candv Documentation

Release 1.3.0

Alexander Oblovatniy

Oct 30, 2020

Contents

candv stands for *Constants & Values*. It is a little Python library which provides an easy way for creating complex constants.

CHAPTER 1

Contents

1.1 Dive in

How often do you need to define names which can be treated as constants? How about grouping them into something integral? What about giving names and descriptions for your constants? Attaching values to them? Do you need to find constants by their names or values? What about combining groups of constants into an hierarchy? And finally, how do you imagine documenting process of this all?

1.1.1 Simple example

Well, if you have ever asked yourself one of these questions, this library may answer you. Just look:

```
>>> from candv import Constants, SimpleConstant
>>> class BAR(Constants):
...     """
...     This is an example container of named constants.
...     """
...     ONE = SimpleConstant()
...     TWO = SimpleConstant()
...     NINE = SimpleConstant()
...
```

Let's see some stuff. What's `BAR`?

```
>>> BAR
<constants container 'BAR'>
>>> BAR.name
'BAR'
>>> BAR.full_name
'BAR'
```

What constants does it have? What's their order?

```
>>> BAR.names()
['ONE', 'TWO', 'NINE']
```

What are those constants?

```
>>> BAR.constants()
[<constant 'BAR.ONE'>, <constant 'BAR.TWO'>, <constant 'BAR.NINE'>]
>>> BAR.items()
[('ONE', <constant 'BAR.ONE'>), ('TWO', <constant 'BAR.TWO'>), ('NINE', <constant
↪'BAR.NINE'>)]
```

How much constants are there in the container?

```
>>> len(BAR)
3
```

Does `BAR` have a constant named `ONE`?

```
>>> BAR.has_name('ONE')
True
>>> 'ONE' in BAR
True
```

How to get a constant by name?

```
>>> BAR['TWO']
<constant 'BAR.TWO'>
```

How to get a constant by name with fallback to default value?

```
>>> BAR.get('XXX', default=123)
123
```

How to access a single constant?

```
>>> BAR.ONE
<constant 'BAR.ONE'>
```

What attributes does it have?

```
>>> BAR.ONE.name
'ONE'
>>> BAR.ONE.full_name
'BAR.ONE'
>>> BAR.ONE.container
<constants container 'BAR'>
```

1.1.2 Complex example

Was it too simple for you? Watchout:

```
>>> from candv import (
...     Constants, Values, SimpleConstant, VerboseConstant, ValueConstant,
...     VerboseValueConstant,
... )
```

(continues on next page)

(continued from previous page)

```
>>> class FOO(Constants):
...     """
...     Example container of constants which shows the diversity of the library.
...     """
...     #: just a named constant
...     ONE = SimpleConstant()
...     #: named constant with verbose name
...     BAR = VerboseConstant("bar constant")
...     #: named constant with verbose name and description
...     BAZ = VerboseConstant(verbose_name="baz constant",
...                           help_text="description of baz constant")
...     #: named constant with value
...     QUX = ValueConstant(4)
...     #: another named constant with another value (list)
...     SOME = ValueConstant(['1', 4, True])
...     #: yet another named constant with another value, verbose name and description
...     SOME_VERBOSE = VerboseValueConstant("some value",
...                                         "some string",
...                                         "this is just some string")
...     #: named group of constants with values
...     GROUP = SimpleConstant().to_group(Values,
...         SIX=ValueConstant(6),
...         SEVEN=ValueConstant("S373N"),
...     )
...     #: subgroup with name, value and verbose name
...     MEGAGROUP = VerboseValueConstant(
...         value=100500,
...         verbose_name="megagroup"
...     ).to_group(Values,
...         HEY=ValueConstant(1),
...         #: group inside another group. How deep can you go?
...         YAY=ValueConstant(2).to_group(Constants,
...             OK=SimpleConstant(),
...             ERROR=SimpleConstant(),
...         ),
...     )
```

Whew! This looks like a big mess, but it shows all tasty things in one place. If you need something simple, you can have it.

Let's try to investigate this example.

