inkscapeMadeEasy Documentation

Release 0.9x

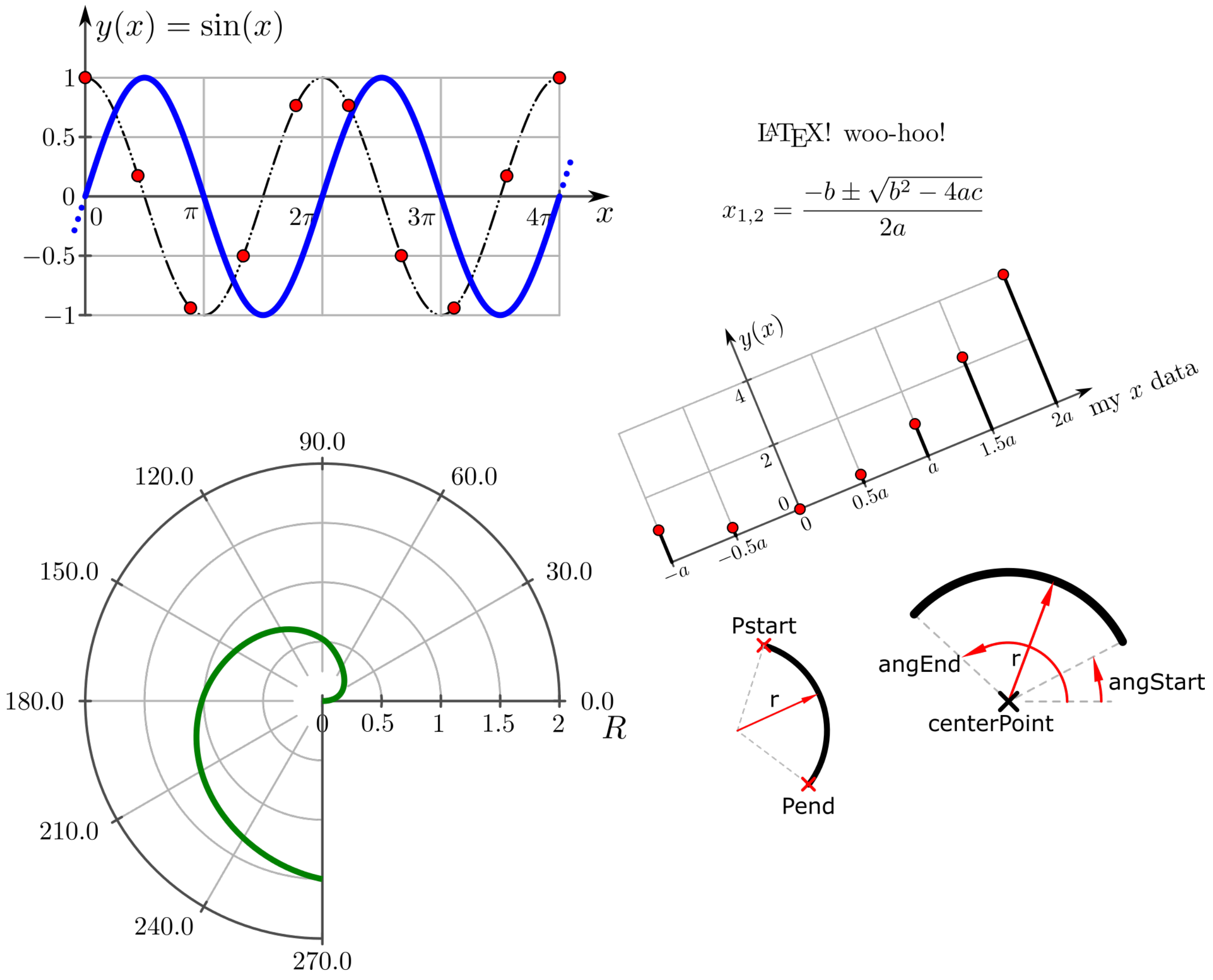
fsmMLK

Sep 14, 2020

CONTENTS

This set of python modules is intended to extend Aaron Spike's inkex.py module, adding functions to help the development of new extensions for Inkscape <https://inkscape.org>.

Here you will find methods and classes to deal with drawings, styles, markers, texts, plots, etc. It is a work-in-progress project and new features will be added in the future. However there is no roadmap right now.



This project is not intended to provide an end-user Inkscape extension by itself but to provide easier backstage functions and classes to facilitate the development of Inkscape extensions.

For end-user extensions see my other projects on GitHub (more to come soon):

- **createMarkers** <https://github.com/fsmMLK/inkscapeCreateMarkers>
- **cartesianAxes2D** <https://github.com/fsmMLK/inkscapeCartesianAxes2D>
- **cartesianPlotFunction2D** <https://github.com/fsmMLK/inkscapeCartesianPlotFunction2D>
- **cartesianPlotData2D** <https://github.com/fsmMLK/inkscapeCartesianPlotData2D>
- **cartesianStemPlot** <https://github.com/fsmMLK/inkscapeCartesianStemPlot>
- **polarAxes2D** <https://github.com/fsmMLK/inkscapePolarAxes2D>
- **logicGates** <https://github.com/fsmMLK/inkscapeLogicGates>
- **circuitSymbols** <https://github.com/fsmMLK/inkscapeCircuitSymbols>
- **dimensions** <https://github.com/fsmMLK/inkscapeDimensions>

- **SlopeField** <https://github.com/fsmMLK/inkscapeSlopeField>

CHAPTER

ONE

HISTORY AND OBJECTIVES

Historically this project started as a way to help myself while creating extensions, namely focusing on scientific/academic diagrams and graphs. In the academy, it is very common to prepare plots/diagrams to explain concepts during lectures, seminars or congresses.

There are many consecrated mathematical tools that can produce them, e.g., gnuplot, octave, matlab, R, etc. They all can produce nice plots, however it might be a little complicated if we want to add other elements to these plots, like texts, comments, arrows, etc. These packages have tools to do it but they are cumbersome to use. A better approach would be using a proper graphic software.

