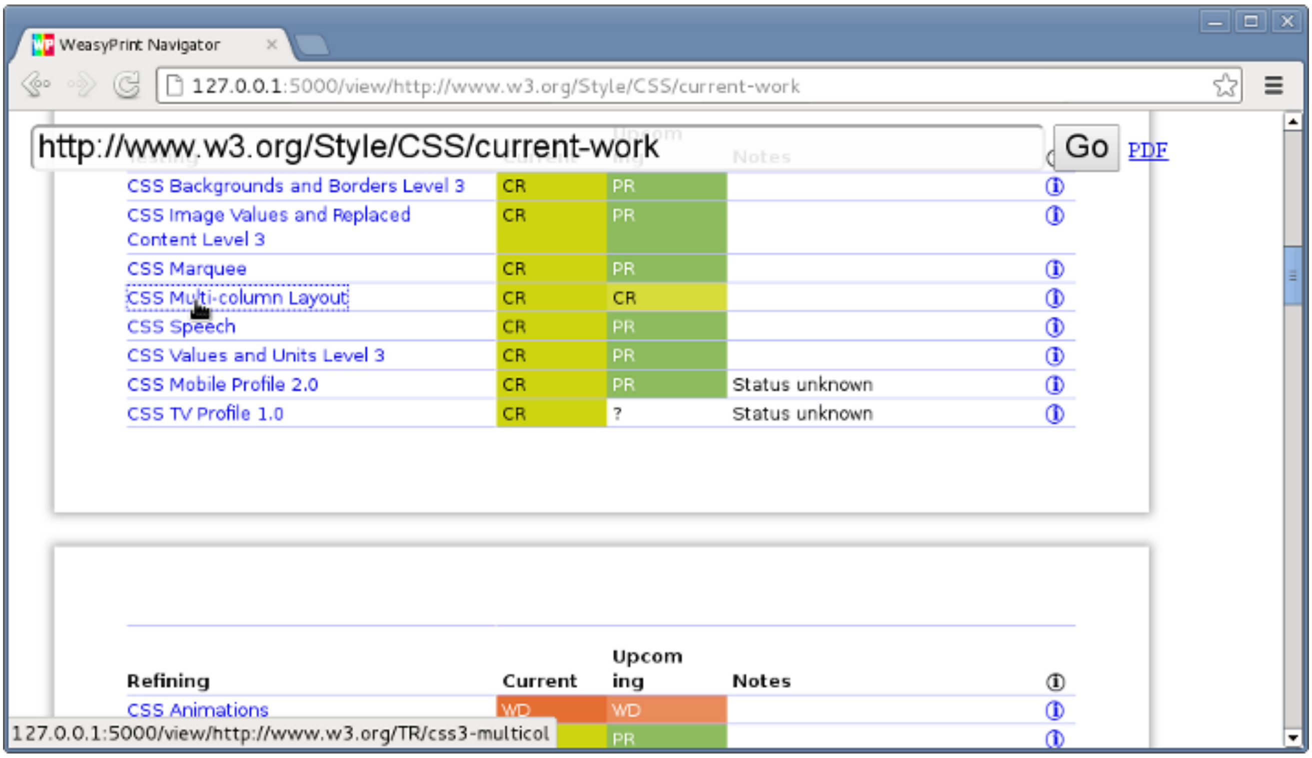
These tools are just “toys” and are not intended to be significantly improved in the future.

WeasyPrint Navigator

WeasyPrint Navigator is a web browser running in your web browser. Start it with:

```
python -m weasyprint.tools.navigator
```

& and open your browser at http://127.0.0.1:5000/.


WeasyPrint Navigator
127.0.0.1:5000/view/http://www.w3.org/Style/CSS/current-work
http://www.w3.org/Style/CSS/current-work
Go PDF

CSS Backgrounds and Borders Level 3	CR	PR		ⓘ
CSS Image Values and Replaced Content Level 3	CR	PR		ⓘ
CSS Marquee	CR	PR		ⓘ
CSS Multi-column Layout	CR	CR		ⓘ
CSS Speech	CR	PR		ⓘ
CSS Values and Units Level 3	CR	PR		ⓘ
CSS Mobile Profile 2.0	CR	PR	Status unknown	ⓘ
CSS TV Profile 1.0	CR	?	Status unknown	ⓘ

Refining	Current	Upcoming	Notes	ⓘ
CSS Animations	WD	WD		ⓘ
		PR		ⓘ

127.0.0.1:5000/view/http://www.w3.org/TR/css3-multicol

It does not support cookies, forms, or many other things that you would expect from a “real” browser. It only shows the PNG output from WeasyPrint with overlaid clickable hyperlinks. It is mostly useful for playing and testing.