At first, what do we have?

```
>>> FOO
<constants container 'FOO'>
>>> FOO.name
'FOO'
>>> FOO.full_name
'FOO'
```

What's inside?

```
>>> FOO.names()
['ONE', 'BAR', 'BAZ', 'QUX', 'SOME', 'SOME_VERBOSE', 'GROUP', 'MEGAGROUP']
```

What are all these things?

```
>>> FOO.constants()
[<constant 'FOO.ONE'>, <constant 'FOO.BAR'>, <constant 'FOO.BAZ'>, <constant 'FOO.QUX
↪'>, <constant 'FOO.SOME'>, <constant 'FOO.SOME_VERBOSE'>, <constants group 'FOO.
↪GROUP'>, <constants group 'FOO.MEGAGROUP'>]
```

Okay, we've seen *SimpleConstant* in action. What is *VerboseConstant*?

```
>>> FOO.BAZ
<constant 'FOO.BAZ'>
>>> FOO.BAZ.name
'BAZ'
>>> FOO.BAZ.full_name
'FOO.BAZ'
>>> FOO.BAZ.verbose_name
'baz constant'
>>> FOO.BAZ.help_text
'description of baz constant'
```

Yes, verbose constants can carry name and description for humans.

What about *ValueConstant*?

```
>>> FOO.QUX
<constant 'FOO.QUX'>
>>> FOO.QUX.name
'QUX'
>>> FOO.QUX.full_name
'FOO.QUX'
>>> FOO.QUX.value
4
```

How about adding verbosity to values?

```
>>> FOO.SOME_VERBOSE
<constant 'FOO.SOME_VERBOSE'>
>>> FOO.SOME_VERBOSE.value
'some value'
>>> FOO.SOME_VERBOSE.verbose_name
'some string'
>>> FOO.SOME_VERBOSE.help_text
'this is just some string'
```

What is a group?

```
>>> FOO.GROUP
<constants group 'FOO.GROUP'>
>>> FOO.GROUP.name
'GROUP'
>>> FOO.GROUP.full_name
'FOO.GROUP'
```

It's a constant!

```
>>> FOO.GROUP.constant_class
<class 'candv.ValueConstant'>
>>> FOO.GROUP.names()
['SIX', 'SEVEN']
```

(continues on next page)

(continued from previous page)

```
>>> FOO.GROUP.constants()
[<constant 'FOO.GROUP.SIX'>, <constant 'FOO.GROUP.SEVEN'>]
>>> FOO.GROUP.values()
[6, 'S373N']
>>> FOO.GROUP.get_by_value(6)
<constant 'FOO.GROUP.SIX'>
```

And it's a container! Groups, like photons, have dual nature: they are both constants and containers according to your needs.

Can we attach values and other stuff to groups? Surely!

```
>>> FOO.MEGAGROUP.value
100500
>>> FOO.MEGAGROUP.verbose_name
'megagroup'
>>> FOO.MEGAGROUP.names()
['HEY', 'YAY']
```

Can groups contain nested groups? Yes, they can:

```
>>> FOO.MEGAGROUP.YAY
<constants group 'FOO.MEGAGROUP.YAY'>
>>> FOO.MEGAGROUP.YAY.full_name
'FOO.MEGAGROUP.YAY'
>>> FOO.MEGAGROUP.YAY.names()
['OK', 'ERROR']
```

Visit hierarchies section for more info about groups.

1.1.3 Any real examples?

Yeah. There are some real public examples. See some examples.

In most cases you will be satisfied with standard facilities of the libraries. But you are not limited. You can *create your own* containers and constants. Examples mentioned above also may help you with this.

And of course, instead of a thousand words you can dig around tests.

Note: By the way, verbose names taste more sweet if you use verboselib for I18N (or any other suitable for you mechanism).

1.2 Installation

Just as easy as:

```
pip install candv
```

1.3 Usage

The main idea is that `constants` are *instances* of *`Constant`* class (or its subclasses) and they are stored inside *subclasses* of *`ConstantsContainer`* class which are called `containers`.

Every constant has its own name which is equal to the name of container's attribure they are assigned to. Every container is a singleton, i.e. you just need to define container's class and use it. You are not permitted to create instances of containers. This is unnecessary. Containers have class methods for accessing constants in different ways.

Constants remember the order they were defined inside container.

Constants may have custom attributes and methods. Containers may have custom class methods. *See customization docs*.

Constants may be converted into groups of constants providing ability to create different constant hierarchies (*see Hierarchies*).

1.3.1 Simple constants

Simple constants are really simple. They look like enumerations in Python 3.4:

```
>>> from candv import SimpleConstant, Constants
>>> class STATUS(Constants):
...     SUCCESS = SimpleConstant()
...     FAILURE = SimpleConstant()
...
```

And they can be used just like enumerations.

