WCAG-Zoo Documentation

Release 0.2.5

Samuel Spencer

Dec 28, 2017

Contents:

CHAPTER 1

API documentation

class wcag_zoo.utils.**WCAGCommand**(**args*, ***kwargs*)

The base class for all WCAG validation commands

classmethod as_cli()

Exposes the WCAG validator as a click-based command line interface tool.

check_skip_element(*node*)

Performs checking to see if an element can be skipped for validation, including check if it has an id or class to skip, or if it has a CSS rule to hide it.

THis class calls `WCAGCommand.skip_element` to get any additional skip logic, override `skip_element` not this method to add custom skip logic.

Returns True if the node is to be skipped.

run_validation_loop(*xpath=None*, *validator=None*)

Runs validation of elements that match an xpath using the given validation method. By default runs *self.validate_element*

skip_element(*node*)

Method for adding extra checks to determine if an HTML element should be skipped by the validation loop.

Override this to add custom skip logic to a wcag command.

Return true to skip validation of the given node.

validate_document(*html*)

Main validation method - validates an entire document, single node from a HTML tree.

Note: This checks the validitity of the whole document and executes the validation loop.

By default, returns a dictionary with the number of successful checks, and a list of failures, warnings and skipped elements.

validate_element(*node*)

Validate a single node from a HTML element tree. Errors and warnings are attached to the instances `failures` and `warnings` properties.

By default, returns nothing.

validate_file(*filename*)
Validates a file given as a string filenames

By returns a dictionary of results from `validate_document`.

validate_files(**filenames*)
Validates the files given as a list of strings of filenames

By default, returns nothing.

validate_whole_document(*html*)
Validates an entire document from a HTML element tree. Errors and warnings are attached to the instances `failures` and `warnings` properties.

Note: This checks the validatity of the whole document, but does not execute the validation loop.

By default, returns nothing.

wcag_zoo.utils.**build_msg**(*node*, ***kwargs*)
Assistance method that builds a dictionary error message with appropriate references to the node

wcag_zoo.utils.**get_applicable_styles**(*node*)
Generates a list of dictionaries that contains all the styles that *could* influence the style of an element.

This is the collection of all styles from an element and all it parent elements.

Returns a list, with each list item being a dictionary with keys that correspond to CSS styles and the values are the corresponding values for each ancestor element.

CHAPTER 2

Command list

class wcag_zoo.validators.anteater.**Anteater**(**args*, ***kwargs*)
Anteater checks for alt and title attributes in image tags in HTML against the requirements of the WCAG2.0 standard

class wcag_zoo.validators.ayeaye.**Ayeaye**(**args*, ***kwargs*)
Checks for the existance of access key attributes within a HTML document and confirms their uniqueness. Fails if any duplicate access keys are found in the document Warns if no access keys are found in the document

class wcag_zoo.validators.glowworm.**Glowworm**(**args*, ***kwargs*)
Glowworm checks for supressed focus outlines.

class wcag_zoo.validators.molerat.**Molerat**(**args*, ***kwargs*)
Molerat checks color contrast in a HTML string against the WCAG2.0 standard

It checks foreground colors against background colors taking into account opacity values and font-size to conform to WCAG2.0 Guidelines 1.4.3 & 1.4.6.

However, it *doesn't* check contrast between foreground colors and background images.

Paradoxically:

a failed molerat check doesn't mean your page doesn't conform to WCAG2.0

but a successful molerat check doesn't mean your page will conform either...

Command line tools aren't a replacement for good user testing!

class wcag_zoo.validators.parade.**Parade**(**args*, ***kwargs*)
Run a number of validators together across a file or collection of files in a single command.

class wcag_zoo.validators.tarsier.**Tarsier**(**args*, ***kwargs*)
Tarsier reads heading levels in HTML documents (H1,H2,...) to verify the order and completion of headings against the requirements of the WCAG2.0 standard.

CHAPTER 3

Frequently Asked Questions

3.1 Can I use this to check my sites accessibility at different breakpoints?

Yes! Making sure your site is accessible at different screen sizes is important so this is vitally important. By default, WCAG-Zoo validators ignore `@media` rules, but if you are using CSS `@media` rules to provide different CSS rules to different users, you can declare which media rules to check against when running commands.

These can be added using the `--media_rules` command line flag (`-M`) or using the `media_rules` argument in Python. Any CSS `@media` rule that matches against *any* of the listed `media_rules` to check will be used, *even if they conflict*.