WeasyPrint Renderer

WeasyPrint Renderer is a web app providing on the same web page a textarea where you can type an HTML/CSS document, and this document rendered by WeasyPrint as a PNG image. Start it with:

```
python -m weasyprint.tools.renderer
```

& and open your browser at http://127.0.0.1:5000/.

Security

When used with untrusted HTML or untrusted CSS, WeasyPrint can meet security problems. You will need extra configuration in your Python application to avoid high memory use, endless renderings or local files leaks.

This section has been added thanks to the very useful reports and advice from Raz Becker.

Long renderings

WeasyPrint is pretty slow and can take a long time to render long documents or specially crafted HTML pages.

When WeasyPrint used on a server with HTML or CSS files from untrusted sources, this problem can lead to very long time renderings, with processes with high CPU and memory use. Even small documents may lead to really long rendering times, restricting HTML document size is not enough.

If you use WeasyPrint on a server with HTML or CSS samples coming from untrusted users, you should:

" limit rendering time and memory use of your process, for example using `evil-reload-on-as` and `harakiri` options if you use uWSGI,

- limit memory use at the OS level, for example with `ulimit` on Linux,
- automatically kill the process when it uses too much memory or when the rendering time is too high, by regularly launching a script to do so if no better option is available,
- truncate and sanitize HTML and CSS input to avoid very long documents and access to external URLs.

Infinite requests

WeasyPrint can reach files on the network, for example using `http://` URIs. For various reasons, HTTP requests may take a long time and lead to problems similar to Long renderings.

WeasyPrint has a default timeout of 10 seconds for HTTP, HTTPS and FTP resources. This timeout has no effect with other protocols, including access to `file://` URIs.

If you use WeasyPrint on a server with HTML or CSS samples coming from untrusted users, or need to reach network resources, you should:

- use a custom URL fetcher,
- follow solutions listed in Long renderings.

Infinite loops

WeasyPrint has been hit by a large number of bugs, including infinite loops. Specially crafted HTML and CSS files can quite easily lead to infinite loops and infinite rendering times.

If you use WeasyPrint on a server with HTML or CSS samples coming from untrusted users, you should:

- follow solutions listed in Long renderings.

Huge values

WeasyPrint doesn’t restrict integer and float values used in CSS. Using huge values for some properties (page sizes, font sizes, block sizes) can lead to various problems, including infinite rendering times, huge PDF files, high memory use and crashes.

This problem is really hard to avoid. Even parsing CSS stylesheets and searching for huge values is not enough, as it is quite easy to trick CSS pre-processors using relative units (`em` and `%` for example).

If you use WeasyPrint on a server with HTML or CSS samples coming from untrusted users, you should:

- follow solutions listed in Long renderings.

Access to local files

As any web renderer, WeasyPrint can reach files on the local filesystem using `file://` URIs. These files can be shown in `img` or `embed` tags for example.

When WeasyPrint used on a server with HTML or CSS files from untrusted sources, this feature may be used to know if files are present on the server filesystem, and to embed them in generated documents.

Unix-like systems also have special local files with infinite size, like `/dev/urandom`. Referencing these files in HTML or CSS files obviously lead to infinite time rendering.

If you use WeasyPrint on a server with HTML or CSS samples coming from untrusted users, you should:

- restrict your process access to trusted files using sandboxing solutions,
- use a custom URL fetcher that doesn’t allow `file://` URLs or filters access depending on given paths.
- follow solutions listed in Long renderings.

System information leaks

WeasyPrint relies on many libraries that can leak hardware and software information. Even when this information looks useless, it can be used by attackers to exploit other security breaches.

Leaks can include (but are not restricted to):

- locally installed fonts (using `font-family` and `@font-face`),
- network configuration (IPv4 and IPv6 support, IP addressing, firewall configuration, using `http://` URIs and tracking time used to render documents),
- hardware and software used for graphical rendering (as Cairo renderings can change with CPU and GPU features),
- Python, Cairo, Pango and other libraries versions (implementation details lead to different renderings).

SVG images

WeasyPrint relies on CairoSVG to render SVG files. CairoSVG more or less suffers from the same problems as the ones listed here for WeasyPrint.

Security advices apply for untrusted SVG files as they apply for untrusted HTML and CSS documents.

Note that WeasyPrint gives CairoSVG its URL fetcher.

Errors

If you get an exception during rendering, it is probably a bug in WeasyPrint. Please copy the full traceback and report it on our issue tracker.

Stylesheet origins

HTML documents are rendered with stylesheets from three *origins*:

- The HTML5 user agent stylesheet (defines the default appearance of HTML elements);
- Author stylesheets embedded in the document in `<style>` elements or linked by `<link rel=stylesheet>` elements;

- User stylesheets provided in the API.

Keep in mind that *user* stylesheets have a lower priority than *author* stylesheets in the cascade, unless you use !important in declarations to raise their priority.