Here `STATUS` is a subclass of `candv.Constants`. The latter can contain any instances of `candv.SimpleConstant` class or its subclasses.

Note: `candv.SimpleConstant` and `candv.Constants` are aliases for *`candv.base.Constant`* and *`candv.base.ConstantsContainer`* respectively.

`STATUS` is a container:

```
>>> STATUS
<constants container 'STATUS'>
```

All containers have the following attributes:

```
>>> STATUS.name
'STATUS'
>>> STATUS.full_name
'STATUS'
```

They have an API which is similar to the API of Python's `dict` (in the mater of accessing its members):

```
>>> len(STATUS)
2
>>> 'SUCCESS' in STATUS
True
>>> STATUS.has_name('PENDING')
False
```

(continues on next page)

(continued from previous page)

```
>>> STATUS.names()
['SUCCESS', 'FAILURE']
>>> STATUS.constants()
[<constant 'STATUS.SUCCESS'>, <constant 'STATUS.FAILURE'>]
>>> STATUS.items()
[('SUCCESS', <constant 'STATUS.SUCCESS'>), ('FAILURE', <constant 'STATUS.FAILURE'>)]
>>> STATUS['FAILURE']
<constant 'STATUS.FAILURE'>
>>> STATUS.get('XXX', 999)
999
```

Note: Since 1.1.2 you can list constants and get the same result by calling `values()` and `itervalues()` also. Take into account, those methods are overridden in *Values* (see section below).

Also, you can access constants directly:

```
>>> STATUS.SUCCESS
<constant 'STATUS.SUCCESS'>
```

And access its attributes:

```
>>> STATUS.SUCCESS.name
'SUCCESS'
>>> STATUS.SUCCESS.full_name
'STATUS.SUCCESS'
>>> STATUS.SUCCESS.container
<constants container 'STATUS'>
```

1.3.2 Constants with values

Constants with values behave like simple constants, except they can have any object attached to them as a value. It's something like an ordered dictionary:

```
>>> from candv import ValueConstant, Values
>>> class TEAMS(Values):
...     NONE = ValueConstant('#EEE')
...     RED = ValueConstant('#F00')
...     BLUE = ValueConstant('#00F')
...
```

Here `TEAMS` is a subclass of *Values*, which is a more specialized container than `Constants`. As you may guessed, *ValueConstant* is a more specialized constant class than `SimpleConstant` and its instances have own values.

Note: `Values` and its subclasses treat as constants only instances of `ValueConstant` or its sublasses:

```
>>> class UNBOUND_CONSTANTS(Values):
...     FOO = SimpleConstant()
...     BAR = SimpleConstant()
...
```

Here `UNBOUND_CONSTANTS` container contains 2 instances of `SimpleConstant`, which is more gerenal then `ValueConstant`. It's not an error, but those 2 constants will be invisible for the container:

```
>>> UNBOUND_CONSTANTS.constants()
[]
>>> UNBOUND_CONSTANTS.FOO
<constant '__UNBOUND__.FOO'>
```

So, `TEAMS` is just another container:

```
>>> TEAMS
<constants container 'TEAMS'>
```

It has extra methods for working with valued constants. For example, you can list all values:

```
>>> TEAMS.values()
['#EEE', '#F00', '#00F']
```

Note: Since 1.1.2 methods *`values()`* and *`itervalues()`* from *`Values`* override methods `values()` and `itervalues()` from *`ConstantsContainer`* accordingly.

And you can get a constant by its value:

```
>>> TEAMS.get_by_value('#F00')
<constant 'TEAMS.RED'>
```

If you have different constants with equal values, it's OK anyway:

```
>>> class FOO(Values):
...     ATTR1 = ValueConstant('one')
...     ATTRX = ValueConstant('x')
...     ATTR2 = ValueConstant('two')
...     ATTR1_DUB = ValueConstant('one')
...
```

Here `FOO.ATTR1` and `FOO.ATTR1_DUB` have identical values. In this case method *`get_by_value()`* will return first constant with given value:

```
>>> FOO.get_by_value('one')
<constant 'FOO.ATTR1'>
```

If you need to get all constants with same value, use *`filter_by_value()`* method instead:

```
>>> FOO.filter_by_value('one')
[<constant 'FOO.ATTR1'>, <constant 'FOO.ATTR1_DUB'>]
```

