QCOBJ a Python package to handle quantity-aware configuration files

Roberto Vidmar[1] **and Nicola Creati**[2]

[1]**OGS - Istituto Nazionale di Oceanografia e di Geofisica Sperimentale**
[2]**OGS - Istituto Nazionale di Oceanografia e di Geofisica Sperimentale**

Corresponding author:
Roberto Vidmar[1]

Email address: rvidmar@inogs.it

ABSTRACT

Configuration files are widely used by scientists and researchers to configure the parameters and initial settings for their computer programs.
We present here a Python package that adds physical quantities to these parameters and validates them against user defined specifications to ensure that they are in the correct range and eventually converted to the requested unit of measurement. The package contains also a graphical user interface class to display, edit configuration file content, and to compare them side by side highlighting their differences.

INTRODUCTION

Scientists often use configuration files (*cfg* files) to set the parameters and initial conditions for their computer programs or simulations. When these parameters are not limited to numbers or strings but represent physical quantities their unit of measure must be taken into account. Researchers are used to convert derived physical quantities by hand or with the help of some computer program but this operation slows down the process and is inherently error prone.

We developed a package to give an answer to this problem integrating unit of measure and hence dimensionality into parameters. This approach ensures that programs using this package will always get numbers in the requested range and in the correct unit of measure independently of the units used in the configuration file.

CODE DESIGN

Scientific work implies the writing of many lines of code and researchers try to capitalize their production creating reusable code that is driven by *cfg* files. These are essentially text files in which keys are associated to values and these can be numeric or symbolic. Usually comments can be added to explain the meaning of the keywords and other useful information for the end users.

The standard packages that allow Python programmers to take benefit of *cfg* files are ConfigParser (Langa, 2017) and ConfigObj (Foord and Larosa, 2017).

The main difference between them is that ConfigObj has the capability to validate a *cfg* file against a specification file called *configspec*. The *configspec* defines the allowed data types and ranges for the keywords and can set default values. Thus the *cfg* file just needs to specify values that differ from defaults.

However when programs use various physical quantities the user must take care of converting them to the *numbers* assigned in the *cfg* file that will be validated against the *configspec* specifications.

We found this process time consuming and error prone because it needs frequent review of correctly developed and tested functions to search for not existing bugs. Our solution was the integration of physical quantities into configuration files.

Many Python libraries exists that manage physical units manipulation but we found that Pint (Grecco, 2017) is the most easy to read and has a clean syntax to specify units. Pint integrates unit parsing: prefixed and pluralized forms of units are recognized without explicitly defining them. In other words: since the

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3418v1 | CC BY 4.0 Open Access | rec: 17 Nov 2017, publ: 17 Nov 2017

prefix kilo and the unit meter are defined, Pint understands kilometers. Pint can also handle units provided as strings and this capability was the main reason of our choice.

