Colosseum Documentation

Release 0.2.0

Russell Keith-Magee

Sep 11, 2023

Contents

Colosseum is an independent implementation of the CSS layout algorithm. This implementation is completely standalone - it isn't dependent on a browser, and can be run over any box-like set of objects that need to be laid out on a page (either physical or virtual)

CHAPTER 1

Table of contents

1.1 Tutorial

Get started with a hands-on introduction for beginners

1.2 How-to guides

Guides and recipes for common problems and tasks, including how to contribute

1.3 Background

Explanation and discussion of key topics and concepts

1.4 Reference

Technical reference - commands, modules, classes, methods

CHAPTER 2

Community

Colosseum is part of the BeeWare suite. You can talk to the community through:

- @beeware@fosstodon.org on Mastodon
- Discord
- The Colosseum Github Discussions forum

We foster a welcoming and respectful community as described in our BeeWare Community Code of Conduct.

2.1 Tutorials

These tutorials are step-by step guides for using Colosseum.

2.1.1 Your first Colosseum layout

In a virtualenv, install Colosseum:

```
$ pip install colosseum
```

Colosseum provides a `CSS` class that allows you to define CSS properties and apply them to any DOM-like tree of objects. There is no required base class; Colosseum will duck-type any object providing the required API. The simplest possible DOM node is the following:

```
from colosseum.dimensions import Size, Box

class MyDOMNode:
    def __init__(self, style):
        self.parent = None
        self.children = []
        self.intrinsic = Size(self)
        self.layout = Box(self)
```

(continues on next page)

(continued from previous page)

```
        self.style = style.copy(self)

    def add(self, child):
        self.children.append(child)
        child.parent = self
```

That is, a node must provide:

- a `parent` attribute, declaring the parent in the DOM tree; the root of the DOM tree has a `parent` of `None`.
- a `children` attribute, containing the list of DOM child nodes.
- an `intrinsic` attribute, storing any intrinsic size hints for the node.
- a `layout` attribute, for storing the final position of the node.
- a `style` attribute - generally a `CSS` declaration, bound to the node.

We also need an area on to which we're going to render. This could be a screen, a page, a canvas, or any other rectangular content area. Again, colosseum will duck-type any class that has the required API:

```
class Page:
    def __init__(self, width, height):
        self.content_width = width
        self.content_height = height
```

With a compliant DOM node definition, you can create a content page, an instance of the DOM node class, and some children for that node:

```
>>> from colosseum import CSS
>>> from colosseum.constants import BLOCK
>>> page = Page(2000, 2000)
>>> node = MyDOMNode(style=CSS(display=BLOCK, width=1000, height=1000))
>>> node.add(MyDOMNode(style=CSS(display=BLOCK, width=100, height=200)))
>>> node.add(MyDOMNode(style=CSS(display=BLOCK, width=300, height=150)))
```

You can then ask for a layout to be computed, and query the results:

```
>>> from colosseum.engine import layout
>>> layout(page, node)
>>> print(node.layout)
<Box (1000x2000 @ 0,0)>
>>> node.layout.content_width
1000
>>> node.layout.content_height
2000
>>> node.layout.content_top
0
>>> node.layout.content_left
0
>>> for child in node.children:
...     print(child.layout)
<Box (100x200 @ 0,0)>
<Box (300x150 @ 0,200)>
```

Note that although the root node specifies a height of 1000, the computed content size is 2000 (i.e., the size of the display area). This matches how the root element is sized in a HTML5 document.

Style attributes can also be set in bulk, using the `update()` method on the style attribute:

```
>>> node.style.update(width=1500, height=800)
>>> layout(page, node)
>>> print(node.layout)
<Box (1500x2000 @ 0,0)>
```

Style attributes can also be removed by deleting the attribute on the style attribute. The value of the property will revert to the default:

```
>>> node.style.update(margin_top=10, margin_left=20)
>>> layout(page, node)
>>> print(node.layout)
<Box (1500x1990 @ 20,10)>
>>> del(node.style.margin_left)
>>> layout(page, node)
>>> print(node.style.margin_left)
0px
>>> print(node.layout)
<Box (1500x1990 @ 0,10)>
```

2.2 How-to Guides

How-to guides are recipes that take the user through steps in key subjects. They are more advanced than tutorials and assume a lot more about what the user already knows than tutorials do, and unlike documents in the tutorial they can stand alone.