One possible approach is to export these plots as raster images and use a raster graphic software to produce the annotations, like Gimp. Personally, I prefer to have the plots in a vector graphic format to keep it aesthetically pleasing and add the annotations in a vector graphic software. For this, Inkscape is very sound.

Unfortunately, exporting the plots as vector graphics is not always successful in the sense that the resulting document is quite "dirty" (unorganized groups, isolated elements, etc.). Therefore I decided to make my own plotting/diagram tools for Inkscape.

In the process of creating these tools (I will upload them to GitHub in a near future) I realized that many of the low level classes and methods used to manipulate elements of the svg file could be grouped in a general purpose set of core modules that extended inkex.py module. **inkscapeMadeEasy** was born! The core modules I created do not intend to provide an extensive array of methods and classes to cover all possibilities of manipulations/transformations and drawing. These modules were born and expanded as I felt the necessity to have new methods to help my workflow. Nevertheless the number of methods created allows many possibilities and is still under development so new features can (will) appear in future versions.

Obs: Since it is not very easy to find documentation on other Inkscape modules, there might be other modules with similar/better features that I was not aware of when I was producing my extensions.

Enough mumbo-jumbo. Let's start! =D

CHAPTER TWO

CONTENTS:

CHAPTER

THREE

MAIN FEATURES

3.1 Optional LaTeX support via textext extension

Many of the functions implemented in this project can use LaTeX to generate text if the support is enabled. To this end I decided to employ the excellent extension **textext** by Pauli Virtanen <https://pav.iki.fi/software/textext/>.

Since I made very few modifications to his module, I decided to include the modified files here. The modifications were merely designed to facilitate debugging and to remove additional GUI modules (GTK or TkInter). If you prefer, you can stick with the original version.

Please keep in mind that you will need to install 'pstoedit' converter. Linux users can install from your preferred package manager. Windows users can download it from its website.

3.2 Scientific plotting system

inkscapeMadeEasy_Plot module provides simple yet powerful tools to generate 2D Cartesian and Polar axes and 2D plots (lines and Octave's stem plot style). It was inspired by Tavmjong Bah's (and collaborators) extension **Function Plotter** already presented in Inkscape. Function Plotter extension is not required here.

3.3 Control over text styles, line markers and other drawing features

inkscapeMadeEasy_Draw module provides powerful tools to:

- manipulate colors, including pre-defined named colors, gray scale, RGB conversion and the color picker widget of .inx
- create custom markers, including a few pre-defined often used types. All can be assigned to custom colors in both stroke and filling colors
- create line styles, with or without custom markers and custom line dash pattern
- create text styles
- create LaTeX contents (thanks to **textext** by Pauli Virtanen)
- direct control over text size and color of LaTeX contents
- draw straight polylines using absolute or relative coordinates
- draw arcs given the start and end points and its radius
- draw arcs given the center point and start and end angles
- draw circles/ellipses given center and radius

3.4 Useful backstage functions

inkscapeMadeEasy_Base module inherits inkex.py module, extending it by providing useful core functions:

- Dump objects to a text file. Rationale: As of today, 2016, Inkscape does not return the stdout() during python code run. This method partially overcomes this issue by sending the object to a text file.
- unique ID number generator (adapted from inkex.py)
- list defined elements (e.g. markers) in the current document
- create groups
- get the resulting transformation matrix of an object, even when multiple transformations are stacked
- rotate, scale and move objects
- find markers
- get the list of point coordinates of an object or group. If the object is a group the method searches points recursively. Any eventual transformations are properly applied to the points before listing them
- get the bounding box of an object or group
- get the center of the bounding box of an object or group

CHAPTER

FOUR

INSTALLATION AND REQUIREMENTS (ALL USERS)

These modules were partially developed in Inkscape 0.48, 0.91, 0.92 in Linux (Kubuntu 12.04 and 14.04). They should work on both versions of Inkscape. Also, they should work in different OSs too as long as all requirements are met.

The following python modules are required: inkex (comes with inkscape), re, lxml, numpy, math, simplestyle (comes with inkscape), copy, tempfile, sys and os.

Please keep in mind that you will need to install 'pstoedit' converter if you want to use LaTeX support. Linux users can install from your preferred package manager. Windows users can download it from its website.

Some combinations of pstoedit and ghostscript versions fails to produce svg on most distributions

Problematic combinations:

- pstoedit 3.70 + ghostscript 9.22
- pstoedit <= 3.74 + ghostscript 9.27**

In order to install inkscapeMadeEasy, you must download inkscapeMadeEasy files from github and place them inside Inkscape's extension directory (see below how to find it). In the end you must have the following files and directories in your Inkscape extension directory.

```
>>> inkscape/extensions/
>>>             |-- inkscapeMadeEasy_Base.py
>>>             |-- inkscapeMadeEasy_Draw.py
>>>             |-- inkscapeMadeEasy_Plot.py
>>>             `-- textextLib
>>>                 |-- __init__.py
>>>                 |-- basicLatexPackages.tex
>>>                 |-- textext.inx
>>>                 |-- textext.py
```

Inkscape's extension directory Your inkscape extension directory location can be found opening Inkscape and selecting `Edit>Preferences>System` and checking `User Extensions` field. (Windows users: you can easily open this directory using Windows Explorer by copying the text in this field and pasting on Windows Explorer's search bar)

LaTeX package requirement

If LaTeX support is enabled (see below), you will need in your system the following LaTeX packages: amsmath, amsthm, amsbsy, amsfonts, amssymb, siunitx, steinmetz.

Linux users: You might find useful installing the packages `texlive-science`, `texlive-pictures` and `texlive-latex-base` (Debian based distros) from your package manager. They should provide most (all?) needed LaTeX packages.

Windows users: please check `Installation and requirements (windows users)` section

Disabling LaTeX support

LaTeX support via textext extension requires LaTeX typesetting system in your computer (it's free and awesome! =]). It might be a problem to install for non-Linux systems.