For example, below are some of the media rules used in the Twitter Bootstrap CSS framework

```
1. @media (max-device-width: 480px) and (orientation: landscape) {
2. @media (max-width: 767px) {
3. @media screen and (max-width: 767px) {
4. @media (min-width: 768px) {
5. @media (min-width: 768px) and (max-width: 991px) {
6. @media screen and (min-width: 768px) {
7. @media (min-width: 992px) {
8. @media (min-width: 992px) and (max-width: 1199px) {
9. @media (min-width: 1200px) {
```

The following command will check rules 4, 5 and 6 as all contain the string `(min-width: 768px)`:

```
zookeeper molerat --media_rules="(min-width: 768px)"
```

Note that this command will check media rules where the maximum width is 767px and the minimum width is 768px:

```
zookeeper molerat -M="(min-width: 768px)" -M="(max-width: 767px)"
```

In reality a browser would never render these as the rules conflict, but zookeeper isn't that smart yet.

3.2 Why is it important to check the accesibility of hidden elements?

Elements such as these often have their visibility toggled using Javascript in a browser, as such testing hidden elements ensures that if they become visible after rendering in the browser they conform to accessibility guidelines.

By default, all WCAG commands check that hidden elements are valid, however they also accept a `ignore_hidden` argument (or `-H` on the command line) that prevents validation of elements that are hidden in CSS, such as those contained in elements that have a `display:none` or `visibility:hidden` directive.

3.3 Why does my page fail a contrast check when the contrast between foreground text color and a background image is really high?

Molerat can't see images and determines text contrast by checking the contrast between the calculated CSS rules for the foreground color (`color`) and background color (`background-color`) of a HTML element. If the element hasn't got a

Consider white text in a div with a black background *image* but no background color, inside a div with a white back ground, like that demonstrated below

```
+-------------------------------------------------+
|  (1) Black text / White background              |
|                                                 |
|  +-----------------------------------------+    |
|  | <div class='inner' id='hero_text'>      |    |
|  | (2) White text / Transparent background |    |
|  |                  Black bckrgound image  |    |
|  |                                         |    |
|  +-----------------------------------------+    |
+-------------------------------------------------+
```

In the above example, until the image loads the text in div (2) is invisible. If the connection is interrupted or a user has images disabled, the text would be unreadable. **The ideal way to resolve this is to add a background color to the inner ''div'' to ensure all users can read it.** If this isn't possible, to resolve this error, add the class or id to the appropriate exclusion rule. For example, from the command line:

```
zookeeper molerat somefile.html --skip_these_classes=inner
zookeeper molerat somefile.html --skip_these_ids=hero_text
```

Or when calling as a module:

```
Molerat(..., skip_these_classes=['inner'])
Molerat(..., skip_these_ids=['hero_text'])
```

3.4 Why doesn't WCAG-Zoo support Python 2?

Python 2 is on a long deprecation cycle, and a number of big libraries (such as Django) are beginning the process to remove Python 2 support entirely. Making WCAG-Zoo Python 3 only made building it much easier and removed the need for Python2/3 hacks to support both properly.

If you are building a Python 2 tool and absolutely need support you have a number of options

- Download the code to a place your Python 2 code can import it
- Use the demonstration scripts as way to run the WCAG-Zoo command line tools from within Python 2 code using `subprocess` and parse the JSON
- Consider how import Python 2 is to you or your users and port your code to Python 3 (its not as painful as you think now and there are benefits)

CHAPTER 4

Guide to writing tests for commands

To make writing tests easier, tests can be declaratively written in HTML using `data-*` attributes to specify the command to check a HTML document against, and the expected errors.

The available attributes are:

- **`data-wcag-test-command` (only on the root `html` element) - the specific zookeeper** validator to use for the HTML file.
- **`data-wcag-arg-\*` (only on the root `html` element) - attributes starting with** `data-wcag-arg-` specify arguments to pass when running the given command. Each attribute constitutes a key/value pair, with the key corresponding to everything captured by the asterisk (`*`) above. All values are evaluated in Python and are expected to be valid Python literals - as such numbers are treated as numbers, booleans are booleans, and strings need to be double quoted.
- **`data-wcag-failure-code` and `data-wcag-warning-code` - can be located on any element in the body.** These specify both that a given node should produce a failure or warning when using the above command and arguments, as well as the expected error code.