2.2.1 How to get started

To use Colosseum, create a new virtual environment, and install it:

```
$ python3 -m venv venv
$ source venv/bin/activate.sh
(venv) $ pip install colosseum
```

You're now ready to use Colosseum! Your next step is to work through the *Tutorials*, which will take you step-by-step through your first steps and introduce you to the important concepts you need to become familiar with.

2.2.2 How to contribute to Colosseum

This guide will help walk you through the process of contributing your first contribution to Colosseum.

Set up your development environment

Go to the Colosseum repository on GitHub, and fork the repository into your own Github account.

Then, you need to set up your development environment. To work on Colosseum, you'll need Python 3.4+ installed on your computer. Create a virtual environment, and clone your Colosseum fork:

```
$ mkdir beeware
$ cd beeware
$ python3 -m venv venv
$ source venv/bin/activate
```

(continues on next page)

(continued from previous page)

```
$ git clone git@github.com:<your github username>/colosseum.git
$ cd colosseum
```

You can then run the test suite:

```
$ python setup.py test
```

This will run around 2300 tests - most of which are currently marked as "expected failures". This means that we have the test, but we *know* that they're failing at the moment, for reasons that we understand – usually, that we just haven't implemented the section of the CSS specification that the test is exercising.

You shouldn't *ever* get any **FAIL** or **ERROR** test results. We run our full test suite before merging every patch. If that process discovers any problems, we don't merge the patch. If you *do* find a test error or failure, either there's something odd in your test environment, or you've found an edge case that we haven't seen before - either way, let us know!

Now you are ready to start hacking on Colosseum. Have fun!

What should I do?

Having run the test suite, you now know which tests are currently expected failures. Your task is to take one of these tests, and make whatever modifications are necessary to get that test into a passing state, without breaking any other tests.

About the test suite

The bulk of Colosseum's test suite is automatically generated based on the W3C's Web Platform Tests. This is the canonical test suite for W3C standard compliance.

Unfortunately, this test suite is very browser centric, and isn't an automated test suite. It relies on a human loading a series of web pages, and comparing each page against a reference rendering. This is necessary in a web context, because browsers are allowed to have minor variations in rendering. However, it's a problem for Colosseum, for two reasons: firstly, because we want a fully automated test suite; but also because we're not targeting a web browser, so "load this web page" isn't a test we can use.

So - Colosseum's `utils` directory has a set of scripts that take the W3C repository, and converts each page into raw data. It uses Toga to create a 640x480 web view, loads the test document, and then injects some Javascript to extract the document structure and the rendering information about the document.

The output of this process is two documents - a *data* file, and a *reference* file.

The *data* file contains a JSON-formatted version of the DOM elements in the test, with their style declarations.

The *reference* file contains the same DOM structure - but instead of recording the style, it records the size, position, borders, padding and margin of each element in the DOM.

The test suite takes the data file, reconstructs the DOM nodes in Colosseum, and apply the test styles to those nodes. It then lays out the document, and evaluates if the size and position of each DOM node in the document matches that in the reference rendering.

Effectively, what we're doing is checking that given the same inputs, Colosseum will produce the same output as the web view - except that we don't ever have to draw anything on a screen. We're checking the layout at a basic mathematical level.

Of course, this assumes that the webview that was used to render the document is, itself, correct. The test suite was generated on a Mac using a Safari 11 Webkit webview, so the CSS2 components **should** be fully compliant; however, some CSS3 tests (especially in the flexbox and grid modules) may be incorrect.

Picking a test

All the tests are located under the tests directory. Each test directory contains a `not_implemented` file. This is a list of tests that are currently known to fail - because the logic necessary to make the test pass isn't implemented. Pick one of these tests, and delete it from the `not_implemented` file.

For the purposes of this example, we're going to pick the `block-formatting-contexts-006` test in `web_platform/CSS2/normal_flow/test_block_formatting_contexts` (this test is no longer available, as it has already been fixed).

