Standard Decimal Notation Documentation

Release 0.0.1

Shawn Pringle

September 12, 2014

CONTENTS

Contents:

... module:: qsdn

platform Unix, Windows

synopsis This module allows for parsing, validation and production of numeric literals, written with thousand separators through out the number. Often underlying system libraries for working with locales neglect to put thousand separators (commas) after the decimal place or they sometimes use scientific notation. The classes inherit from the Qt classes for making things less complex.

Thousand separators in general will not always be commas but instead will be different according to the locale settings. In Windows for example, the user can set his thousand separator to any character. Support for converting strings directly to Decimals and from Decimals to strings is included.

Also, numbers are always expressed in standard decimal notation.

Care has been taken to overload all of the members in a way that is consistent with the base class QLocale and QValidator.

This module requires PyQt4. It doesn't need but it can use KDE. If KDE and PyKDE are installed on your system, KDE's settings for thousands separator and decimal symbol will be used. Otherwise the system's locale settings will be used to determine these values.

class qsdn.**QSDNLocale**(*_name=None*, *p_mandatory_decimals=Decimal('0')*, *p_maximum_decimals=Decimal('Infinity')*)

For a QSDNlocale, locale: To get the Decimal of a string, s, use:

(d, ok) = locale.toDecimal(s, 10)

The value d is your decimal, and you should check ok before you trust d.

To get the string representation use:

s = locale.toString(d)

By default QSDNLocale will use the settings specified in the control panel. This is guaranteed to be true for Mac OS, Windows and KDE-GUIs.

static c()
Returns the C locale. In the C locale, to* routines will not accept group separtors and do not produce them.

static system()
Returns the system default for QSDNLocale.

toDecimal(*s*, *base=0*)
This creates a decimal representation of s.

It returns an ordered pair. The first of the pair is the Decimal number, the second of the pair indicates whether the string had a valid representation of that number. You should always check the second of the ordered pair before using the decimal returned.

Note Make sure you use 10 as the second argument or it may interpret the string as octal!

Like the other to* functions of QLocale as well as this class QSDNLocale, interpret a a string and parse it and return a Decimal. The base value is used to determine what base to use.

If base is not set, numbers such as '0777' will be interpreted as octal. The string '0x33' will be interpreted as hexadecimal and '777' will be interpreted as a decimal. It is done this way so this works like toLong, toInt, toFloat, etc... Leading and trailing whitespace is ignored.

toDouble(*s*)
This creates a floating point representation of s.

It returns an ordered pair. The first of the pair is the number, the second of the pair indicates whether the string had a valid representation of that number. You should always check the second of the ordered pair before using the number returned.

toFloat (*s*)
This creates a floating point representation of s.

It returns an ordered pair. The first of the pair is the number, the second of the pair indicates whether the string had a valid representation of that number. You should always check the second of the ordered pair before using the number returned.

toLongLong (*s*, *base=0*)
This creates a numeric representation of s.

It returns an ordered pair. The first of the pair is the number, the second of the pair indicates whether the string had a valid representation of that number. You should always check the second of the ordered pair before using the number returned.

note Make sure you use 10 as the second argument or it may interpret the string as octal!

If base is not set, numbers such as '0777' will be interpreted as octal. The string '0x33' will be interpreted as hexadecimal and '777' will be interpreted as a decimal. It is done this way so this works like toLong, toInt, toFloat, etc...

Leading and trailing whitespace is ignored.

toShort (*s*, *base=0*)
This creates a numeric representation of s.

It returns an ordered pair. The first of the pair is the number, the second of the pair indicates whether the string had a valid representation of that number. You should always check the second of the ordered pair before using the number returned.

note Make sure you use 10 as the second argument or it may interpret the string as octal!

If base is not set, numbers such as '0777' will be interpreted as octal. The string '0x33' will be interpreted as hexadecimal and '777' will be interpreted as a decimal. It is done this way so this works like toLong, toInt, toFloat, etc...

Leading and trailing whitespace is ignored.

toString (*x*, *arg2=None*, *arg3=None*)
Convert any given Decimal, double, Date, Time, int or long to a string.

Numbers are always converted to Standard decimal notation. That is to say, numbers are never converted to scientifc notation.