Below is an example that tests the `molerat` command, for WCAG 2.0-AA compliance and checks using a specific media rule to check a page for use on small screens.

```
<html
    data-wcag-test-command="molerat"
    data-wcag-arg-level="'AA'"
    data-wcag-arg-skip_these_classes="'sneaky'"
    data-wcag-arg-media_rules="['max-width: 600px']">
    <head>
        <style>
            .snarky {
                color: white;
                background-color: black;
            }
            @media (max-width: 600px) {
              .snarky {
                color: black;
              }
```

```
            }
        </style>
    </head>
    <body>
        <div class="snarky" data-wcag-failure-code="molerat-1">
            I hate mobile phones, but if you are
            reading this page on a mobile you'll never know!!
        </div>
        <div class="snarky sneaky">
            And I'll make sure you never see this either!
            And the zookeeper will never tell!
        </div>
        <p>
            Welcome to my page!
        </p>
    </body>
</html>
```

This is equivilent to running the following at the command line:

```
zookeeper molerat the_file_above.html \
  --level=AA \
  --media_rules='max-width: 600px' \
  --skip_these_classes=sneaky
```

or the following python command:

```
from wcag_zoo.validators.molerat import Molerat

validator = Molerat(
    level="AA",
    media_rules=['max-width: 600px'],
    skip_these_classes=["sneaky"]
    ).validate_document(the_text_above)
```

CHAPTER 5

Using WCAG-Zoo in other languages

Below are a small number of example scripts that show how to call the WCAG-Zoo scripts from a number of target languages to provide runtime support for accessibility checking.

All of the following snippets will either:

- Store a specified string `my_html` as the temporary file accessed by the variable `tmp_file` or
- Pass a specified string `my_html` into the command via stdin

Then:

1. Execute the WCAG command `wcag_zoo.validators.tarsier` using Python and store the result as `results`
2. Capture the `results` string and parse it from JSON into the variable `json_results`
3. Prints the number of failures for the file

All of the following scripts are public domain samples and not guaranteed to work in production in any way. All scripts should output something similar to `/tmp/wcag117015-32930-onps7o 1 failures`

5.1 Node.JS

File based:

Assuming you have temp installed using `npm install temp`:

```
#!/usr/bin/env node
var temp = require('temp'),
    fs   = require('fs'),
    exec = require('child_process').exec;

temp.open('wcag', function(err, info) {
  if (!err) {
    fs.write(
      info.fd,
```

```
      "<html><head><body><h1>Heading 1</h1><h3>This is wrong, it should be h2",
      function(err){
        /* Ignore, we don't care */
      }
    );
    fs.close(info.fd, function(err) {
      exec("zookeeper tarsier '" + info.path + "' -F",
        function(err, stdout) {
          results = stdout;
          json_results = JSON.parse(results);
          console.log(
            json_results[0][0],
            json_results[0][1].failures.length,
            "failures"
          );
        }
      );
    });
  }
});
```

5.2 Perl

File based:

```
#!/usr/bin/env perl
require File::Temp;
use File::Temp ();
use File::Temp qw/ :seekable /;
use JSON;

$my_html = "<html><head><body><h2>This is wrong, it should be h1";

$tmp = File::Temp->new();
print $tmp $my_html;
$tmp->seek( 0, SEEK_END );

$fn = $tmp->filename;

$results = `zookeeper tarsier $fn -J`;
@json_results = decode_json($results);
$filename = @{@{@json_results[0]}[0]}[0];
$len = scalar @{@{@{@{@json_results[0]}[0]}[1]}{'failures'}};
print "$filename $len failures\n";
```

5.3 Python

Included for reference, but WCAG-Zoo can be used in Python by importing validators directly.

File based:

```
#!/usr/bin/env python2
from __future__ import print_function
```

```
import json
import tempfile
import subprocess

my_html = "<html><head><body><h1>1</h1><h3>This is wrong, it should be h2"

tmp_file = tempfile.NamedTemporaryFile()
tmp_file.write(my_html)
tmp_file.seek(0)

process = subprocess.Popen(
    ["zookeeper", "tarsier", tmp_file.name, "-F"],
    stdout=subprocess.PIPE
)

results = process.communicate()[0]
json_results = json.loads(results)

print(json_results[0][0],
    len(json_results[0][1]['failures']),
    "failures"
)
```

5.4 Ruby

Assuming you have installed `json` like so: `gem install json`

File based:

```
#!/usr/bin/env ruby
require 'json'
require 'tempfile'

my_html = "<html><head><body><h1>1</h1><h3>This is wrong, it should be h2"

tmp_file = Tempfile.new('foo')
tmp_file.write(my_html)
tmp_file.close

results = `zookeeper tarsier #{tmp_file.path} -F`
json_results = JSON.parse(results)
print json_results[0][0], " ", json_results[0][1]['failures'].size, " failures\n"
```

CHAPTER 6

Using WCAG-Zoo validators in python

As WCAG-Zoo is a native Python 3 library, it can be imported and used in Python 3 scripts easily.