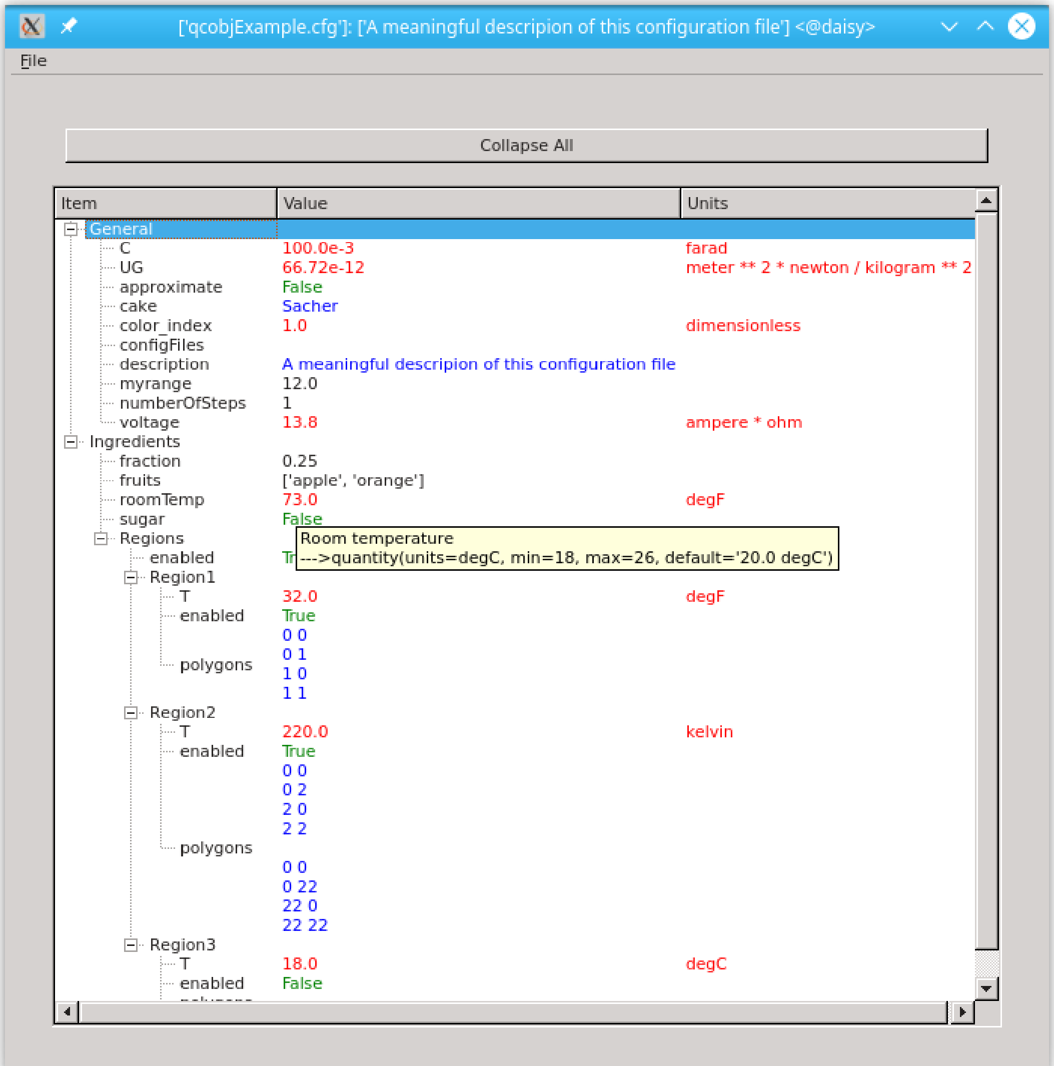

Figure 1. A simple application using *CfgGui* class.

Eventually we created a Python library, or more properly a *pacakge* to integrate ConfigObj and Pint and we called it called **QCOBJ**. QCOBJ is composed by three main classes.

*Q_*

This is the physical quantity container class. It provides all the methods to manipulate the strings that define physical quantities and supports conversion to and from different units. It implements also methods for their representation in clear human readable form. Q_ instances can be added or compared only if they have the same dimensionality.

QValidator

Validation is a transparent layer to access data stored as strings. The validation checks if the data is correct and converts it to the expected type.

The QValidator class is an extension of the original Validator class that understands the new syntax created for the physical quantities. Moreover it ensures that the values used in the *cfg* file are dimensionally

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3418v1 | CC BY 4.0 Open Access | rec: 17 Nov 2017, publ: 17 Nov 2017

correct and converts them to the units specified by *configspec*. If *configspec* defines also a minimum and/or a maximum value the QValidator raises errors when the user supplied values are out of range.

QConfigObj

It extends the ConfigObj class adding methods to integrate the QValidator. Default *cfg* files can be created when the validator instance is supplied to an empty instance of this class. QConfigObj has also methods for converting user defined quantities to the units used in configspec. For example a keyword defining a velocity in m/s in *configspec* can always be converted to m/s even if its value is set to knots in *cfg* or is set during program execution to any other velocity unit of measure.

QConfigObj has a reserved keyword *configFiles* that allows the inclusion of a list of files. Long *cfg*, hundreds or thousands lines, can be split into smaller units thus improving readability and reuse.