And of course, you can access values of constants:

```
>>> TEAMS.RED.value
'#F00'
```

1.3.3 Verbose constants

How often do you do things like below?

```
>>> TYPE_FOO = 'foo'
>>> TYPE_BAR = 'bar'
>>> TYPES = (
...     (TYPE_FOO, "Some foo constant"),
...     (TYPE_BAR, "Some bar constant"),
... )
```

This is usually done to add verbose names to constants which you can use somewhere, e.g in HTML template:

```
<select>
{% for code, name in TYPES %}
  <option value='{{ code }}'>{{ name }}</option>
{% endfor %}
</select>
```

Okay. How about adding help text? Extend tuples? Or maybe create some `TYPES_DESCRIPTIONS` tuple? How far can you go and how ugly can you make it? Well, spare yourself from headache and use verbose constants *`VerboseConstant`* and *`VerboseValueConstant`*:

```
>>> from candv import VerboseConstant, Constants
>>> class TYPES(Constants):
...     FOO = VerboseConstant("Some foo constant", "help")
...     BAR = VerboseConstant(verbose_name="Some bar constant",
...                           help_text="some help")
```

Here you can access `verbose_name` and `help_text` attributes of constants:

```
>>> TYPES.FOO.verbose_name
'Some foo constant'
>>> TYPES.FOO.help_text
'help'
```

Now you can rewrite your code:

```
<select>
{% for constant in TYPES.constants() %}
  <option value='{{ constant.name }}' title='{{ constant.help_text }}'>
    {{ constant.verbose_name }}
  </option>
{% endfor %}
</select>
```

Same thing with values, just use `VerboseValueConstant`:

```
>>> from candv import VerboseValueConstant, Values
>>> class TYPES(Values):
...     FOO = VerboseValueConstant('foo', "Some foo constant", "help")
...     BAR = VerboseValueConstant('bar', verbose_name="Some bar constant",
...                                       help_text="some help")
...
>>> TYPES.FOO.value
'foo'
>>> TYPES.FOO.verbose_name
'Some foo constant'
>>> TYPES.FOO.help_text
'help'
```

Our sample HTML block will look almost the same and will use `value` attribute:

```
<select>
{% for constant in TYPES.constants() %}
  <option value='{{ constant.value }}' title='{{ constant.help_text }}'>
    {{ constant.verbose_name }}
  </option>
{% endfor %}
</select>
```

1.3.4 Hierarchies

candv library supports direct attaching of a group of constants to another constant to create hierarchies. A group can be created from any constant and any container can be used to store children. You may already saw this in the *introduction chapter*, but let's examine simple example:

```
>>> from candv import Constants, SimpleConstant
>>> class TREE(Constants):
...     LEFT = SimpleConstant().to_group(Constants,
...         LEFT=SimpleConstant(),
...         RIGHT=SimpleConstant(),
...     )
...     RIGHT = SimpleConstant().to_group(Constants,
...         LEFT=SimpleConstant(),
...         RIGHT=SimpleConstant(),
...     )
...
```

Here the key point is *to_group()* method which is avaivable for every constant. It accepts class that will be used to construct new container and any number of constant instances passed as keywords. You can access any group as any usual constant and use it as any usual container at the same time:

```
>>> TREE.LEFT
<constants group 'TREE.LEFT'>
>>> TREE.LEFT.name
'LEFT'
>>> TREE.LEFT.full_name
'TREE.LEFT'
>>> TREE.LEFT.constant_class
<class 'candv.base.Constant'>
>>> TREE.LEFT.names()
['LEFT', 'RIGHT']
>>> TREE.LEFT.LEFT
<constant 'TREE.LEFT.LEFT'>
>>> TREE.LEFT.LEFT.full_name
'TREE.LEFT.LEFT'
>>> TREE.LEFT.LEFT.container
<constants group 'TREE.LEFT'>
```

1.3.5 Exporting

New in version 1.3.0.

You can convert constants and containers into Python primitives for further serialization, for example, into JSON.

Use `to_primitive()` method of constants and containers to do that.

Simple constants

Let's see how it works with *simple constants*:

```
>>> STATUS.SUCCESS.to_primitive()
{'name': 'SUCCESS'}
>>> STATUS.to_primitive()
{'items': [{'name': 'SUCCESS'}, {'name': 'FAILURE'}], 'name': 'STATUS'}
```

By default `to_primitive()` returns a `dict` which contains at least a `name`. In addition, containers have `list` of their `items`.