Since many people don't use LaTeX and/or don't have it installed (it might be a pain to install it on Windows machines), LaTeX support is now optional. **By default, LaTeX support is ENABLED.**

If you don't want LaTeX support or your system does not allow it, you can still use **inkscapeMadeEasy** as long as you disable it. You can easily do it by setting a flag in `inkscapeMadeEasy_Draw.py`:

> 1- Open `inkscapeMadeEasy_Draw.py` in any text editor (e.g. Notepad in Windows)
>
> 2- Search for the line containing `#useLatex=False`. It should be somewhere at the beginning of the file.
>
> 3- Remove the comment character `#` of this line, leaving just `useLatex=False`.
>
> 4- Save the file.

CHAPTER

FIVE

LATEX INSTALLATION (WINDOWS USERS)

1) Install Miktex:

Download and install Miktex (https://miktex.org/). You must make sure the following minimal testing example compiles correctly using pdflatex using the command prompt. Se instructions below:

```
\documentclass[11pt]{article}
\usepackage[utf8]{inputenc}
\usepackage{amsmath,amsthm,amsbsy,amsfonts,amssymb}
\usepackage[per=slash]{siunitx}
\usepackage{steinmetz}
\begin{document}
Minimal example. woo-hoo!
\begin{align}
E=mc^2
\end{align}
\end{document}
```

1.2) testing pdflatex (for LaTeX experienced users)

Check whether you can call pdflatex from any folder, in other words, check if pdflatex folder is in the PATH environment variable.

Check whether your pdflatex can compile the testing example from the command prompt. Compiling this example will also make sure you have all packages inkscapeMadeEasy requires.

1.3) testing pdflatex (for LaTeX beginners)

1.3.1) Open notepad and create a text file with the contents of the testing example and save it somewhere.

1.3.2) In File Explorer, go to the folder where you saved the file and click the address bar to select it (or press Alt+D). Type "cmd" into the address bar and hit Enter to open the Command Prompt with the path of the current folder already set.

1.3.3) type: pdflatex <name_of_the_file_you_just_saved>. Lots of stuff should appear on your console window. Obs: Miktex might require to install additional packages. Depending on how you installed Miktex, it can install automatically the needed packages or ask you to confirm. Confirm it!

1.3.4) Check whether pdflatex created a new pdf file with the same name. Open the pdf and see if you can read the short message and equation.

2) install pstoedit (version 3.73)

windows 32 bits https://sourceforge.net/projects/pstoedit/files/pstoedit/3.73/pstoeditsetup_win32.exe

windows 64 bits https://sourceforge.net/projects/pstoedit/files/pstoedit/3.73/pstoeditsetup_x64.exe

2.1) Add pstoedit folder to windows PATH environment variable (if it is not there already)

2.2.1) On the Windows desktop, right-click 'My Computer'.

2.2.2-) In the pop-up menu, click Properties.

2.2.3-) In the System Properties window, click the Advanced tab, and then click Environment Variables.

2.2.4-) In the System Variables window, highlight PATH, and click Edit.

2.2.5-) Create a new entry and provide the path where pstoedit was installed. On my machine the folder was 'C:\Program Files\pstoedit'

3) Install ghostscript (version 9.26)

windows 32 bits https://github.com/ArtifexSoftware/ghostpdl-downloads/releases/download/gs926/gs926aw32.exe

windows 64 bits https://github.com/ArtifexSoftware/ghostpdl-downloads/releases/download/gs926/gs926aw64.exe

CHAPTER

SIX

USAGE

These modules are not intended to serve as extensions by themselves. Instead you can import them into your projects to take advantage of the classes and methods developed here.

For examples on how to use, please take a look at the examples provided below and also check my other extension projects on GitHub (more to come soon):

- **createMarkers** <https://github.com/fsmMLK/inkscapeCreateMarkers>
- **cartesianAxes2D** <https://github.com/fsmMLK/inkscapeCartesianAxes2D>
- **cartesianPlotFunction2D** <https://github.com/fsmMLK/inkscapeCartesianPlotFunction2D>
- **cartesianPlotData2D** <https://github.com/fsmMLK/inkscapeCartesianPlotData2D>
- **polarAxes2D** <https://github.com/fsmMLK/inkscapePolarAxes2D>
- **logicGates** <https://github.com/fsmMLK/inkscapeLogicGates>
- **circuitSymbols** <https://github.com/fsmMLK/inkscapeCircuitSymbols>
- **dimensions** <https://github.com/fsmMLK/inkscapeDimensions>

CHAPTER

SEVEN

MODULE DEFINITIONS

7.1 inkscapeMadeEasy_Base

class inkscapeMadeEasy_Base.**inkscapeMadeEasy**

Bases: `inkex.deprecated.Effect`

Base class for extensions.

This class extends the inkex.Effect class by adding several basic functions to help manipulating inkscape elements. All extensions should inherit this class.

displayMsg(*msg*)

Displays a message to the user.

Parameters **msg** (*string*) – message

Returns nothing

Return type

-

getBasicLatexPackagesFile()

Returns the full path of the `basicLatexPackages.tex` file with commonly used Latex packages

The default packages are:

```
\usepackage{amsmath,amsthm,amsbsy,amsfonts,amssymb}
\usepackage[per=slash]{siunitx}
\usepackage{steinmetz}
```

You can add other packages to the file `basicLatexPackages.tex`, located in the extension directory.

Returns Full path of the file with commonly used Latex packages

Return type string

Dump(*obj*, *file='./dump_file.txt'*, *mode='w'*)

Function to easily output the result of `str(obj)` to a file

This function was created to help debugging the code while it is running under inkscape. Since inkscape does not possess a terminal as today (2016), this function overcomes partially the issue of sending things to stdout by dumping result of the function `str()` in a text file.