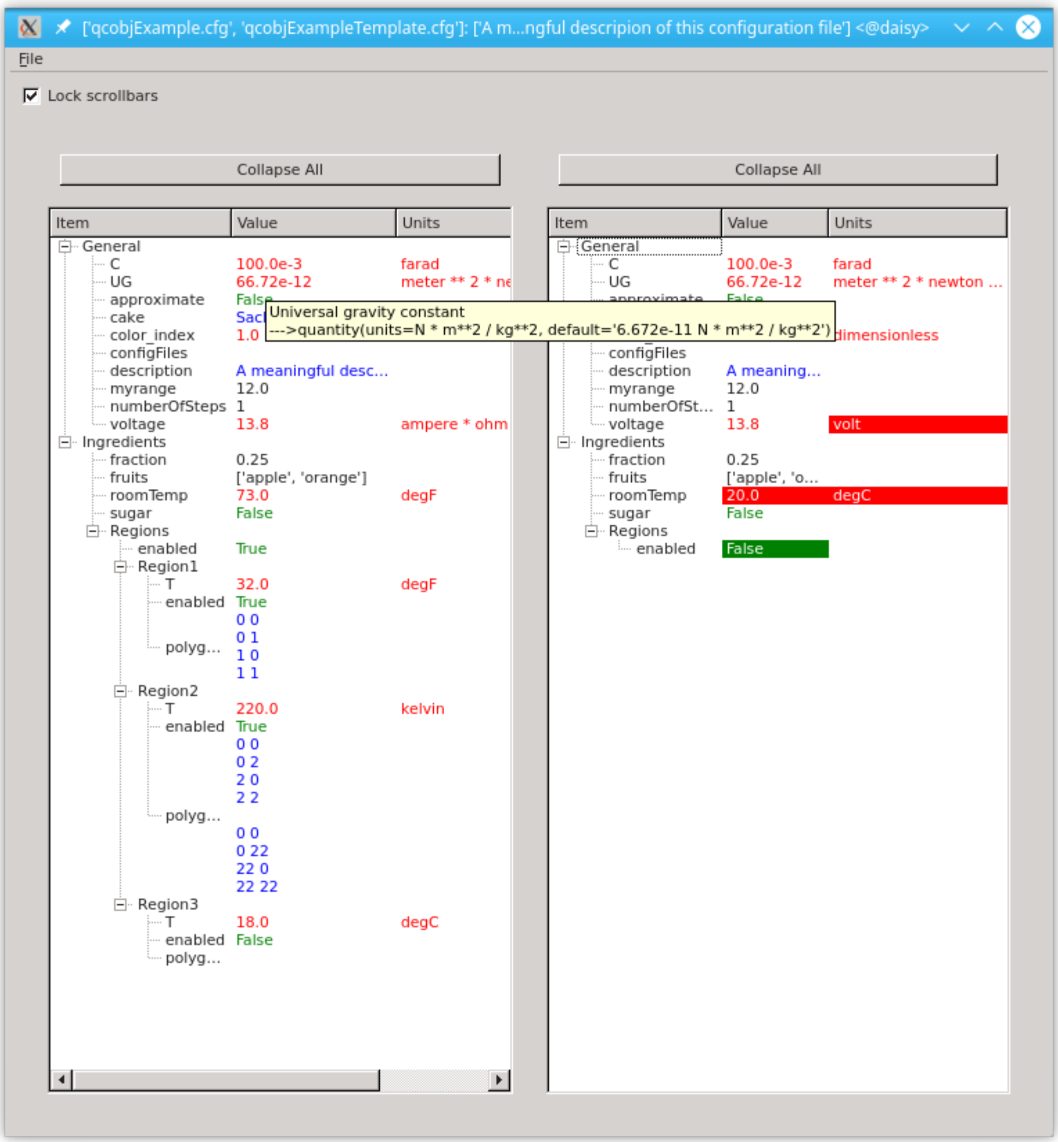

Figure 2. The expanded tree view of the comparison of two *cfg* files with the differences highlighted in reverse text/background colors.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3418v1 | CC BY 4.0 Open Access | rec: 17 Nov 2017, publ: 17 Nov 2017

Graphical User Interface (GUI)

While comparing long *cfg* files we noticed that spotting differences between them was quite difficult. Standard tools are available (Wikipedia, 2017a) to compare files but these highlight also difference in indentation and in comment lines hence we found them rather unusable. The *CfgGui* class included in the package defines a simple GUI to explore, edit and compare already defined configuration files.

We chose a tree diagram representation for the GUI as configuration files are organized in sections and these can be nested to any level. In Figure 1 there is an expanded tree view of a *cfg* file. Values are coloured according their data types: quantities in red, boolean in green, strings in blue. Other data types in black. *CfgGui* can display more than one *cfg* file side by side as in Figure 2 and allows the expansion and compression of every single section as well as synchronized scrolling. Comment lines are ignored, values are colored according to their data types and differences between them are highlighted reverting background and foreground colors. Hovering with the mouse over a value pops up a help tooltip window with the valid range for that parameter. Comparison of more than two files (3-way comparison) is also supported.

