Confetti Documentation

Release 1.1.0

Rotem Yaari

October 07, 2013

CONTENTS

Confetti deals mostly with `Config` objects. These objects represent nodes and leaves in a configuration structure, and provide most of the functionality for querying and manipulating the configuration.

By default, parts of the configuration, even scalar values, are wrapped in Config objects when possible. However, Confetti provides ways to access values as simple Python values.

CHAPTER

ONE

BASIC OPERATIONS

1.1 Initializing Configurations

The most convenient way to initialize a configuration structure is by simply passing a nested dictionary into the `Config` constructor:

```
from confetti import Config
CONFIG = Config({
    "a" : {
        "b" : 2,
    }
})
```

Confetti also has convenience helpers to load from files that contain the above structure (NB the capital `CONFIG`), via the `Config.from_filename()`, `Config.from_file()` and `Config.from_string()` methods.

1.2 Querying the Configuration Tree

1.2.1 Getting Direct Values

The simplest and most memorizable way to access values in the configuration structure is through the `root` member of the Config object. This member is a proxy to the Config object and allows accessing values through attributes:

```
>>> from confetti import Config
>>> c = Config({
...   "a" : {"b" : {"c" : 12}},
... })
>>> c.root.a.b.c
12
```

You can also use `__getitem__` syntax (as in Python dicts) to access nodes and values:

```
>>> c["a"]["b"]["c"]
12
```

For getting a nested value you can also use **configuration path**, which are dotted notations of the above:

```
>>> c.get_path("a.b.c")
12
```

1.2.2 Getting Config Objects

For most manipulation and advanced querying purposes, a user would want to work with *config objects*, rather than direct values. Those provide more reflection capabilities and smarter traversal (e.g. finding the parent of a node). This is possible via the `Config.get_config()` function:

```
>>> c.get_config("a")
<Config {'b': {'c': 12}}>
```

You can also use dotted notation:

```
>>> c.get_config("a.b.c")
<Config 12>
```

1.3 Modifying Configurations

1.3.1 Existing Values

Existing values can be changed pretty easily, both by using the `root` proxy, and by using `__setitem__`:

```
>>> c["a"]["b"]["c"] = 100
>>> c.root.a.b.c = 100
```

1.3.2 New Values and Nodes

To avoid mistakes when using or updating configurations, Confetti does not allow setting nonexistent values:

```
>>> c["new_value"] = 1
Traceback (most recent call last):
   ...
CannotSetValue: ...
```

Configuration objects have the `Config.extend()` method to assign new values or nested structures to an existing configuration, that does the trick:

```
>>> c.extend({"new_value" : 1})
>>> c.root.new_value
1
```

CHAPTER

TWO

ADVANCED USES

2.1 Path Assignments

Config objects can assign to paths using the `Config.assign_path()` method:

```
>>> c.assign_path("a.b.c", 2)
>>> c.root.a.b.c
2
```

Which is a synonym for:

```
>>> c.get_config("a.b.c").set_value(2)
```

In some cases you want to process config overrides from various sources that are not completely type safe, e.g. command-line or environment variables. Such variables would look like `'some.value=2'`. Confetti provides a utility for easily assigning such expressions, optionally deducing the leaf type:

```
>>> c.assign_path_expression("a.b.c=234", deduce_type=True)
>>> c.root.a.b.c
234
```

The default is no type deduction, which results in string values always:

```
>>> c.assign_path_expression("a.b.c=230")
>>> c.root.a.b.c
'230'
```

2.2 Cross References

In many cases you want to set a single value in your configuration, and have other leaves take it by default. Instead of repeating yourself like so:

```
>>> cfg = Config(dict(
...     my_value = 1337,
...     value_1 = 1337,
...     x = dict(
...         y = dict(
...             z = 1337,
...         )
...     )
... ))
```

You can do this:

```
>>> from confetti import Ref
>>> cfg = Config(dict(
...     my_value = 1337,
...     value_1 = Ref(".my_value"),
...     x = dict(
...         y = dict(
...             z = Ref("...my_value"),
...         )
...     )
... ))
>>> cfg.root.x.y.z
1337
```

Or you can apply a custom filter to the reference, to create derived values:

```
>>> cfg = Config(dict(
...     my_value = 1337,
...     value_1 = Ref(".my_value", filter="I am {0}".format),
... ))
>>> cfg.root.value_1
'I am 1337'
```

2.3 Metadata

Confetti supports attaching metadata to configuration values. This is can be done directly with manipulating the `metadata` attribute of the `Config` class, but also has a handy syntax making use of the `//` operator:

```
>>> from confetti import Config, Metadata
>>> cfg = Config({
...     "name" : "value" // Metadata(metadata_key="metadata_value"),
... })
>>> cfg.get_config("name").metadata
{'metadata_key': 'metadata_value'}
```

CHAPTER

THREE

THE CONFETTI.CONFIG.CONFIG CLASS

class `confetti.config.`**`Config`**(*value=<NOTHING>*, *parent=None*, *metadata=None*)

`__contains__`(*child_name*)
Checks if this config object has a child under the given child_name

`__getitem__`(*item*)
Retrieves a direct child of this config object assuming it exists. The child is returned as a value, not as a config object. If you wish to get the child as a config object, use `Config.get_config()`.

Raises KeyError if no such child exists

`__setitem__`(*item*, *value*)
Sets a value to a value (leaf) child. If the child does not currently exist, this will succeed only if the value assigned is a config object.

`__weakref__`
list of weak references to the object (if defined)

`assign_path`(*path*, *value*)
Assigns `value` to the dotted path `path`.

```
>>> config = Config({"a" : {"b" : 2}})
>>> config.assign_path("a.b", 3)
>>> config.root.a.b
3
```

`backup`()
Saves a copy of the current state in the backup stack, possibly to be restored later

`extend`(*conf*)
Extends a configuration files by adding values from a specified config or dict. This permits adding new (previously nonexisting) structures or nodes to the configuration.

classmethod `from_file`(*f*, *filename='?'*, *namespace=None*)
Initializes the config from a file object `f`. The file is expected to contain a variable named `CONFIG`.

classmethod `from_filename`(*filename*, *namespace=None*)
Initializes the config from a file named `filename`. The file is expected to contain a variable named `CONFIG`.

classmethod `from_string`(*s*, *namespace=None*)
Initializes the config from a string. The string is expected to contain the config as a variable named

`CONFIG`.

get (*child_name*, *default=None*)
: Similar to `dict.get()`, tries to get a child by its name, defaulting to None or a specific default value

get_config (*path*)
: Returns the child under the name `path` (dotted notation) as a config object.

get_parent ()
: Returns the parent config object

get_path (*path*)
: Gets a value by its dotted path

```
>>> config = Config({"a" : {"b" : 2}})
>>> config.get_path("a.b")
2
```

get_value ()
: Gets the value of the config object, assuming it represents a leaf

See Also:

`is_leaf`

is_leaf ()
: Returns whether this config object is a leaf, i.e. represents a value rather than a tree node.

keys ()
: Similar to `dict.keys()` - returns iterable of all keys in the config object

pop (*child_name*)
: Removes a child by its name

restore ()
: Restores the most recent backup of the configuration under this child

serialize_to_dict ()
: Returns a recursive dict equivalent of this config object

set_value (*value*)
: Sets the value for the config object assuming it is a leaf

traverse_leaves ()
: A generator, yielding tuples of the form (subpath, config_object) for each leaf config under the given config object

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*