Parameters

- **obj** (any, as long as `str(obj)` is implemented (see `__str__()` metaclass definition)) – object to sent to a file. Any type that can be used in `str()`
- **file** (*string*) – file path. Default: `./dump_file.txt`

- **mode** (*string*) – writing mode of the file Default: `w` (write)

Returns nothing

Return type

-

Example

```
>>> from inkscapeMadeEasy_Base import inkscapeMadeEasy
>>> x=inkscapeMadeEasy
>>> vector1=[1,2,3,4,5,6]
>>> x.Dump(vector1,file='~/temporary.txt',mode='w')   # writes the list to a␣
↪file
>>> vector2=[7,8,9,10]
>>> x.Dump(vector2,file='~/temporary.txt',mode='a')   # append the list to a␣
↪file
```

removeElement (*element*)
Function to remove one element or group. If the parent of the element is a group and has no other children, then the parent is also removed.

Parameters **element** (*element object*) – element object

Returns nothing

Return type

-

Example

```
>>> rootLayer = self.document.getroot()                              #␣
↪retrieves the root layer of the file
>>> groupA = self.createGroup(rootLayer,label='temp')                #␣
↪creates a group inside rootLayer
>>> line1 = inkscapeMadeEasy_Draw.line.relCoords(groupA, [[5,0]],[0,0])
↪# creates a line in groupA
>>> line2 = inkscapeMadeEasy_Draw.line.relCoords(rootLayer, [[5,0]],[0,0])
↪# creates a line in rootLayer
>>> line3 = inkscapeMadeEasy_Draw.line.relCoords(groupA, [[15,0]],[10,0])
↪# creates a line in groupA
>>> self.removeElement(line1)                              # removes line 1
>>> self.removeElement(line2)                              # removes line 2
>>> self.removeElement(line3)                              # removes line 3.␣
↪Also removes groupA since this group has no other children
>>> groupB = self.createGroup(rootLayer,label='temp1')                #␣
↪creates a group inside rootLayer
>>> line4 = inkscapeMadeEasy_Draw.line.relCoords(groupB, [[5,0]],[0,0])
↪# creates a line in groupB
>>> self.removeElement(groupB)                              # removes group B␣
↪and all its children
```

exportSVG (*element, fileOut*)
Export the elements in a new svgfile

This function will export the element in a new SVG file. If a list of elements is passed as argument. All elements in the list will be exported to the same file.

Parameters

- **element** (*element or list of elements*) – element or list of elements to be exported
- **fileOut** – file path, including the extension.

Returns nothing

Return type

-

Note: All the defs of the original file will be copied to the new file. Therefore you might want to run te vacuum tool to cleanup the new SVG file

Example

```
>>> from inkscapeMadeEasy_Base import inkscapeMadeEasy
>>> import inkscapeMadeEasy_Draw as inkDraw
>>> x=inkscapeMadeEasy
>>> rootLayer = x.document.getroot()                              # retrieves␣
↪the root layer of the file
>>> groupA = x.createGroup(rootLayer,label='temp')                # creates a␣
↪group inside rootLayer
>>> groupB = x.createGroup(rootLayer,label='child')               # creates a␣
↪group inside groupA
>>> line1 = inkDraw.line.relCoords(groupA, [[10,0]],[0,0])        # creates a␣
↪line in groupA
>>> line2 = inkDraw.line.relCoords(groupB, [[20,0]],[0,0])        # creates a␣
↪line in groupB
>>> self.exportSVG(line1,'file1.svg')                             # exports␣
↪only line1
>>> self.exportSVG(groupA,'file2.svg')                            # exports␣
↪groupA (and all elements contained in it)
>>> self.exportSVG([groupA,groupB],'file3.svg')                   # exports␣
↪groupA and groupB (and all elements contained in it) to the same file
```

uniqueIdNumber (*prefix_id*)

Generates a unique ID with a given prefix ID by adding a numeric suffix

This function is used to generate a valid unique ID by concatenating a given prefix with a numeric suffix. The overall format is `prefix-%05d`.

This function makes sure the ID is unique by checking in `doc_ids` member. This function is specially useful for creating an unique ID for markers and other elements in defs.

Parameters prefix_id (*string*) – prefix of the ID

Returns the unique ID

Return type string

Note: This function has been adapted from Aaron Spike's inkex.py https://github.com/hacktoon/ink2canvas/blob/master/ink2canvas/lib/inkex.py

Example

```
>>> from inkscapeMadeEasy_Base import inkscapeMadeEasy
>>> x=inkscapeMadeEasy
```

(continues on next page)

(continued from previous page)

```
>>> a=x.uniqueIdNumber('myName')    # a=myName-00001
>>> b=x.uniqueIdNumber('myName')    # b=myName-00002
>>> c=x.uniqueIdNumber('myName')    # c=myName-00003
```

getDefinitions()

retrieves the Defs element of the svg file.

This function returns the element Defs of the current svg file. This elements stores the definition (e.g. marker definition)

if no Defs can be found, a new empty Defs is created

Returns the defs element

Return type etree element

unifyDefs()

Unify all <defs> nodes in a single <defs> node.

Returns None

Return type

-

Note: This function does not check whether the ids are unique

getDefsByTag(*tag='marker'*)

retrieves the Defs elements of the svg file of a given a tag.

Returns a list with the def elements

Return type list of etree element

getDefsById(*id*)

return a list of elements in the defs node, given (part of) the id

Returns a list with the def elements

Return type list of etree element

getElemFromXpath(*xpath*)

returns the element from the xml, given its xpath

Parameters **xpath** (*string*) – tag of the element to be searched

Returns element

Return type element

Example

```
>>> from inkscapeMadeEasy_Base import inkscapeMadeEasy
>>> x=inkscapeMadeEasy
>>> name = x.getElemFromXpath('/svg:svg//svg:defs')    # returns the list of␣
↪definitions of the document
```

getElemAttrib(*elem, attribName*)

Returns the atribute of one element, given the atribute name

Parameters

- **elem** (*element*) – elem under consideration
- **attribName** (*string*) – attribute to be searched. Format: namespace:attrName