Verbose constants

Verbose constants work same way:

```
>>> TYPES.FOO.to_primitive()
{'help_text': 'help', 'verbose_name': 'Some foo constant', 'name': 'FOO'}
```

Valued constants

You can do that with *valued constants* as well:

```
>>> TEAMS.RED.to_primitive()
{'name': 'RED', 'value': '#F00'}
```

Note: Remember: values of constants are out of scope of this library.

You can use anything as value of your constants, but converting values into primitives is almost up to you.

If your value is `callable`, `candv` will call it to get it's value. If your value has `isoformat()` method (`date`, `time`, etc.), `candv` will call it either. Everything else is supposed to be a primitive.

It is unlikely that you will use something complex, but if you will, than it's strongly recommended to implement *a custom constant class* with *custom support of exporting*.

Hierarchies

Hierarchies can be converted to primitives also:

```
>>> class FOO(Constants):
...     A = SimpleConstant()
...     B = VerboseValueConstant(
...         value=10,
...         verbose_name="Constant B",
...         help_text="Just a group with verbose name"
...     ).to_group(
...         group_class=Constants,
...         C=SimpleConstant(),
...         D=SimpleConstant(),
...     )
...
```

(continues on next page)

(continued from previous page)

```
>>> from pprint import pprint
>>> pprint(FOO.B.to_primitive())
    {'help_text': 'Just a group with verbose name',
     'items': [{'name': 'C'}, {'name': 'D'}],
     'name': 'B',
     'value': 10,
     'verbose_name': 'Constant B'}
```

As you can see, result is a mix of constant and container.

1.4 Customization

If all you've seen before is not enough for you, then you can create your own constants and containers for them. Let's see some examples.

1.4.1 Custom constants

Imagine you need to create some constant class. For example, you need to define some operation codes and have ability to create come commands with arguments:

```
>>> from candv import ValueConstant
>>> class Opcode(ValueConstant):
...     def compose(self, *args):
...         chunks = [self.value, ]
...         chunks.extend(args)
...         return '/'.join(map(str, chunks))
...
```

So, just a class with a method. Nothing special. You can use it right now:

```
>>> from candv import Values
>>> class OPERATIONS(Values):
...     REQ = Opcode(100)
...     ACK = Opcode(200)
...
>>> OPERATIONS.REQ.compose(1, 2, 3, 4, 5)
'100/1/2/3/4/5'
```

1.4.2 Adding support of groups

Well, everything looks fine. But what about creating a group from our new constants?

Note: If you don't know what this means, see *Hierarchies*.

So, firstly, let's create some constant:

```
>>> class FOO(Values):
...     BAR = Opcode(300).to_group(Values,
...         BAZ = Opcode(301),
...     )
```

And now let's check it:

```
>>> FOO.BAR.compose(1, 2, 3)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'FOO.BAR' object has no attribute 'compose'
>>> FOO.BAR.BAZ.compose(4, 5, 6)
'301/4/5/6'
```

Oops! Our newborn group does not have a `compose` method. Don't give up! We will add it easily, but in a special manner. Let's redefine our `Opcode` class:

```
>>> class Opcode(ValueConstant):
...     def compose(self, *args):
...         chunks = [self.value, ]
...         chunks.extend(args)
...         return '/'.join(map(str, chunks))
...     def merge_into_group(self, group):
...         super(Opcode, self).merge_into_group(group)
...         group.compose = self.compose
...
>>> class FOO(Values):
...     BAR = Opcode(300).to_group(Values,
...         BAZ = Opcode(301),
...     )
...
>>> FOO.BAR.compose(1, 2, 3)
'300/1/2/3'
```

Here the key point is `merge_into_group` method, which redefines *candv.base.Constant.merge_into_group()*. Firstly, it calls method of the base class, so that internal mechanisms can be initialized. Then it sets a new attribute `compose` which is a reference to `compose` method of our `Opcode` class.

Note: Be careful with attaching methods of existing objects to another objects. Maybe it will be better for you to use some *lambda* or to define some method within `merge_into_group`.

1.4.3 Adding support of exporting

If your constant stores some complex objects, then it's strongly recommended to provide support of exporting for them (see *Exporting*).