The first test run

Each test file contains a single test class. In order to run the test, you must specify the path to this class, separated by dots. For our example, you can run the single test out of the test suite:

```
$ python setup.py test -s tests.web_platform.CSS2.normal_flow.test_block_formatting_
↪contexts.TestBlockFormattingContexts.test_block_formatting_contexts_006
```

This will produce a lot of console output. This output will start with some lines that tells you that the test is being set up and run:

```
running test
running egg_info
writing dependency_links to colosseum.egg-info/dependency_links.txt
writing colosseum.egg-info/PKG-INFO
writing top-level names to colosseum.egg-info/top_level.txt
reading manifest file 'colosseum.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
writing manifest file 'colosseum.egg-info/SOURCES.txt'
running build_ext
```

Then, it will output the name of the test being executed, and whether it passed or failed:

```
test_block_formatting_contexts_006 (tests.web_platform.CSS2.normal_flow.test_block_
↪formatting_contexts.TestBlockFormattingContexts)
When there is no padding or margins on elements the right outer edge of the child box␣
↪will touch the right edge of the containing block. ... FAIL
```

It then gives you a detailed description of **why** the test failed:

```
======================================================================
FAIL: test_block_formatting_contexts_006 (tests.web_platform.CSS2.normal_flow.test_
↪block_formatting_contexts.TestBlockFormattingContexts)
When there is no padding or margins on elements the right outer edge of the child box␣
↪will touch the right edge of the containing block.
----------------------------------------------------------------------
```

This description comes from the W3C test suite - it might point you in the right direction, or it might not.

You'll then see the stack trace at the point the test failed:

```
Traceback (most recent call last):
  File "/Users/rkm/projects/beeware/colosseum/tests/utils.py", line 360, in test_
↪method
    '\n' + '\n'.join(extra)
  File "/Users/rkm/projects/beeware/colosseum/tests/utils.py", line 277, in␣
↪assertLayout
```

(continues on next page)

(continued from previous page)

```
    self.fail('\n'.join(output))
AssertionError:
```

This will tell you the line of code where the test failed. However, as our test is automatically generated, this won't really tell you much, other than the fact that the test generation code is all in `tests/utils.py`.

What *is* helpful is the next piece of output:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    * <body> 624x96 @ (8, 8)
             padding: 624x96 @ (8, 8)
             border: 624x96 @ (8, 8)
        * <div> 96x96 @ (8, 8)
                padding: 96x96 @ (8, 8)
                border: 101x96 @ (8, 8)
            * <div> 91x96 @ (8, 8)
                    padding: 91x96 @ (8, 8)
>>                  padding: 96x96 @ (8, 8)
                    border: 96x96 @ (8, 8)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

This is a comparison between what was generated by Colosseum, and what was expected. In this example, the two documents are almost identical, except for the padding box of one element. Colosseum has determined that the padding box is 96 pixels wide (the line indicated with the >> marker); the reference rendering says it should be 91 pixels wide.

The test output then gives us some pointers for where to look in the standard for the rules that need to be followed:

```
See http://www.w3.org/TR/CSS21/visuren.html#block-formatting
```

and the W3C test itself:

```
Test: http://test.csswg.org/harness/test/css21_dev/single/block-formatting-contexts-
↪006/
```

lastly, as the test shuts down, we get some summary output for the test run:

```
----------------------------------------------------------------------
Ran 1 test in 0.005s

FAILED (failures=1)
Test failed: <unittest.runner.TextTestResult run=1 errors=0 failures=1>
error: Test failed: <unittest.runner.TextTestResult run=1 errors=0 failures=1>
```

This example shows a case where the code is running to completion without error, but is generating incorrect output. It's also possible that you might see errors caused when the code cannot run to completion. For example, in the following case, a `None` value has mistaken leaked into a padding box calculation:

```
  File "/Users/rkm/projects/beeware/colosseum/colosseum/dimensions.py", line 594, in␣
↪padding_box_width
    return self._padding_left + self._content_width + self._padding_right

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

You may also see exceptions that are raised specifically mentioning parts of the CSS specification that have not yet been implmemented. For example, this exception would be raised if the layout example requires the rules of Section 9.4.2 of the CSS2.2 specification, but that section hasn't been implemented:

```
  File "/Users/rkm/projects/beeware/colosseum/colosseum/engine.py", line 239, in␣
↪layout_box
    raise NotImplementedError("Section 9.4.2 - relative positioning")  # pragma: no␣
↪cover

NotImplementedError: Section 9.4.2 - relative positioning
```

Lastly, you might see errors where the rendering engine has become confused. In this example, a impossible branch of code has been reached while calculating a width of an element in normal flow:

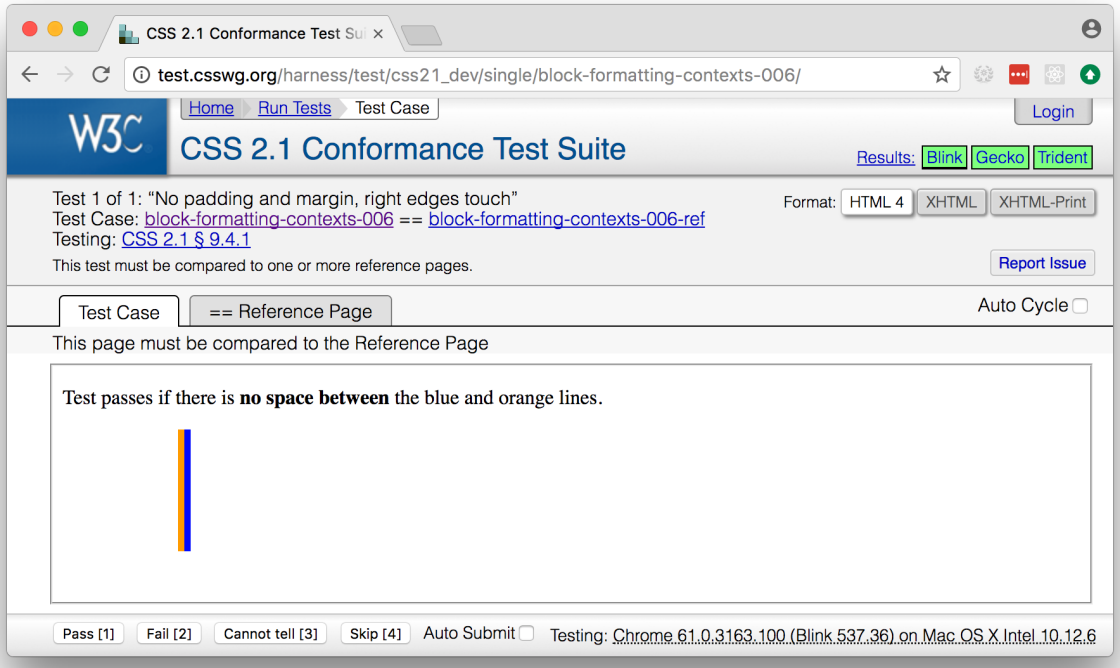
```
  File "/Users/rkm/projects/beeware/colosseum/colosseum/engine.py", line 284, in␣
↪calculate_width_and_margins
    raise Exception("Unknown normal flow width calculation")  # pragma: no cover

Exception: Unknown normal flow width calculation
```

Ok! So we now have a failing test. What do we do about it?

Is the test case correct?

Since the test suite is automatically generated, and there are over 2000 tests, we can't be 100% certain that the test *itself* is correct. So - we need to confirm whether the test itself is valid.

Click on the link to the W3C test that was in the test output. You should see a page that looks something like:



Using this view, confirm that the test acutally passes. In this case, the page tells us to compare to the reference page; you can flick between the "Test Case" tab and the "Reference Page" tab and confirm that the output is as expected.

Note: Many of the tests in the test suite use a special font, called "Ahem". Ahem is a font that has a limited character set, but known (and simple) geometries for each character - for example, the M glyph (used to establish the size of the "em" unit) is a solid black square. This helps make test results repeatable. You'll need to install this font before confirming the output of any test that uses it.

The reference rendering won't always be pixel perfect, so you'll need to check any text on the page to see whether the test is passing in the browser.

If the test appears to be failing, then there's no point trying to reproduce the browser's behavior in Colosseum! Look for a file called `not_compliant` in the same directory as the `not_implemented` file. If this file doesn't exist - create it. Then, add to the *not_compliant* file the same line that you *deleted* from `not_implemented`. Rerun the test - it should come back reporting as an expected failure:

```
running test
running egg_info
writing colosseum.egg-info/PKG-INFO
writing top-level names to colosseum.egg-info/top_level.txt
writing dependency_links to colosseum.egg-info/dependency_links.txt
reading manifest file 'colosseum.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
writing manifest file 'colosseum.egg-info/SOURCES.txt'
running build_ext
test_block_formatting_contexts_006 (tests.web_platform.CSS2.normal_flow.test_block_
↪formatting_contexts.TestBlockFormattingContexts)
When there is no padding or margins on elements the right outer edge of the child box␣
↪will touch the right edge of the containing block. ... expected failure

----------------------------------------------------------------------
Ran 1 test in 0.004s

OK (expected failures=1)
```

And you're done! You've just told the test suite that yes, the test will fail, but because the Webkit test result isn't correct.

Note: Most of the tests in the CSS test suite *should* pass. If you think you've found a failure in a CSS2 test, you should try and confirm with others before you submit your patch. You may find the W3C's test results helpful - these are results reported by other users.

Sometimes, the test will pass, but it will be validating something that Colosseum is not concerned with. For example, some of the tests deal with behavior during DOM manipulation (insertion or removal of elements from the DOM with JavaScript). DOM manipulation isn't something Colosseum is trying to model, so this test isn't of any use to us. In this case, you should move the test line from the `not_implemented` file to the `not_valid` file (again, you may need to create this file if it doesn't exist). This flags that it is a test that doesn't need to be executed at all.

If you find an invalid or non-compliant test, submit a pull request for the change moving the line out of the `not_implemented` file, and you're done! That pull request should detail *why* you believe the test in invalid, so that the person reviewing your pull request understands your reasoning. You can now pick another test, and work on your second pull request!

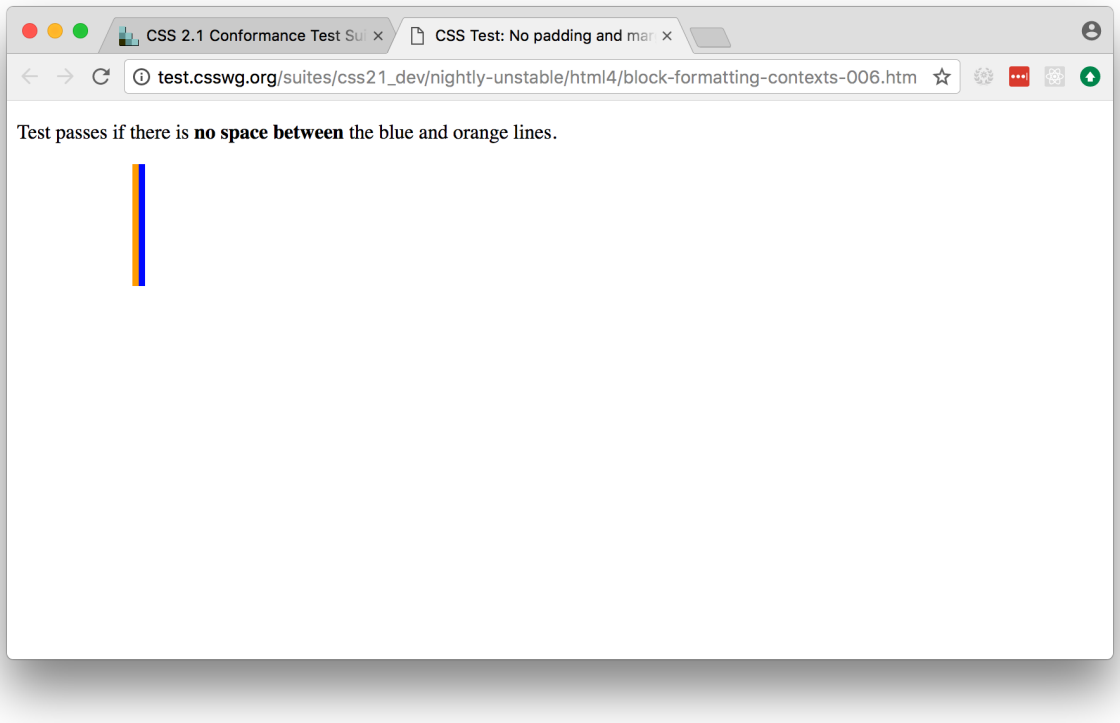
However, if the test passes, the next step is to try and fix it.

Note: If you're a newcomer to Colosseum, and the test you've chosen involves rendering text, displaying an image, or testing the color of an element (other than where color is used purely to make an element visible), you might want

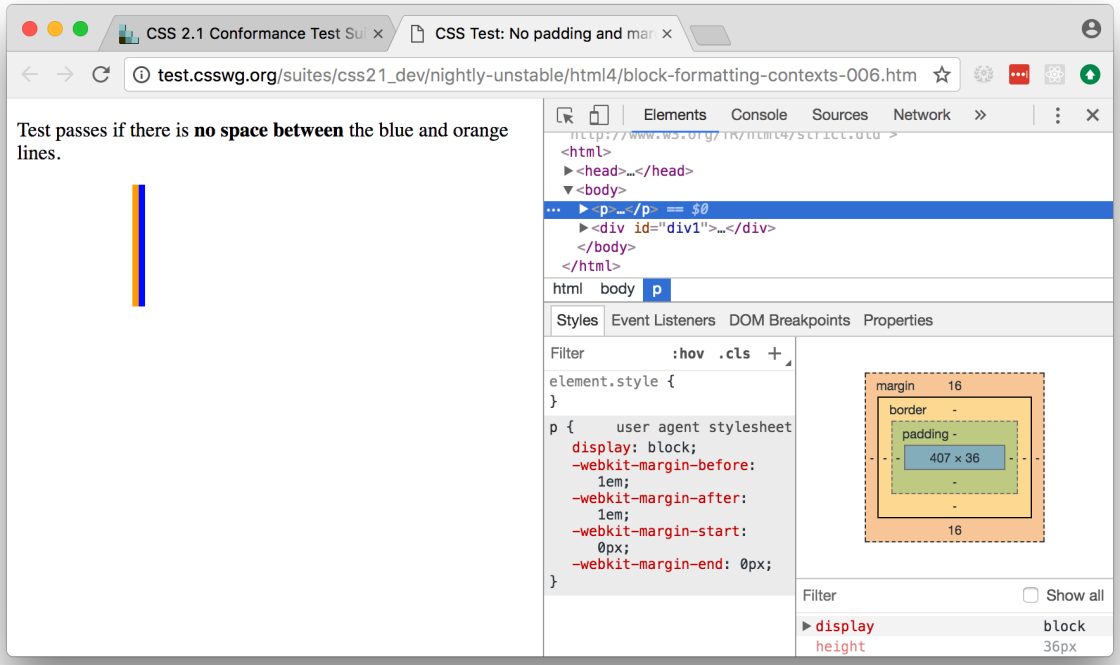
pick another test to work on. The easiest tests to work on will involve the positioning of boxes on a page, without any images or text.

The raw test

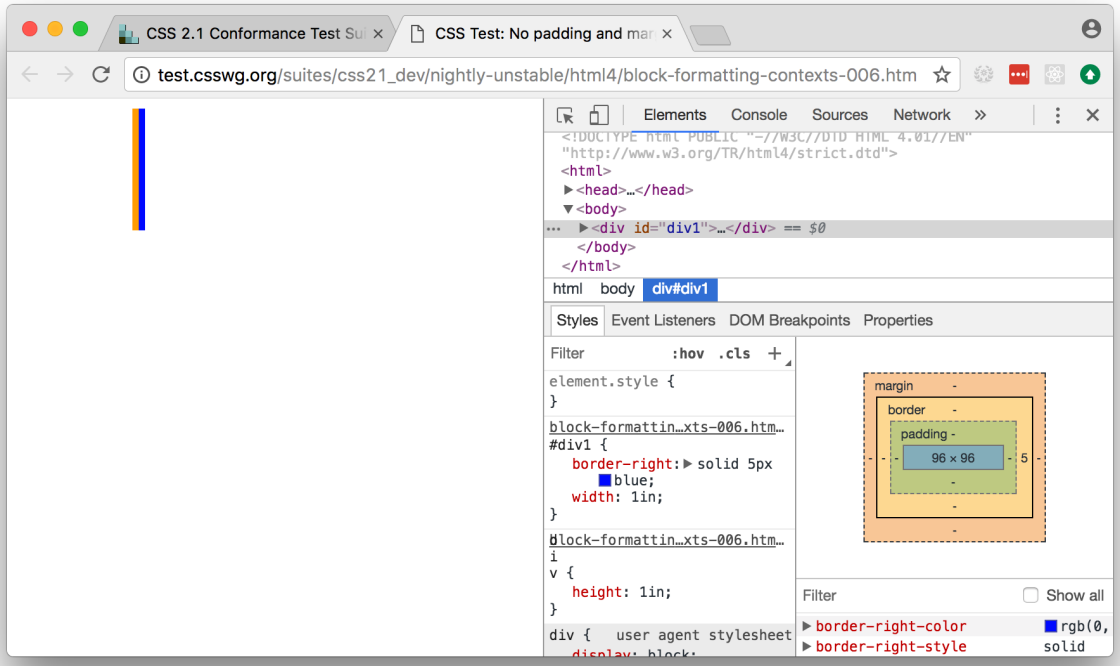