Once installed from `pip` (`pip install wcag_zoo`), load any of the validators and call `validate_document` on an instance as shown in the example below.

```
#!/usr/bin/env python3
from wcag_zoo.validators.tarsier import Tarsier

my_html = b"<html><head><body><h1>1</h1><h3>This is wrong, it should be h2"
instance = Tarsier()
results = instance.validate_document(my_html)

print("/no/tmp/dir", len(results['failures']), "failures")
```

CHAPTER 7

WCAG guideline index and validator reference

This is an index of Web Accessibility Content Guidelines that can be verified using WCAG-Zoo as well as the techniques used for verification and the validators which perform the validation.

- 1.1.1 - Nontext Content
  - H37: Using alt attributes on img elements - `anteater`
- 1.3.1 - Info and Relationships
  - H42: Using h1-h6 to identify headings - `tarsier`
- 1.4.3 - Contrast (Minimum)
  - G18: Ensuring that a contrast ratio of at least 4.5:1 exists between text (and images of text) and background behind the text - `molerat`
  - G145: Ensuring that a contrast ratio of at least 3:1 exists between text (and images of text) and background behind the text - `molerat` (large text only)
- 1.4.6 - Contrast (Enhanced)
  - G17: Ensuring that a contrast ratio of at least 7:1 exists between text (and images of text) and background behind the text - `molerat` (using AAA compliance)
  - G18: Ensuring that a contrast ratio of at least 4.5:1 exists between text (and images of text) and background behind the text - `molerat` (large text only, using AAA compliance)
- 2.1.1 - Keyboard
  - G90: Providing keyboard-triggered event handlers - `ayeaye` (check for clashes with accesskeys).

7.1 Future additions

The following guidelines and techniques have been identified as potential additions to the WCAG-Zoo.

- 1.1.1 - Nontext Content

  - H53, H44
  - H36 - Make sure input tags with src have alt or text
  - H30 - Make sure links have text
- 1.2.3 - Audio Description or Media Alternative (Prerecorded) - H96
- 1.3.1 - Info and Relationships
- 2.4.1 - Bypass blocks - H69
- 2.4.2 - Page titled - H25 + G88 (very basic)
- 2.4.7 - Focus Visible - (for :focus)
- 3.1.1 - Language of Page - H57
- 4.1.1 - Parsing H74, H75

CHAPTER 8

Disclaimer

WCAG-Zoo does not *gaurantee* complete compliance with the Web Content Accessibility Guidelines. There are many aspects of the WCAG recommendations that are subjective and are difficult or even impossible to verify automatically.

Where WCAG compliance is a contractual or legislative requirement, always speak to an accesibility expert.

CHAPTER 9

What is it?

WCAG-Zoo is a set of command line tools that help provide basic validation of HTML against the accessibility guidelines laid out by the W3C Web Content Accessibility Guidelines 2.0.

Each tool checks against a limited set of these and is designed to return simple text output and returns an error (or success) code so it can be integrated into continuous build tools like Travis-CI or Jenkins. It can even be imported into your Python code for additional functionality.

CHAPTER 10

Why should I care about accessibility guidelines?

Accessibility means that everyone can use your site. We often forget that not everyone has perfect vision - or even has vision at all! Complete or partial blindess, color-blindness or just old-age can all impact how readily accessible your website can be.

By building accessibility checking into your build scripts you can be relatively certain that all people can readily use your website. And if you come across an issue, you identify it early - before you hit production and they start causing problems for people.

Plus, integrating accessibility into your build scripts shows that you really care about the usability of your site. These tools won't pick up every issue around accessibility, but they'll pick up enough (and do so automatically) and helps demonstrate a commitment to accessibility where possible.

CHAPTER 11

That sounds like a lot of work, is it really that useful?

Granted, accessibility is tough - and you might question how useful it is. If you have an app targeted to a very niche demographic and are working on tight timeframes, maybe accessibility isn't important right now.