Do do that, you need to define `to_primitive()` method for your class. Example:

```
>>> from fractions import Fraction
>>> from pprint import pprint
>>> from candv import SimpleConstant, Constants
>>>
>>> class FractionConstant(SimpleConstant):
...     def __init__(self, value):
...         super(FractionConstant, self).__init__()
...         self.value = value
...
...     def to_primitive(self, context=None):
...         primitive = super(FractionConstant, self).to_primitive(context)
...         primitive.update({
```

(continues on next page)

(continued from previous page)

```
...                 'numerator': self.value.numerator,
...                 'denominator': self.value.denominator
...             })
...             return primitive
...
>>> class Fractions(Constants):
...     one_half = FractionConstant(Fraction(1, 2))
...     one_third = FractionConstant(Fraction(1, 3))
...
>>> Fractions.one_half.to_primitive()
{'denominator': 2, 'numerator': 1, 'name': 'one_half'}
>>> pprint(Fractions.to_primitive())
{'items': [{'denominator': 2, 'name': 'one_half', 'numerator': 1},
           {'denominator': 3, 'name': 'one_third', 'numerator': 1}],
 'name': 'Fractions'}
```

Note: This example is quite hypothetical and it's intended just to show implementation of custom `to_primitive()` method.

The plot in a nutshell:

1. Define `to_primitive()` method which accepts `context` argument.
2. Call parent's method and get primitive.
3. Update that primitive with your data, which may depend on context.
4. Return updated primitive.

Same can be applied to *custom constant containers* as well.

1.4.4 Adding verbosity

If you need to add verbosity to your constants, just use *VerboseMixin* mixin as the first base of your own class:

```
>>> from candv import VerboseMixin, SimpleConstant
>>> class SomeConstant(VerboseMixin, SimpleConstant):
...     def __init__(self, arg1, agr2, verbose_name=None, help_text=None):
...         super(SomeConstant, self).__init__(verbose_name=verbose_name,
...                                            help_text=help_text)
...         self.arg1 = arg1
...         self.arg2 = arg2
...
```

Note: Here note, that during call of `__init__` method of the super class, you need to pass `verbose_name` and `help_text` as keyword arguments.

1.4.5 Custom containers

To define own container, just derive new class from existing containers, e.g. from `Constants` or *Values*:

```
>>> class FOO(Values):
...     constant_class = Opcode
...
...     @classmethod
...     def compose_all(cls, *args):
...         return '!'.join(map(lambda x: x.compose(*args), cls.constants()))
...
```

Here `constant_class` attribute defines top-level class of constants. Instances whose class is more general than `constant_class` will be invisible to container (see *constant_class*). Our new method `compose_all` just joins compositions of all its opcodes.

Note: Since *1.2.0* you can use *with_constant_class()* mixin factory to make definitions of your containers more readable, e.g.:

```
>>> from candv import with_constant_class
>>> class FOO(with_constant_class(Opcode), Values):
...
...     @classmethod
...     def compose_all(cls, *args):
...         return '!'.join(map(lambda x: x.compose(*args), cls.constants()))
...
```

This will produce the same class as above.

Now it's time to use new container:

```
>>> class BAR(FOO):
...     REQ = Opcode(1)
...     ACK = Opcode(2)
...
...     @classmethod
...     def decompose(cls, value):
...         chunks = value.split('/')
...         opcode = int(chunks.pop(0))
...         constant = cls.get_by_value(opcode)
...         return constant, chunks
```

Here we add new method `decompose` which takes a string and decomposes it into tuple of opcode constant and its arguments. Let's test our conainer:

```
>>> BAR.compose_all(500, 600, 700)
'1/500/600/700!2/500/600/700'
>>> BAR.decompose('1/100/200')
(<constant 'BAR.REQ'>, ['100', '200'])
```

Seems to be OK.

1.5 Misc

This chapter covers miscellaneous things which are not related to the library usage.

1.5.1 Tests

See the output of tests execution at Travis CI.

If you need to run tests locally, you need to have nose installed. Then just run

```
$ nosetests
```

to run all tests inside the project.

Visit Coveralls to see the tests coverage online.

If you need to see coverage locally, install coverage additionally. Then run:

```
$ coverage run `which nosetests` --nocapture && coverage report -m
```

1.5.2 Building docs

If you need to have a local copy of these docs, you will need to install Sphinx and make. Then:

```
$ cd docs
$ make html
```

This will render docs in `HTML` format to `docs/_build/html` directory.