Our implementation uses the Model-View-Controller (MVC) (Wikipedia, 2017b) pattern and among the many GUi toolkits available from the Python literature (Alves, 2017) (Polo, 2017) we stick to PyQt/PySide (Riverbank Computing Limited, 2017) (The Qt Company, 2017) since they provide great flexibility and user control.

PyQt and PySide are almost identical from the user point of view, PyQt being a much more mature, efficient and stable project. On the other side PySide provides LGPL-licensed Python bindings for the Qt framework and this feature can be important when deciding how to distribute the software. *QCOBJ* includes a compatibility module to leave the user free to choose the preferred library at runtime.

USAGE

Since configuration files use physical quantities it is mandatory to create first a *configspec* file that defines the keywords and the valid data types allowed for each of them. The keywords can be organized into sections and subsections in a hierarchical form.

Cfg files can be written with any text editor but we provided a utility function *makeSpec* that helps building *configspec* sections with the correct syntax and indentation through a short Python script.

```
level = 1  # The hierarchical level of this section
secname = 'Ingredients'
subsection = collections.OrderedDict((
    ('sugar', (
        'Enable sugar',
        'boolean',
        False)),
    ('fruits', (
        'A list of exactly two fruits at your choice',
         'string_list 2 2',
        "apple, orange")),
    ('roomTemp', (
        'Room temperature',
        'degC, 18, 26',
        20.0)),
    ('fraction', (
        'Some decimal value (floats are welcome, as always)',
        'float, 0, 1',
        0.25)),
    ))
subspec = makeSpec(secname, subsection, level)
```

Listing 1. makeSpec.py - Python script to create a section using the makeSpec function.

Every section/subsection can be built filling an ordered Python dictionary in which each keyword is associated with a tuple. The last two elements of it are the range of valid values and the default while the remaining values will appear as comments in the *configspec*. The second last element defines also the type of the keyword that can be a physical quantity according to the *Pint* syntax. The keyword *roomTemp* of listing 1 al line 12 for example defines a temperature that can assume any value between 18 and 26 degrees Celsius with a default value of 20 °C. The *subsection* instance of the same listing at line 3 is then processed by *makeSpec* at line 21 of listing 1 leaving in the subspec string what appears in listing 2 with the correct syntax and proper indentation.

Listing 3 is an example of *configspec* file.

```
[Ingredients]
    # Enable sugar
    sugar = boolean(default=False)
    # A list of two fruits at your choice
    fruits = string_list(default=list(apple, orange))
    # Room temperature
    roomTemp = quantity(units=degC, min=18, max=26, default='20.0 degC')
    # Some decimal value (floats are welcome, as always)
    fraction = float(min=0, max=1, default=0.25)
```

Listing 2. Section created with the script of Listing 1.

```
1  #
2  # Header of this configspec file, date, authors and version
3  #
4  description = string(default='A meaningful descripion of this configuration file')
5  ...
6  voltage = quantity(units=V, min=0, max=100, default='13.8 V')
7  UG = quantity(units=N * m**2 / kg**2, default='6.672e-11 N * m**2 / kg**2')
8  # List of more configuration files blank separated
9  configFiles = string(default='')
10     [Ingredients]
11         ...
12         roomTemp = quantity(units=degC, min=18, max=26, default='20.0 degC')
13         [[Regions]]
14             ...
15             # More Sections like this can be added with different names
16             [[[__many__]]]
17                 ...
18                 # Constant Temperature
19                 T = quantity(units=degC, default='18.0 degC')
```

Listing 3. my_configspec.cfg - Example of configspec file.

The *quantity* in line 7 defines the unit of mesure for the keyword *UG* and hence its physical dimensions. The accepted quantities as well as their alias are defined in the *Pint* unit definitions file but the user can easily edit this text file to add any other unit needed.

Lower and upper limits for each quantity can be specified like in line 6 leaving to the *QValidator* class the duty to check that the value defined in the actual configuration file is in the defined range.

The special section *__many__* defines sub-sections to be validated using the same keywords and specification.