But some industries, such as Government, Healthcare, Legal and Retail all care **a lot** about WCAG compliance. To the point that in some areas it is legislated or mandated. In some cases not complying with certain accessibility guidelines can even get sued can lead to large, expensive lawsuits!

If you care about working in any of the above sectors, being able to *prove* you are compliant can be a big plus, and having that proof built in to your testing suite means identiying issues earlier before they are a problem.

CHAPTER 12

But all my pages are dynamically created and I use a CSS pre-processor

Doesn't matter. If you can generate them, you can output your HTML and CSS in a build script and feed them into the WCAG-Zoo via the command line.

CHAPTER 13

But I have lots of user-generated content! How can I possibly test that?

It doesn't matter if your site is mostly user-generated pages. Testing what you can sets a good example to your users. Plus many front-end WYSIWYG editors have their own compliance checkers too. This also sets a good example to your end-users as they know that the rest of the site is WCAG-Compliant so they should probably endevour to make sure their own content is too.

Since this is a Python library if you are building a dynamic site where end users can edit HTML that uses Python on the server side you can import any of the validators directly into your code so you can confirm that the user created markup is valid as well.

Lastly, if you are building a dynamic site in a language other than Python you can run any of the command line scripts with the `--json` or `-J` flag and this will produce a JSON output that can be parsed and used in your preferred target language.

For details on this see the section in the documentation titled "Using WCAG-Zoo in languages other than Python".

CHAPTER 14

Do I have to check *every* page?

The good news is probably not. If your CSS is reused across across lots of your site then checking a handful of generate pages is probably good enough.

CHAPTER 15

You convinced me, how do I use it?

Two ways:

1. In your build and tests scripts, generate some HTML files and use the command line tools so that you can verify your that the CSS and HTML you output can be read.
2. If you are using Python, once installed from pip, you can import any or all of the tools and inspect the messages and errors directly using:

```
from wcag_zoo.molerat import molerat
messages = molerat(html=some_text, ... )
assert len(messages['failed']) == 0
```

CHAPTER 16

I've done all that can I have a badge?

Of course! You are on the honour system with these for now. So if you use WCAG-Zoo in your tests and like Github-like badges, pick one of these:

- `https://img.shields.io/badge/WCAG_Zoo-AA-green.svg`
- `https://img.shields.io/badge/WCAG_Zoo-AAA-green.svg`

ReSTructured Text:

```
.. image:: https://img.shields.io/badge/WCAG_Zoo-AA-green.svg
   :target: https://github.com/data61/wcag-zoo/wiki/Compliance-Statement
   :alt: This repository is WCAG-Zoo compliant
```

Markdown:

```
![This repository is WCAG-Zoo compliant][wcag-zoo-logo]

[wcag-zoo-logo]: https://img.shields.io/badge/WCAG_Zoo-AA-green.svg "WCAG-Zoo␣
↪Compliant"
```

CHAPTER 17

Installing

- Stable: `pip3 install wcag-zoo`
- Development: `pip3 install https://github.com/LegoStormtroopr/wcag-zoo`

CHAPTER 18

How to Use

All WCAG-Zoo commands are exposed through `zookeeper` from the command line.

Current critters include:

- Anteater - checks `img` tags for alt tags:

```
zookeeper anteater your_file.html --level=AA
```

- Ayeaye - checks for the presence and uniqueness of accesskeys:

```
zookeeper ayeaye your_file.html --level=AA
```

- Molerat - color contrast checking:

```
zookeeper molerat your_file.html --level=AA
```

- Parade - runs all validators against the given files with allowable exclusions:

```
zookeeper parade your_file.html --level=AA
```

- Tarsier - tree traveral to check headings are correct:

```
zookeeper tarsier your_file.html --level=AA
```

For more help on zookeeper from the command line run:

```
zookeeper --help
```

Or for help on a specific command:

```
zookeeper ayeaye --help
```

CHAPTER 19

Limitations

At this point, WCAG-Zoo commands **do not** handle nested media queries, but they do support single level media queries. So this will be interpreted:

```
@media (min-width: 600px) and (max-width: 800px) {
    .this_rule_works {color:red}
}
```

But this won't (plus this isn't supported across some browsers):

```
@media (min-width: 600px) {
    @media (max-width: 800px) {
        .this_rule_wont_work {color:red}
    }
}
```

CHAPTER 20

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

Index