Near the top of the test suite page, there is a "Test Case:" label, followed by two links. These are links to the raw documents that are used in the test. If you click on the first link (the test document), you'll see a page that looks just like the test case, but without the test harness around it:



In this test, the raw test page is a line of test instructions. This won't exist on every test case; but if it *does* exist, we need to strip it out to simplify the test for our purposes. Open the web inspector, select the `<p>` element corresponding to the test instruction:



Right click on the element, and select “Delete element”. This will remove the instruction from the page:



Note: Sometimes, the test instruction *is* the test - for example, the test might read "This text should not be red". If this is the case, you *shouldn't* delete the instructional text. You only delete the instructional text if it is *purely* instructional - if it doesn't actually participate in the layout being tested.

Once you've deleted - for example, in the screenshot, you can see that the `<div>` element that is the child of the `<body>` should have a content size of 96x96, and a right border of 5 pixels. We can compare this to the output produced when we ran our test, and see that yes - during the test, the border box was 101x96, the inner content was 96x96, but both the border box and the inner content of that element had an origin of 8x8. This means the right border extended 5 pixels past the content.

We can now start digging into the code to see if we can identify why the margin box hasn't received the correct size.

Fixing the problem

At this point, you're in bug country! Every bug will have a slightly different cause, and it's your job to find out what is going on.

The entry point for rendering is the `layout()` method in `src/colosseum/engine.py`. This method calls `layout_box()` recursively on the document being rendered. The code in `src/colosseum/engine.py` is extensively documented with references to the CSS specification - especially the CSS2.2 Specification, the Flexible Box Layout Module, and the Grid Layout Module. Any changes you make should be include equally verbose documentation, and cross references to any paragraphs in the specification.

The test suite uses a `TestNode` as the basis for it's document model. A test node has three attributes of particular interest:

- `style`, storing the CSS style declaration that applies to the node. These values may be expressed in any units allowed by CSS (including pixels, points, em, percent, and more). The sub-attributes of the `style` attribute match those of the CSS specification (e.g., `width`, `margin_top`, `z_index`, and so on).
