settingscascade

Release 03

Paul Becotte

Jul 15, 2019

DOCUMENTATION

Settings cascade is designed for situations where you need to merge configuration settings from different hierarchical sources. The model is the way that CSS cascades onto elements. You can define config the same way that css rules get specified-

```
task.default:
    command: "echo hello"
    on_complete: "echo world"
project_name: "my project"
```

Then your app can use the config

```
class Task(SettingsSchema):
        _name_ = task
        command: str
        on_complete: str

config = SettingsManager(yaml.load("config.yml"), [Task])
task_config = config.task(class="default")
run_task(
        command=task_config.command,
        on_complete=task_config.on_complete,
        name=config.project_name,
)
```

Read the full documentation at https://settingscascade.readthedocs.io/en/latest/

CHAPTER

ONE

INSTALLATION

You can install settingscascade from pypi-

```
pip install settingscascade
```

1.1 Loading Data

1.1.1 Defining Your Schema

Considering the html/css model, a Schema lines up with an element (such as a <p> or). Once you define the Schemas that make up your config, elements in your settings will be mapped against the schema. An HTML document may look like

```
<body>
        <div class="outer">
                <p>Hello!</p>
        </div>
</body>
```

In that example the elements are *body*, *div*, and *p*. In SettingsCascade you define the elements that make sense for your app. Imagine a task runner app like Fabric. You may have the concept of Tasks and Environments.

```
from settingscascade import SettingsSchema

class Environment(SettingsSchema):
        _name_ = "env"
        python: str
        pythonpath: List[str]

class Task(SettingsSchema):
        _name_ = "task"
        command: List[str]
        wait: bool
```

A SettingsSchema class has one mandatory field- _name_. This is the name of the element as it will appear in setting files. It is the equivalent of *div* of *p*. Each schema defines annotations for the valid variables that make up a config for that element. When you load data, if the rule has an element defined, the system will do some validation of the type of the data you are loading. This is not a comprehensive check- for example, if you define List[str] this will only verify that the data is a list, it will not look at the values. typing.Any can be used as expected.

1.1.2 Specificity

When loading data, each section of rules will be associated with a selector, and then when your app tries to look up a rule, the most specific rule whose selector matches the current context will be returned. Consider

```
".env":
        val_a = "a"

"class.env":
        val_a = "b"
```

If you try to look up *val_a* from the context *module.env* it would return "a", while from context *class.env* it would return "b". A rule section must match ALL elements of the context to be used, but not all elements of the context need to be used in the selector. (The first example would work because there is no rule that matches *module.env*, but there IS a rule that matches **.env*) The specificity rules are the same as CSS- the score is a 3-tuple- - Count of #ID values - Count of .class values - count of element values.

So- - .myclass == 0, 1, 0 - el.myclass == 0, 1, 1 - parent child#thechild == 1, 0, 2

Specificity scores are compared pairwise, the first value, then the second, then the third. The three examples above are listed from least specific to most.

1.1.3 Data Loader

The core class of SettingsCascade it the SettingsManager class. You create a settings manager by passing it a list of data dictionaries and a list of ElementSchema classes that you have defined. It will then build its internal cascade of rule definitions (verified according to the schemas you passed). The dictionaries themselves can be created any way you want- load from TOML, JSON, Yaml, a Python dict, whatever.

```
from pathlib import Path
from toml import loads
from settingscascade import SettingsManager

els = {EnvironmentSchema, TaskSchema}
data =  loads(Path("pyproject.toml").read_text())
default_data = {"mydefault": 42}
config = SettingsManager([default_data, data], els)
```

The algorithm for loading the data for each rule section is 1. Determine the selector term for this section. 2. Check for any key named *_name_* - add it to the selector as a class. 3. check for any key named _id_ - add it to the selector as an id. 4. Append the selector to the context passed in from the parent to get the full selector for this section. 5. Iterate through the remaining key, value pairs. 6. If the key matches the *_name_* of one of the element schemas or contains a . or #, load that value as a new section, passing the selector as the current context and the key as that sections selector term. 7. For each remaining key, if this section matches an element schema, verify that the key is in the annotations for the schema and the value has the correct type. 8. Load the key, value pairs into the config manager as a Rules section.

There are two ways to add classes or ids to a rule section selector. first, you can just add them directly as though it were css. The toml file below has four sections. The specifiers are read as *environment*, *.prod*, *environment.prod*, *environment.prod task*. This winds up working exactly like CSS, and is the most obvious way to use this library.

```
[environment]
setting_a = "outer"

[".prod"]
some_setting = "production"
```

(continues on next page)

(continued from previous page)

```
["environment.prod"]
        name = "default_task"
        task_setting = "less"
["environment.prod".task]
        setting_a = "inner"
```

You'll notice that in toml, to put a . or a # in the key of a section, you'll have to use quotes. Because of that, the loader will also look for magic names *_name_* and *_id_* to pull them from the object. Below, the selector for section 2 would be *tasks.default_task*

```
[environment]
setting_a = "outer"

[[task]]
        _name_ = "default_task"
        task_setting = "less"
        [task.environment]
                setting_a = "inner"

[[task]]
task_setting = "more"
```

Note there are two special cases to consider from the previous example. The first is a list of dictionaries (like *task*). In this case the library will use the key of the list to build the selector for each element of the list. In this case it would be *task.default_task* and *task* respectively. The other is that the second list there has no _name_ variable, so will just get the selector from the list- if there were more then one item in that list with the same situation, they would override each other. In the event that two rule section have the SAME specificity, they get priority in reverse order of how they were loaded- the last section beats the first.