The creation of this *configspec* can be speeded using a template file instead of writing it from scratch. Such a file can be generated form its *configspec* assigning to all keywords their default values with the following command:

```
python -c 'from qcobj.qconfigobj import QConfigObj;
    template = QConfigObj("my_configuration_file.cfg", configspec="
        configspec.cfg");
    template.write()'
```

The just created *my_configuration_file.cfg* can later be tailored by the user with less effort.

The physical quantities designated in *configspec* can now be assigned in any unit of measurement provided its value is dimensionally correct and, if converted to the units already specified in *configspec*, it is in the allowed range (if defined). For example voltage can be assigned as

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3418v1 | CC BY 4.0 Open Access | rec: 17 Nov 2017, publ: 17 Nov 2017

```
voltage = 12.6 V # or
voltage = 12.6 volt # or
voltage = 12.6 ohm * ampere
```

Pressure can be expressed in pascal or Pa or newton / m**2 or force_kilogram / cm**2.

Configuration file content is accessible to Python scripts in this way:

```
from qcobj import qconfigobj
qcobj = qconfigobj.QConfigObj("my_configuration_file.cfg",
        configspec="my_configspec.cfg")
```

Once the QCOBJ object has been instantiated, physical quantities in it can be accessed with the standard ConfigObj syntax:

```
section = qcobj['Ingredients']
roomTemperature = section['roomTemp']
# Being a validated quantity, roomTemperature can be converted by
# the script to any other units and eventually its value is
# available for computation
absoluteRoomTemperature = roomTemperature.to('K').magnitude
# Increase temperature
absoluteRoomTemperature += 5.2
# Convert now this **number** back to its physical quantity
# as in configspec: in this case degC **regardless** of the units used in
# my_configuration_file.cfg
newTempQuantity = qconfigobj.qLike(
    absoluteRoomTemperature, section, 'roomTemp')
```

More utility functions are available in the qconfigobj package to simplify the use of quantities from Python scripts. Full documentation of the package and of all the classes and functions in it is available both in HTML and pdf format. A few examples are also available to learn the basic usage.

FINAL REMARKS

QCOBJ has been evaluated and tuned during the developemnt of a geodynamic parallel numerical simulation suite (Nicola Creati et al., 2015). We found it of great help in managing the hundreds of physical quantities (parameters) needed for the computation and its use solved the troublesome process of units conversions leaving more time available for the research. *CfgGui* was necessitous for comparing *cfg* files that drove models with hundreds of parameters.

ACKNOWLEDGMENTS

We thank Hernán Grecco for the time and effort he spent in helping us to achieve this result.

REFERENCES

Alves, M. (2017). *GuiProgramming - Python Wiki*. https://wiki.python.org/moin/GuiProgramming [Accessed: 20 September"].

Foord, M. and Larosa, N. (2017). *Reading and Writing Config Files*. http://www.voidspace.org.uk/python/configobj.html [Accessed: 20 September"].

Grecco, H. E. (2017). *Pint: makes units easy*. https://pint.readthedocs.io/ [Accessed: 20 September"].

Langa, Ł. (2017). *14.2. configparser - Configuratin file parser*. https://docs.python.org/3/library/configparser.html# [Accessed: 20 September"].

Nicola Creati, Roberto Vidmar, and Paolo Sterzai (2015). Geodynamic simulations in HPC with Python. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 158 – 163.

Polo, G. (2017). *PyGTK, PyQT, Tkinter and wxPython comparison*. http://ojs.pythonpapers.org/index.php/tpp/article/download/61/57 [Accessed: 20 September"].

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3418v1 | CC BY 4.0 Open Access | rec: 17 Nov 2017, publ: 17 Nov 2017

Riverbank Computing Limited (2017). *Riverbank — Software — PyQt — What is PyQt?* https://riverbankcomputing.com/software/pyqt/ [Accessed: 20 September"].

The Qt Company (2017). *PySide - Qt Wiki*. https://wiki.qt.io/PySide [Accessed: 26 October"].

Wikipedia (2017a). *Comparison of file comparison tools - Wikipedia*. https://en.wikipedia.org/wiki/Comparison_of_file_comparison_tools [Accessed: 20 September"].

Wikipedia (2017b). *Model-view-controller - Wikipedia*. https://en.wikipedia.org/wiki/Model-view-controller [Accessed: 20 September"].

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3418v1 | CC BY 4.0 Open Access | rec: 17 Nov 2017, publ: 17 Nov 2017