To see all available output formats, run:

```
$ make help
```

CHAPTER 2

Changelog

You can click a version name to see a diff with the previous one.

- 1.3.1 (Aug 1, 2015)
  1. Fix the way constants are compared. Now comparison is based on constant's `full_name` attribute (issue #11).
- 1.3.0 (Dec 31, 2014)
  1. Implement `to_primitive()` method, which can be used for serialization, for example, into JSON (issue #1). See *usage* and *customization* for more info.
- 1.2.0 (Oct 11, 2014)
  1. Core classes were significantly refactored.
  2. `constant_class` uses *`Constant`* as default value (instead of `None`, see *Custom containers* for more info).
  3. Support of groups was reimplemented: now they are classes just as other constants containers (earlier groups were instances of patched containers). So, groups automatically gain all those attributes and methods which usual containers have.
  4. Constant's `container` attribute was made public. Groups of constants now have it too (see *Hierarchies*).
  5. API of containers was made really close to API of Python's `dict` (*see usage* for more info):
     - `__getitem__`, `__contains__`, `__len__` and `__iter__` magic methods were implemented;
     - `contains` method was renamed to `has_name`;
     - `get_by_name` method was removed in favor of `__getitem__` method.
     - `get` method with support of default value was introduced.
  6. All objects (contaners, groups and constants) now have `name` and `full_name` attributes. This may be useful if you use names of constants as key values (e.g. for Redis).
  7. Also, all objects have good `repr` now.

8. Mixin factory *with_constant_class()* was introduced. It may help you to define more readable containers.
9. A potential bug of uninitialized unbounded constants was fixed. Unbounded constant is an instance of a class which is differ from container's `constant_class` or its subclasses. This is unnatural case, but if you really need it, it will not break now.
10. Exception messages are more informative now.
11. Tests were moved out the package.
12. *Introductory documentation* was improved. Other docs were updated too.

- 1.1.2 (Jul 6, 2014)
  - add `values` and `itervalues` attributes to `ConstantsContainer`.
- 1.1.1 (Jun 21, 2014)
  - switch license from `GPLv2` to `LGPLv3`.
- 1.1.0 (Jun 21, 2014)
  1. remove `Choices` container, move it to django-candv-choices library;
  2. update docs and fix typos;
  3. strip utils from requirements.
- **1.0.0 (Apr 15, 2014)** Initial version.

CHAPTER 3

Sources

Feel free to explore, fork or contribute:

https://github.com/oblalex/candv

CHAPTER 4

Authors

Alexander Oblovatniy (@oblalex) created `candv` and these fine people have contributed.

CHAPTER 5

Modules

5.1 candv

5.1.1 candv package

candv.base module

This module defines base constant and base container for constants. All other stuff must be derived from them.

Each container has *constant_class* attribute. It specifies class of constants which will be defined within contaner.

class candv.base.**Constant**

Bases: object

Base class for all constants. Can be merged into a container instance.

Variables ***name*** (*str*) – constant's name. Is set up automatically and is equal to the name of container's attribute

full_name

merge_into_group(*group*)

Called automatically by container after group construction.

Note: Redefine this method in all derived classes. Attach all custom attributes and methods to the group here.

Parameters **group** – an instance of *ConstantsContainer* or it's subclass this constant will be merged into

Returns None

to_group(*group_class*, ***group_members*)
Convert a constant into a constants group.

Parameters

- **group_class** (*class*) – a class of group container which will be created
- **group_members** – unpacked dict which defines group members.

Returns a lazy constants group which will be evaluated by container. During group evaluation *merge_into_group()* will be called.

Example:

```
from candv import Constants, SimpleConstant

class FOO(Constants):
    A = SimpleConstant()
    B = SimpleConstant().to_group(
        group_class=Constants,
        B2=SimpleConstant(),
        B0=SimpleConstant(),
        B1=SimpleConstant(),
)
```

to_primitive(*context=None*)
New in version 1.3.0.

class candv.base.**ConstantsContainer**
Bases: object

Base class for creating constants containers. Each constant defined within container will remember it's creation order. See an example in `constants()`.