- `layout`, storing the computed values for the layout of the TestNode. These values are *always* in integer pixels. The layout describes the position of a content box (defined by `content_top`, `content_right`, `content_bottom` and `content_left`), relative to the content box of it's parent (with an offset defined by `offset_top` and `offset_left`). Surrounding the content box is a padding box, surrounded by a border box, surrounded by a margin box. These are also given in pixels, relative to the content box of the parent element.
- `children`, a list of TestNodes that are descendents of this node. A leaf node in the DOM tree is a node with an empty children list.

The layout algorithm roughly consists of:

1. Set up and copy over initial layout values by computing the style values.
2. Calculate the width of the node
3. Iterate over the children of the node
4. Calulate the height of the node
5. Make an adjustments for relative positioning.

Dig into the code, and work out why Colosseum is giving the wrong result.

Re-run the test suite

Once you've identified the problem, and the single test passes, you can re-run the *entire* Colosseum test suite. One of three things will happen:

1. The test suite will pass without any errors. In this case, you've fixed exactly one bug. Submit a pull request with your fix, and try another one!
2. The test suite will report one or more FAIL or ERROR results. In this case, you've fixed one bug, but broken existing behavior in the process. This means there's something subtle wrong with your fix. Go back to the code, and see if you can find a way to make your chosen test pass *without* breaking other tests.
3. The test suite will report one or more UNEXPECTED PASS results. This is good news - it means that the fix you've made has indirectly fixed one more more *other* tests! Quickly verify that those tests are valid (using the same process that you used to verify the test you *deliberately* fixed), and if they're valid tests, remove them from the `not_implemented` file. Submit a pull request with your fix, and try another one!

What if the test itself is incorrect?

Since the reference test data is automatically extracted from a running browser, and browsers don't provide a great API for extracting rendering data, it's entirely possible that the reference test data that the test is using might be incorrect. If you look at the test suite output, and it doesn't match what you see in a browser, open a ticket describing what you've found. This may indicate a bug in the reference rendering; which will require a fix to the script that generates the test data.

Add change information for release notes

Colosseum uses towncrier to automate building release notes. To support this, every pull request needs to have a corresponding file in the `changes/` directory that provides a short description of the change implemented by the pull request.

This description should be a high level summary of the change from the perspective of the user, not a deep technical description or implementation detail. It should also be written in past tense (i.e., "Added an option to enable X" or "Fixed handling of Y").

