
candv Documentation

Release 1.0.0

Alexander Oblovatniy

Oct 30, 2020

Contents

1	Contents	3
1.1	Introduction	3
1.2	Installation	5
1.3	Usage	5
1.4	Customization	9
2	Modules	13
2.1	candv	13
3	Project's repository	15
4	Indices and tables	17
	Python Module Index	19
	Index	21

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

1.1 Introduction

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?

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

```
>>> class BAR(Constants):
...     ONE = SimpleConstant()
...     TWO = SimpleConstant()
...     NINE = SimpleConstant()
...
>>> BAR.ONE
<constant 'BAR.ONE'>
>>> BAR.names()
['ONE', 'TWO', 'NINE']
>>> 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'>)]
>>> BAR.contains('NINE')
True
>>> BAR.get_by_name('TWO')
<constant 'BAR.TWO'>
>>> BAR.get_by_name('TWO').name
'TWO'
```

Too simple for you? Watchout:

```
>>> class FOO(Constants):
...     """
...     Some group of constants showing 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
...     SOME = ValueConstant(['1', 4, '2'])
...     #: yet another named constant with another value, verbose name and
...     #: description
...     SOME_VERBOSE = VerboseValueConstant("some value",
...                                          "some string",
...                                          "this is just some string")
...     #: subgroup with name
...     SUBGROUP = SimpleConstant().to_group(Values,
...     SIX=ValueConstant(6),
...     SEVEN=ValueConstant("S373N"),
...     )
...     #: subgroup with name, value and verbose name
...     MEGA_SUBGROUP = VerboseValueConstant(100500,
...     "mega subgroup").to_group(Values,
...     hey=ValueConstant(1),
...     #: subgroup inside another subgroup. How deep can you go?
...     yay=ValueConstant(2).to_group(Constants,
...     OK=SimpleConstant(),
...     ERROR=SimpleConstant(),
...     ),
...     )
...
>>> FOO.names()
['ONE', 'BAR', 'BAZ', 'QUX', 'SOME', 'SOME_VERBOSE', 'SUBGROUP', 'MEGA_SUBGROUP']
>>> FOO.BAR.verbose_name
'bar constant'
>>> FOO.BAZ.help_text
'description of baz constant'
>>> FOO.QUX.value
4
>>> FOO.SOME_VERBOSE.value, FOO.SOME_VERBOSE.verbose_name
('some value', 'some string')
>>> FOO.SUBGROUP
<constant 'FOO.SUBGROUP'>
>>> FOO.SUBGROUP.names()
['SIX', 'SEVEN']
>>> FOO.SUBGROUP.SIX.value
6
>>> FOO.SUBGROUP.get_by_value('S373N')
<constant 'FOO.SUBGROUP.SEVEN'>
>>> FOO.MEGA_SUBGROUP.value
100500
>>> FOO.MEGA_SUBGROUP.name
```

(continues on next page)

(continued from previous page)

```
'MEGA_SUBGROUP'
>>> FOO.MEGA_SUBGROUP.verbose_name
'mega subgroup'
>>> FOO.MEGA_SUBGROUP.names()
['hey', 'yay']
>>> FOO.MEGA_SUBGROUP.get_by_value(2).ERROR
<constant 'FOO.MEGA_SUBGROUP.yay.ERROR'>
```

Okay, this looks like a big mess, but it shows all-in-one. If you need something simple, you can have it.

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 attribute 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 their 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:

```
>>> 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 Constant class or its subclasses. SimpleConstant is just an alias to candv.base.Constant.

Access some constant:

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

Access its name:

```
>>> STATUS.SUCCESS.name
'SUCCESS'
```

List names of all constants in the container:

```
>>> STATUS.names()
['SUCCESS', 'FAILURE']
```

List all constants in the container:

```
>>> STATUS.constants()
[<constant 'STATUS.SUCCESS'>, <constant 'STATUS.FAILURE'>]
```

Check whether the container has constant with a given name:

```
>>> STATUS.contains('SUCCESS')
True
>>> STATUS.contains('XXX')
False
```

Get constant by name or get a `KeyError`:

```
>>> STATUS.get_by_name('FAILURE')
<constant 'STATUS.FAILURE'>
>>> STATUS.get_by_name('XXX')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "candv/base.py", line 316, in get_by_name
    .format(name, cls.__name__)
KeyError: "Constant with name 'XXX' is not present in '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(0)
...     RED = ValueConstant(1)
...     BLUE = ValueConstant(2)
... 
```

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

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

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

```
>>> INVALID.constants()
[]
```

Ok, let's get back to our TEAMS. You can access values of constants:

```
>>> TEAMS.RED.value
1
```

Get constant by its value or get `ValueError`:

```
>>> TEAMS.get_by_value(2)
<constant 'TEAMS.BLUE'>
>>> TEAMS.get_by_value(-1)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    File "candv/__init__.py", line 146, in get_by_value
      value, cls.__name__)
ValueError: Value '-1' is not present in 'TEAMS'
```

List all values inside the container:

```
>>> TEAMS.values()
[0, 1, 2]
```

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'>]
```

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` as 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, except 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 Choices

If you are familiar with Django's [field choices](#) then you may find Choices container helpful:

```
>>> class TYPES(Choices):
...     FOO = VerboseConstant('Foooo')
...     BAR = VerboseConstant('Barrrr')
... 
```

It accepts instances of VerboseConstant class or its subclasses and can build tuple of tuples with names and verbose names of constants:

```
>>> TYPES.choices()
(('FOO', 'Foooo'), ('BAR', 'Barrrr'))
```

1.3.5 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 [introduction](#), 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 available 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.LEFT
<constant 'TREE.LEFT.LEFT'>
>>> TREE.RIGHT.names()
['LEFT', 'RIGHT']
```

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 some 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 Constants
>>> class OPERATIONS(Constants):
...     REQ = Opcode(100)
...     ACK = Opcode(200)
...
>>> OPERATIONS.REQ.compose(1, 9, 3, 2, 0)
'100/1/9/3/2/0'
```

1.4.2 Providing groups support

Well, everything looks fine. But what about creating a group from our new constants? First, let's create some:

```
>>> class FOO(Constants):
...     BAR = Opcode(300).to_group(Constants,
...     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(Constants):
...     BAR = Opcode(300).to_group(Constants,
...     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 define `somemethod` within `merge_into_group`.

1.4.3 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
... 
```

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

1.4.4 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 `candv.base.ConstantsContainer.constant_class`). Our new method `compose_all` just joins compositions of all its opcodes.

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 it's arguments. Let's test our container:

```
>>> BAR.compose_all(1, 9, 30)
'1/1/9/30!2/1/9/30'
>>> BAR.decompose('1/100/200')
(<constant 'BAR.REQ'>, ['100', '200'])
```

Seems to be OK.

2.1 candv

2.1.1 candv package

candv.base module

Module contents

CHAPTER 3

Project's repository

Feel free to explore, fork or contribute:

<https://github.com/oblalex/candv>

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- `search`

C

candv, [13](#)

C

candv (*module*), 13