Variables ***constant_class*** – stores a class of constants which can be stored by container. This attribute **MUST** be set up manually when you define a new container type. Otherwise container will not be initialized. Default: `None`

Raises **TypeError** – if you try to create an instance of container. Containers are singletons and they cannot be instantiated. Their attributes must be used directly.

constant_class
Defines a top-level class of constants which can be stored by container

alias of *Constant*

full_name = 'ConstantsContainer'

name = 'ConstantsContainer'

candv.base.**with_constant_class**(*the_class*)
A mixin factory which allows to set constant class for constants container outside container itself. This may help to create more readable container definition, e.g.:

```
>>> from candv import Constants, SimpleConstant, with_constant_class
>>>
>>> class SomeConstant(SimpleConstant):
...     pass
...
>>> class FOO(with_constant_class(SomeConstant), Constants):
...     A = SomeConstant()
...     B = SomeConstant()
```

(continues on next page)

(continued from previous page)

```
...
>>> FOO.constant_class
<class '__main__.SomeConstant'>
```

Module contents

This module provides ready-to-use classes for constructing custom constants.

class candv.**ValueConstant**(*value*)

Bases: *candv.base.Constant*

Extended version of `SimpleConstant` which provides support for storing values of constants.

Parameters **value** – a value to attach to constant

Variables **value** – constant's value

merge_into_group(*group*)

Redefines *merge_into_group()* and adds `value` attribute to the target group.

to_primitive(*context=None*)

New in version 1.3.0.

class candv.**Values**

Bases: `candv.base.ConstantContainerMixin`, *candv.base.ConstantsContainer*

Constants container which supports getting and filtering constants by their values, listing values of all constants in container.

classmethod filter_by_value(*value*)

Get all constants which have given value.

Parameters **value** – value of the constants to look for

Returns list of all found constants with given value

full_name = 'Values'

classmethod get_by_value(*value*)

Get constant by its value.

Parameters **value** – value of the constant to look for

Returns first found constant with given value

Raises **ValueError** – if no constant in container has given value

classmethod itervalues()

Same as *values()* but returns an interator.

Note: Overrides `itervalues()` since 1.1.2.

---

name = 'Values'

classmethod values()

List values of all constants in the order they were defined.

Returns `list` of values

Example:

```
>>> from candv import Values, ValueConstant
>>> class FOO(Values):
...     TWO = ValueConstant(2)
...     ONE = ValueConstant(1)
...     SOME = ValueConstant("some string")
...
>>> FOO.values()
[2, 1, 'some string']
```

Note: Overrides `values()` since 1.1.2.

class candv.**VerboseConstant**(*verbose_name=None*, *help_text=None*)

Bases: *candv.VerboseMixin*, *candv.base.Constant*

Constant with optional verbose name and optional description.

Parameters

- **verbose_name** (*str*) – optional verbose name of the constant
- **help_text** (*str*) – optional description of the constant

Variables

- **verbose_name** (*str*) – verbose name of the constant. Default: `None`
- **help_text** (*str*) – verbose description of the constant. Default: `None`

class candv.**VerboseMixin**(**args*, ***kwargs*)

Bases: `object`

Provides support of verbose names and help texts. Must be placed at the left side of non-mixin base classes due to Python's MRO. Arguments must be passed as kwargs.

Parameters

- **verbose_name** (*str*) – optional verbose name
- **help_text** (*str*) – optional description

Example:

```
class Foo(object):

    def __init__(self, arg1, arg2, kwarg1=None):
        pass


class Bar(VerboseMixin, Foo):

    def __init__(self, arg1, arg2, verbose_name=None, help_text=None,␣
↪kwarg1=None):
        super(Bar, self).__init__(arg1, arg2, verbose_name=verbose_name, help_
↪text=help_text, kwarg1=kwarg1)
```

merge_into_group(*group*)

Redefines *merge_into_group()* and adds `verbose_name` and `help_text` attributes to the target group.

to_primitive(*context=None*)

New in version 1.3.0.

class candv.**VerboseValueConstant**(*value*, *verbose_name=None*, *help_text=None*)

Bases: *candv.VerboseMixin*, *candv.ValueConstant*

A constant which can have both verbose name, help text and a value.

Parameters

- **value** – a value to attach to the constant
- **verbose_name** (*str*) – optional verbose name of the constant
- **help_text** (*str*) – optional description of the constant

Variables

- **value** – constant's value
- **verbose_name** (*str*) – verbose name of the constant. Default: `None`
- **help_text** (*str*) – verbose description of the constant. Default: `None`

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index