1.1.4 Detailed config to selector rules

Parse map into selector/ruleset ! Any key that is not an element is considered to be a rule ! There are two more special keynames - _name_ and _id_. If these are contained in a map, they update the selector of the parent key

keyname "keyname": {...

keyname.typename "keyname.typename": {...

keyname.typename "keyname": {"_name_": "typename, ...

keyname#id "keyname": {"_id_": "id", ...

keyname.typename "keyname": [{"_name_": "typename", ...

keyname otherkeyname.nestedname "keyname": {"otherkeyname": {"_name_": "nestedname", ...

1.2 Accessing Config values

1.2.1 The Data Context

Once the data is loaded, it can be accessed anywhere in your application by just accessing the attribute on your config object.

```
var = config.my_var
```

Whenever you access a value on the config object, it searches through all of the rulesets that is has for the specified key. It then uses its *current_context* to pick the one that is a match with the highest specificity. In the above example, there is no context, so it searches with "" as the selector string. You can use *config.context(selector)* as a context manager to get deeper values-

```
with config.context("task.default_task environment"):
        assert config.setting_a == "inner"
```

The context works like the nested tree of an html structure. the model

```
with config.context("el.myclass"):
        config.the_value
        with config.context("child"):
                config.the_value
                with config.context("par#inner"):
                        config.the_value
```

would have the same effect as an html structure of

```
<el class="myclass">
        the_value
        <child>
                the_value
                <par id=inner>
                        the_value
                </par>
        </child>
</el>
```

if you had a ruleset like

```
the_value: 0
el:
        the_value: 1
child:
        the_value: 2
par:
        the_value: 3
.myclass:
        the_value: 4
#inner:
        the_value: 5
.myclass #inner:
        the_value: 6
el child:
        the_value: 7
```

The output would be - 4 - 7 - 6

1.2.2 Using Schemas

Schemas let you enforce structure on rules- what attributes are actually valid for any particular element type. We have seen how to define them, lets see how you can use them.

```
class Task(ElementSchema):
        taskval: str
```

(continues on next page)

(continued from previous page)

```
config = SettingsManager([data], [Task])
print(config.task().someval)
```

Each ElementSchema has a _name_ associated with it. When you access that name on the SettingsManager object, it will create an instance of the Schema for you. Basically, it will push the element onto the search context so that any values you lookup will come from that element. You can pass extra context as well using the *name* and *identifier* arguments-

```
with config.context("parent"):
        config.task(identifier="mytask").someval
```

In this case, someval would be looked up with the context "parent task#mytask". Schema objects aren't context managers, they keep a closure of the context when they were created, so this

```
with config.context("parent"):
        task = config.task(identifier="mytask")
val  = task.someval
```

would work the same way as the previous example. Schema objects also provides a convenience method *load()* which will return a dictionary of all of the resolved properties that are defined on the schema. Its the equivalent of

```
data = {key: getattr(task, key) for key in Task.__props__}
```

1.3 Jinja templating

Any string value returned from your config will be run through a Jinja2 template resolver before being returned. Any missing variables in the templates will be looked up in the config using the current context.

```
config = SettingsManager({
        "basic": "Var {{someval}}",
        "someval": "default",
        "task": {"someval": "override"}
}, {"task"})

config.basic == "Var default"
with config.context("task"):
        config.basic == "Var override"
```

This could allow you to be more flexible when merging data from multiple sources such as default, org, and user level config files. You can even add custom filters to the environment such as

```
config = SettingsManager({
        "myval": "{{ (1, 3) | add_two_numbers }}"
})
config.add_filter("add_two_numbers", lambda tup: tup[0] + tup[1])
config.myval == "4"
```

1.4 API

class settingscascade.**SettingsManager**(*data: List[dict], els: Optional[List[Type[settingscascade.schema.ElementSchema]]] = None*)

A Settingsmanager object.

Parameters

- **data** – a list of settings dictionaries
- **els** – a List of ElemeentSchema objects, If not specified, no elements will be created

context(*new_context: str = ''*)

Add context onto the current context. This takes a string and appends it to the existing context. For example (using html elements-)

```
with config.context("body h1.intro"):
    with config.context("div.myel"):
        config.current_context == "body h1.intro div.myel"
```

property current_context

Gets a string that represents the current context used for settings lookups :return: str

load_data(*data: dict, next_item: str = '', selector: str = ''*)

Loads a settings dictionary into the rule stack.

Parameters

- **data** – A settings dictionary. Keys should either be selectors or value names.
- **next_item** – The key for this rule-set. Pulled from the parent dict when loading recursively.
- **selector** – The full context selector for any parent rule-sets that should be added to the selector for this one

class settingscascade.**ElementSchema**(*configManager*)

Class that defines the schema for a particular element in your settings heirarchy. Subclass this and add annotations to define the allowed values for this element type-

```
class Element(ElementSchema):
    color: str
    height: int
```

property context

The context stack that will be used to look up settings for this object

load()

Loads the settings for this schema into a python dictionary. Looks up the value for each property using the current context stack for this object.

Note: This will throw an error if there are settings defined on the schema that can't be found in any level!

PYTHON MODULE INDEX

S

INDEX