See News Fragments for more details on the types of news fragments you can add. You can also see existing examples of news fragments in the `changes/` folder.

2.3 Background

Want to know more about the Colosseum project, it's history, community, and plans for the future? That's what you'll find here!

2.3.1 Frequently Asked Questions

So... why the name Colosseum?

The Colosseum, also known as the Flavian Amphitheater, is an ancient Roman Amphitheater in the center of Rome. It is an astonding piece of ancient architecture, noted for it's three layers of arches, framed by Doric, Ionic and Corinthian half-columns, with an attic decorate with Corinthian pilasters.

Much like Doric, Ionic and Corithian columns form the fundamental architecture of the ancient Roman world, CSS is part of the fundamental architecture of modern display computing. The regular repeating structure of the Colosseum's arches and columns mirror the regular grid-based layout of many modern web and print designs.

The Colosseum was also a massive undertaking for it's time. Undertaking to reproduce the entire CSS specification, with all it's quirks and eccentricities, is a similarly massive undertaking.

But most importantly: the name Colosseum contains, in order, the letters CSS.

2.3.2 The Colosseum Developer and User community

Colosseum is part of the BeeWare suite. You can talk to the community through:

- @beeware@fosstodon.org on Mastodon
- Discord
- The Colosseum Github Discussions forum

Code of Conduct

The BeeWare community has a strict Code of Conduct. All users and developers are expected to adhere to this code.

If you have any concerns about this code of conduct, or you wish to report a violation of this code, please contact the project founder *Russell Keith- Magee*.

Contributing

If you experience problems with Colosseum, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

2.3.3 Release History

0.2.0

From scratch rewrite, using Travertino base classes.

v0.1.6

- Fixed the MANIFEST.in file.

v0.1.5

- Modify hints so that they only mark layout as modified if the value changes.

v0.1.4

- Made the dirty flag a tri-state variable, to allow for "layout in progress" tracking.

v0.1.3

- Added ability to extract absolute position of DOM nodes

v0.1.2

- Added first W3C-based test suite.
- Added hinting for style objects.
- Separated style definition from DOM definition.

v0.1.1

- Added protection against invalid values for string-based properties.
- Added support for bulk setting of attributes.

v0.1.0

Initial release of a version based on a port of Facebook's CSS-layout (now Yoga) project.

2.3.4 Colosseum Roadmap

2.4 Reference

This is the technical reference for public APIs provided by Colosseum.