Returns attribute

Return type string

```
>>> from inkscapeMadeEasy_Base import inkscapeMadeEasy
>>> x=inkscapeMadeEasy
>>> elem= x.getElemFromXpath('/svg:svg')
>>> docNAme = x.getElemAttrib(elem,'sodipodi:docname')
```

getDocumentScale()

returns the scale of the document

Example

```
>>> scale = x.getDocumentScale()
```

getDocumentName()

returns the name of the document

Returns fileName

Return type string

Example

```
>>> from inkscapeMadeEasy_Base import inkscapeMadeEasy
>>> x=inkscapeMadeEasy
>>> name = x.getDocumentName()
```

getDocumentUnit()

returns the unit of the document

Returns unit string code. See table below

Return type string

Units

The list of available units are:

Name	string code	relation
millimetre	mm	1in = 25.4mm
centimetre	cm	1cm = 10mm
metre	m	1m = 100cm
kilometre	km	1km = 1000m
inch	in	1in = 96px
foot	ft	1ft = 12in
yard	yd	1yd = 3ft
point	pt	1in = 72pt
pixel	px	
pica	pc	1in = 6pc

Example

```
>>> docunit = self.getDocumentUnit()    #returns 'cm', 'mm', etc.
```

getcurrentLayer()

returns the current layer of the document

Returns name of the current layer

Return type string

Example

```
>>> from inkscapeMadeEasy_Base import inkscapeMadeEasy
>>> x=inkscapeMadeEasy
>>> name = x.getDocumentName()
```

removeAbsPath(*element*)

unit2userUnit(*value*, *unit_in*)

Converts a value from given unit to inkscape's default unit (px)

Parameters

- **value** (*float*) – value to be converted
- **unit_in** (*string*) – input unit string code. See table below

Returns converted value

Return type float

Units

The list of available units are:

Name	string code	relation
millimetre	mm	1in = 25.4mm
centimetre	cm	1cm = 10mm
metre	m	1m = 100cm
kilometre	km	1km = 1000m
inch	in	1in = 96px
foot	ft	1ft = 12in
yard	yd	1yd = 3ft
point	pt	1in = 72pt
pixel	px	
pica	pc	1in = 6pc

Example

```
>>> x_cm = 5.0
>>> x_px = self.unit2userUnit(x_cm,'cm')        # converts  5.0cm -> 188.97px
```

userUnit2unit(*value*, *unit_out*)

Converts a value from inkscape's default unit (px) to specified unit

Parameters

- **value** (*float*) – value to be converted
- **unit_out** (*string*) – output unit string code. See table below

Returns converted value

Return type float

Units

The list of available units are:

Name	string code	relation
millimetre	mm	1in = 25.4mm
centimetre	cm	1cm = 10mm
metre	m	1m = 100cm
kilometre	km	1km = 1000m
inch	in	1in = 96px
foot	ft	1ft = 12in
yard	yd	1yd = 3ft
point	pt	1in = 72pt
pixel	px	
pica	pc	1in = 6pc

Example

```
>>> x_px = 5.0
>>> x_cm = self.userUnit2unit(x_px,'cm')       # converts  5.0px -> 0.1322cm
```

unit2unit (*value*, *unit_in*, *unit_out*)

Converts a value from one provided unit to another unit

Parameters

- **value** (*float*) – value to be converted
- **unit_in** (*string*) – input unit string code. See table below
- **unit_out** (*string*) – output unit string code. See table below

Returns converted value

Return type float

Units

The list of available units are:

Name	string code	relation
millimetre	mm	1in = 25.4mm
centimetre	cm	1cm = 10mm
metre	m	1m = 100cm
kilometre	km	1km = 1000m
inch	in	1in = 96px
foot	ft	1ft = 12in
yard	yd	1yd = 3ft
point	pt	1in = 72pt
pixel	px	
pica	pc	1in = 6pc

Example

```
>>> x_in = 5.0
>>> x_cm = self.unit2unit(x_in,'in','cm')       # converts  5.0in -> 12.7cm
```

createGroup (*parent*, *label='none'*)
Creates a new empty group of elements.

This function creates a new empty group of elements. In order to create new elements inside this groups you must create them informing the group as the parent element.

Parameters

- **parent** (*element object*) – parent object of the group. It can be another group or the root element
- **label** (*string*) – label of the group. Default: 'none'

Returns the group object

Return type group element

Note: The label does not have to be unique

Example

```
>>> rootLayer = self.document.getroot()                              #␣
↪retrieves the root layer of the file
>>> groupA = self.createGroup(rootLayer,label='temp')                #␣
↪creates a group inside rootLayer
>>> groupB = self.createGroup(groupA,label='child')                  #␣
↪creates a group inside groupA
>>> line1 = inkscapeMadeEasy_Draw.line.relCoords(groupA, [[10,0]],[0,0])      ␣
↪ # creates a line in groupA
>>> line2 = inkscapeMadeEasy_Draw.line.relCoords(groupB, [[20,0]],[0,0])      ␣
↪ # creates a line in groupB
```

ungroup (*group*)
Ungroup elements

The new parent element of the ungrouped elements will be the parent of the removed group. See example below

Parameters **group** (*group element*) – group to be removed

Returns list of the elements previously contained in the group

Return type list of elements

Example

```
>>> rootLayer = self.document.getroot()                              #␣
↪retrieves the root layer of the file
>>> groupA = self.createGroup(rootLayer,label='temp')                #␣
↪creates a group inside rootLayer
>>> groupB = self.createGroup(groupA,label='temp')                # creates a␣
↪group inside groupA
>>> line1 = inkscapeMadeEasy_Draw.line.relCoords(groupA, [[10,0]],[0,0])      ␣
↪ # creates a line in groupA
>>> line2 = inkscapeMadeEasy_Draw.line.relCoords(groupB, [[20,0]],[0,0])      ␣
↪ # creates a line in groupB
>>> line3 = inkscapeMadeEasy_Draw.line.relCoords(groupB, [[30,0]],[0,0])      ␣
↪ # creates a line in groupB
>>>  # at this point, the file struct is:   groupA[ line1, groupB[ line2,␣
↪line3 ] ]
```

(continues on next page)

(continued from previous page)

```
>>> elemList = self.ungroup(groupB)                                          #␣
↪ungroup line2 and line3. elemList is a list containing line2 and line3␣
↪elements.
>>>  # now the file struct is:   groupA[ line1, line2, line3 ]
```

getTransformMatrix (*element*)

Returns the transformation attribute of the given element and the resulting transformation matrix (numpy Array)

This function is used to extract the transformation operator of a given element.