The way toString is controlled: If passing a decimal.Decimal typed value, the precision is recorded in the number itself. So, D('4.00') will be expressed as '4.00' and not '4'. D('4') will be expressed as '4'.

When a number passed is NOT a Decimal, numbers are created in the following way: Two extra parameters, set during creation of the locale, determines how many digits will appear in the result of toString(). For example, we have a number like 5.1 and mandatory decimals was set to 2, toString(5.1) should return '5.10'. A number like 6 would be '6.00'. A number like 5.104 would depend on the maximum decimals setting, also set at construction of the locale: _maximum_decimals controls the maximum number of decimals after the decimal point So, if _maximum_decimals is 6 and _mandatory_decimals is 2 then toString(Decimal('3.1415929')) is '3.141,592'. Notice the number is truncated and not rounded. Consider rounding a copy of the number before displaying.

toUInt (*s*, *base=0*)
This creates a numeric representation of s.

It returns an ordered pair. The first of the pair is the number, the second of the pair indicates whether the string had a valid representation of that number. You should always check the second of the ordered pair before using the number returned.

note Make sure you use 10 as the second argument or it may interpret the string as octal!

If base is not set, numbers such as '0777' will be interpreted as octal. The string '0x33' will be interpreted as hexadecimal and '777' will be interpreted as a decimal. It is done this way so this works like toLong, toInt, toFloat, etc...

Leading and trailing whitespace is ignored.

toULongLong(*s*, *base=0*)

This creates a numeric representation of s.

It returns an ordered pair. The first of the pair is the number, the second of the pair indicates whether the string had a valid representation of that number. You should always check the second of the ordered pair before using the number returned.

note Make sure you use 10 as the second argument or it may interpret the string as octal!

If base is not set, numbers such as '0777' will be interpreted as octal. The string '0x33' will be interpreted as hexadecimal and '777' will be interpreted as a decimal. It is done this way so this works like toLong, toInt, toFloat, etc...

Leading and trailing whitespace is ignored.

toUShort(*s*, *base=0*)

This creates a numeric representation of s.

It returns an ordered pair. The first of the pair is the number, the second of the pair indicates whether the string had a valid representation of that number. You should always check the second of the ordered pair before using the number returned.

note Make sure you use 10 as the second argument or it may interpret the string as octal!

If base is not set, numbers such as '0777' will be interpreted as octal. The string '0x33' will be interpreted as hexadecimal and '777' will be interpreted as a decimal. It is done this way so this works like toLong, toInt, toFloat, etc...

Leading and trailing whitespace is ignored.

class qsdn.**QSDNNumericValidator**(*maximum_decamals=1000*, *maximum_decimals=1000*, *use_space=False*, *parent=None*)

QSDNNumericValidator limits the number of digits after the decimal point and the number of digits before.

bitcoin : QSDNNumericValidator(8, 8) US dollars less than $1,000,000 : QSDNNumericValidator(6, 2)

If use space is true, spaces are added on the left such that the location of decimal point remains constant. Numbers like '10,000.004', '102.126' become aligned. Bitcoin amounts:

' 0.004,3' ' 10.4' ' 320.0' ' 0.000,004'

U.S. dollar amounts; dollar = QSDNNumericValidator(6,2) s = '42.1' dollar.validate(s='42.1', 2) => s = ' 42.10' s='50000' dollar.toString(s) => s = ' 50,000.00'

decamals()

gets the number of decimal points that are allowed **before** the decimal point

decimals()

gets the number of decimal points that are allowed *after* the decimal point

locale()
get the locale used by this validator

setDecamals(*i*)
sets the number of decimal digits that should be allowed **before** the decimal point

setDecimals(*i*)
sets the number of decimal digits that should be allowed **after** the decimal point

setLocale(*plocale*)
Set the locale used by this Validator.

validate(*s*, *pos*)
Validates s, by adjusting the position of the commas to be in the correct places and adjusting pos accordingly as well as space in order to keep decimal points aligned when varying sized numbers are put one above the other.

CHAPTER

ONE

INDICES AND TABLES

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

PYTHON MODULE INDEX

q

INDEX