Parameters **element** (`element object`) – element object

Returns

[transfAttrib, transfMatrix]

- transfAttrib: string containing all transformations as it is in the file
- transfMatrix: numpy array with the resulting transformation matrix

Return type tuple

If the element does not have any transformation attribute, this function returns:

- transfAttrib=”
- transfMatrix=identity matrix

rotateElement (*element*, *center*, *angleDeg*)

Rotates the element using the transformation attribute.

It is possible to rotate elements isolated or entire groups.

Parameters

- **element** (`element object`) – element object to be rotated
- **center** (`tuple`) – tuple with center point of rotation
- **angleDeg** (`float`) – angle of rotation in degrees, counter-clockwise direction

Returns nothing

Return type

-

Example

```
>>> rootLayer = self.document.getroot()                                    #␣
↪retrieves the root layer of the file
>>> groupA = self.createGroup(rootLayer,label='temp')                      #␣
↪creates a group inside rootLayer
>>> line1 = inkscapeMadeEasy_Draw.line.relCoords(groupA, [[5,0]],[0,0])
↪# creates a line in groupA
>>> line2 = inkscapeMadeEasy_Draw.line.relCoords(rootLayer, [[5,0]],[0,0])
↪# creates a line in rootLayer
>>> self.rotateElement(line2,[0,0],120)                                    #␣
↪rotates line2 120 degrees around center x=0,y=0
>>> self.rotateElement(groupA,[1,1],-90)                                   #␣
↪rotates groupA -90 degrees around center x=1,y=1
```

copyElement (*element, newParent, distance=None, angleDeg=None*)
Copies one element to the same or other parent Element.

It is possible to copy elements isolated or entire groups.

Parameters

- **element** (*element object*) – element object to be copied
- **newParent** (*element object*) – New parent object. It can be another group or the group
- **distance** (*tuple*) – tuple with the distance to move the object. If None, then the copy is placed at the same position
- **angleDeg** (*float*) – angle of rotation in degrees, counter-clockwise direction

Returns newElement

Return type element object

Example

```
>>> rootLayer = self.document.getroot()                              #␣
↪retrieves the root layer of the file
>>> groupA = self.createGroup(rootLayer,label='temp')                #␣
↪creates a group inside rootLayer
>>> line1 = inkscapeMadeEasy_Draw.line.relCoords(groupA, [[5,0]],[0,0])
↪# creates a line in groupA
>>> line2 = inkscapeMadeEasy_Draw.line.relCoords(rootLayer, [[5,0]],[0,0])
↪# creates a line in rootLayer
>>> self.copyElement(line2,groupA)                                  # create␣
↪a copy of line2 in groupA
>>> self.moveElement(groupA,[10,-10])                                # moves␣
↪line2  DeltaX=10, DdeltaY=-10
```

moveElement (*element, distance*)
Moves the element using the transformation attribute.

It is possible to move elements isolated or entire groups.

Parameters

- **element** (*element object*) – element object to be rotated
- **distance** (*tuple*) – tuple with the distance to move the object

Returns nothing

Return type

-

Example

```
>>> rootLayer = self.document.getroot()                              #␣
↪retrieves the root layer of the file
>>> groupA = self.createGroup(rootLayer,label='temp')                #␣
↪creates a group inside rootLayer
>>> line1 = inkscapeMadeEasy_Draw.line.relCoords(groupA, [[5,0]],[0,0])
↪# creates a line in groupA
>>> line2 = inkscapeMadeEasy_Draw.line.relCoords(rootLayer, [[5,0]],[0,0])
↪# creates a line in rootLayer
```

(continues on next page)

(continued from previous page)

```
>>> self.moveElement(line2,[10,10])                                    # moves␣
↪line2  DeltaX=10, DdeltaY=10
>>> self.moveElement(groupA,[10,-10])                                  # moves␣
↪line2  DeltaX=10, DdeltaY=-10
```

scaleElement (*element*, *scaleX=1.0*, *scaleY=0.0*, *center=None*)
Scales the element using the transformation attribute.

It is possible to scale elements isolated or entire groups.

Parameters

- **element** (*element object*) – element object to be rotated
- **scaleX** (*float*) – scaling factor in X direction. Default=1.0
- **scaleY** – scaling factor in Y direction. Default=0.0
- **center** (*tuple*) – center point considered as the origin for the scaling. Default=None. If None, the origin is adopted

Returns nothing

Return type

-

Note: If scaleY==0, then scaleY=scaleX is assumed (default behavior)

Example

```
>>> rootLayer = self.document.getroot()                                #␣
↪retrieves the root layer of the file
>>> groupA = self.createGroup(rootLayer,label='temp')                  #␣
↪creates a group inside rootLayer
>>> circ1 = centerRadius(groupA,centerPoint=[0,0],radius=1.0)          #␣
↪creates a line in groupA
>>> circ2 = centerRadius(rootLayer,centerPoint=[0,0],radius=1.0)       #␣
↪creates a line in rootLayer
>>> self.scaleElement(circ1,2.0)                                       # scales␣
↪x2 in both X and Y directions
>>> self.scaleElement(circ1,2.0,3.0)                                   # scales␣
↪x2 in X and x3 in Y
>>> self.scaleElement(groupA,0.5)                                      # scales␣
↪x0.5 the group in both X and Y directions
```

findMarker (*markerName*)
Search for markerName in the document.

Parameters **markerName** (*string*) – marker name

Returns True if markerName was found

Return type bool

getPoints (*element*)
Retrieves the list of points of the element.

This function works on paths, texts or groups. In the case of a group, the function will include recursively all its components

Parameters **element** (*element object*) – element object

Returns list of points

Return type list of list

Note: This function will consider any transformation stored in transform attribute, that is, it will compute the resulting coordinates of each object

---

Example

```
>>> rootLayer = self.document.getroot()                              #␣
↪retrieves the root layer of the file
>>> line1 = inkscapeMadeEasy_Draw.line.relCoords(rootLayer, [[5,0],[0,6]],[0,
↪0])    # creates a line in groupA
>>> list = self.getPoints(line1)                                     #␣
↪gets list = [[0.0, 0.0], [5.0, 0.0], [5.0, 6.0]]
```

getBoundingBox(*element*)

Retrieves the bounding Box of the element.

This function works on paths, texts or groups. In the case of a group, the function will consider recursively all its components

Parameters **element** (*element object*) – element object

Returns two lists: [xMin,yMin] and [xMax,yMax]

Return type list

Note: This function will consider any transformation stored in transform attribute, that is, it will compute the resulting coordinates of each object

---

Example

```
>>> rootLayer = self.document.getroot()                              #␣
↪retrieves the root layer of the file
>>> line1 = inkscapeMadeEasy_Draw.line.relCoords(rootLayer, [[5,0],[0,6]],[0,
↪0])    # creates a line in groupA
>>> BboxMin,BboxMax = self.getBoundingBox(line1)                     #␣
↪gets BboxMin = [0.0, 0.0] and BboxMax = [5.0, 6.0]
```

getCenter(*element*)

Retrieves the center coordinates of the bounding Box of the element.

This function works on paths, texts or groups. In the case of a group, the function will consider recursively all its components

Parameters **element** (*element object*) – element object

Returns two lists: [xCenter,yCenter]

Return type list

Note: This function will consider any transformation stored in transform attribute, that is, it will compute the resulting coordinates of each object

---

Example

```
>>> rootLayer = self.document.getroot()                                  #␣
↪retrieves the root layer of the file
>>> line1 = inkscapeMadeEasy_Draw.line.relCoords(rootLayer, [[5,0],[0,6]],[0,
↪0])    # creates a line in groupA
>>> Center = self.getCenter(line1)                                       #␣
↪gets Center = [2.5, 3.0]
```

getSegmentFromPoints (*pointList*, *normalDirection='R'*)

given two points of a straight line segment, returns the parameters of that segment:

length, angle (in radians), tangent unitary vector and normal unitary vector

Parameters

- **pointList** (*list of points*) – start and end coordinates [Pstart, Pend]
- **normalDirection** (*string*) –
  - 'R': normal vector points to the right of the tangent vector (Default)
  - 'L': normal vector points to the left of the tangent vector (Default)

Returns list: [length, theta, t_versor,n_versor]

Return type list

Example

```
>>> segmentParam = getSegmentFromPoints([[1,1],[2,2]],'R')                ␣
↪           # returns [1.4142, 0.78540, [0.7071,0.7071], [0.7071,-0.7071] ]
>>> segmentParam = getSegmentFromPoints([[1,1],[2,2]],'L')                ␣
↪           # returns [1.4142, 0.78540, [0.7071,0.7071], [-0.7071,0.7071] ]
```

getSegmentParameters (*element*, *normalDirection='R'*)

given a path segment composed by only two points, returns the parameters of that segment:

length, angle (in radians), start point, end point, tangent unitary vector and normal unitary vector

This function works with paths only.

Parameters

- **element** (*element object*) – path element object
- **normalDirection** (*string*) –
  - 'R': normal vector points to the right of the tangent vector (Default)
  - 'L': normal vector points to the left of the tangent vector (Default)

Returns list: [Pstart,Pend,length, theta, t_versor,n_versor]

Return type list

If the element is not a path, the function returns an empty list If the path element has more than two points, the function returns an empty list

Note: This function will consider any transformation stored in transform attribute, that is, it will compute the resulting coordinates of each object

Example

```
>>> rootLayer = self.document.getroot()
↪# retrieves the root layer of the file
>>> line1 = inkscapeMadeEasy_Draw.line.absCoords(rootLayer, [[1,1],[2,2]])
↪# creates a line in groupA
>>> segementList = getSegmentParameters(line1,'R')                          ␣
↪   # returns [[1,1], [2,2],1.4142, 0.78540,  [0.7071,0.7071], [0.7071,-0.
↪7071] ]
```

7.2 inkscapeMadeEasy_Draw

`inkscapeMadeEasy_Draw.`**`displayMsg`**`(`*msg*`)`

Displays a message to the user.

Returns nothing

Return type

-

Note: Identical function has been also defined inside inkscapeMadeEasy class

`inkscapeMadeEasy_Draw.`**`Dump`**`(`*obj, file='./dump_file.txt', mode='w'*`)`

Function to easily output the result of `str(obj)` to a file

This function was created to help debugging the code while it is running under inkscape. Since inkscape does not possess a terminal as today (2016), this function overcomes partially the issue of sending things to stdout by dumping result of the function `str()` in a text file.

Parameters

- **`obj`** (any, as long as `str(obj)` is implemented (see `__str__()` metaclass definition)) – object to sent to the file. Any type that can be used in `str()`
- **`file`** (*`string`*) – file path. Default: `./dump_file.txt`
- **`mode`** (*`string`*) – writing mode of the file. Default: `w` (write)

Returns nothing

Return type

-

Note: Identical function has been also defined inside inkscapeMadeEasy class

Example

```
>>> vector1=[1,2,3,4,5,6]
>>> Dump(vector1,file='~/temporary.txt',mode='w')   # writes the list to a file
>>> vector2=[7,8,9,10]
>>> Dump(vector2,file='~/temporary.txt',mode='a')   # append the list to a file
```

`class` `inkscapeMadeEasy_Draw.`**`color`**

Bases: `object`

Class to manipulate colors.

This class manipulates color information, generating a string in inkscape's expected format `#RRGGBB`

This class contains only static methods so that you don't have to inherit this in your class

Note: alpha channel is not implemented yet. Assume alpha=1.0

static defined(*colorName*)

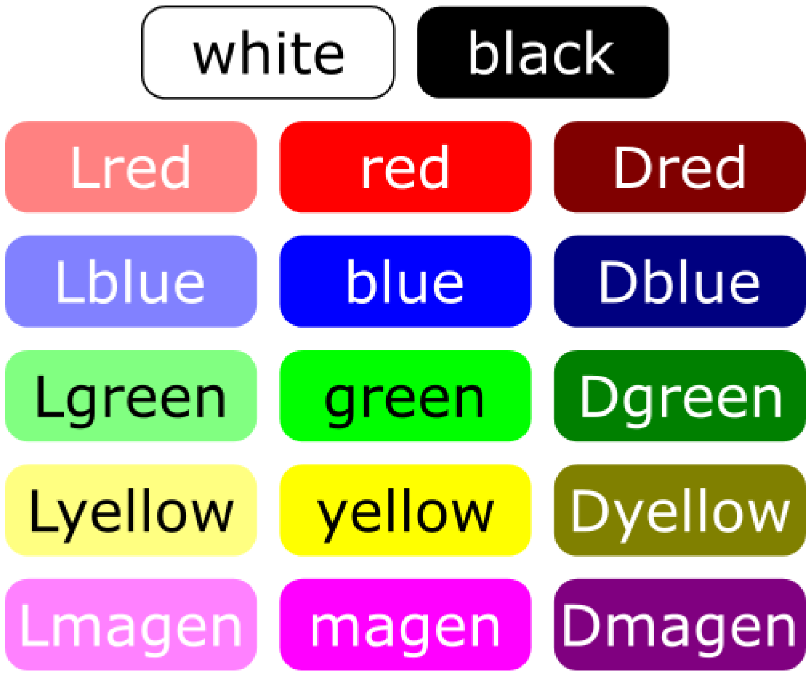
Returns the color string representing a predefined color name

Parameters **colorName** (*string*) – color name

Returns string representing the color in inkscape's expected format `#RRGGBB`

Return type string

Available pre defined colors



Example

```
>>> colorString = color.defined('red')                    # returns #ff0000␣
↪representing red color
```

static RGB(*RGBlist*)

returns a string representing a color specified by RGB level in the range 0-255

Parameters **RGBlist** (*list*) – list containing RGB levels in the range 0-225 each

Returns string representing the color in inkscape's expected format `#RRGGBB`

Return type string

Example

```
>>> colorString = color.RGB([120,80,0])                    # returns a string␣
↪representing the color R=120, G=80, B=0
```

static gray(*percentage*)

returns a gray level compatible string based on white percentage between 0.0 and 1.0

if percentage is higher than 1.0, percentage is truncated to 1.0 (white) if percentage is lower than 0.0, percentage is truncated to 0.0 (black)

Parameters **percentage** (*float*) – value between 0.0 (black) and 1.0 (white)

Returns string representing the color in inkscape's expected format `#RRGGBB`

Return type string

Example

```
>>> colorString = color.gray(0.6)            # returns a string␣
↪representing the gray level with 60% of white
```

static colorPickerToRGBalpha(*colorPickerString*)

Function that converts the string returned by the widget 'color' in the .inx file into 2 strings, one representing the color in format `#RRGGBB` and the other representing the alpha channel `AA`

Parameters **colorPickerString** (*string*) – string returned by 'color' widget

Returns a list of strings: [color,alpha] - color: string in `#RRGGBB` format - alpha: string in `AA` format

Return type list

Note: For more information on this widget, see <http://wiki.inkscape.org/wiki/index.php/INX_Parameters>

Warning: you probably don't need to use this function. Consider using the method `color.parseColorPicker()`

usage

1- in your inx file you must have one attribute of the type 'color':

```
<param name="myColorPicker" type="color"></param>
```

2- in your .py file, you must parse it as a string:

```
>>>   self.OptionParser.add_option("--myColorPicker", action="store",␣
↪type="string", dest="myColorPickerVar", default='0')
```

3- call this function to convert so.myColorPickerVar to two strings

- #RRGGBB with RGB values in hex
- AA with alpha value in hex

Example

Let your .inx file contains a widget of type 'color' with the name myColorPicker:

```
<param name="myColorPicker" type="color"></param>
```

Then in the .py file

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     inkex.Effect.__init__(self)
>>>     self.OptionParser.add_option("--myColorPicker", action="store", type=
↪"string", dest="myColorPickerVar", default='#000000')  # parses the input␣
↪parameter
>>>
>>>   def effect(self):
>>>     color,alpha = inkDraw.color.colorPickerToRGBalpha(self.options.
↪myColorPickerVar)       # returns the string representing the selected␣
↪color and alpha channel
```

static parseColorPicker(*stringColorOption*, *stringColorPicker*)

Function that converts the string returned by the widgets 'color' and 'optiongroup' in the .inx file into 2 strings, one representing the color in format `#RRGGBB` and the other representing the alpha channel `AA`

You must have in your .inx both 'optiongroup' and 'color' widgets as defined below. You don't have to have all the color options presented in the example. That is the most complete example, considering the default colors in color.defined method.

Parameters

- **stringColorOption** (*string*) – string returned by 'optiongroup' widget
- **stringColorPicker** (*string*) – string returned by 'color' widget

Returns a list of strings: [color,alpha] - color: string in `#RRGGBB` format - alpha: string in `AA` format

Return type list

Note: For more information on this widget, see <http://wiki.inkscape.org/wiki/index.php/INX_Parameters>

Example

It works in the following manner: The user select in the optiongroup list the desired color. All pre defined colors are listed there. There is also a 'my default color' where you can set your preferred default color and a 'use color picker' to select from the color picker widget. Keep in mind that the selected color in this widget will be considered ONLY if 'use color picker' option is selected.

Let your .inx file contains a widget of type 'color' with the name 'myColorPicker' and another 'optiongroup' with the name 'myColorOption':

```
<param name="myColorOption" type="optiongroup" appearance="minimal" _gui-text=
↪"some text here">
    <_option value="#FF0022">my default color</_option>    <--you can set␣
↪your pre define color in the form #RRGGBB
    <_option value="none">none</_option>                    <-- no color
    <_option value="black">black</_option>
    <_option value="red">red</_option>
    <_option value="blue">blue</_option>
    <_option value="yellow">yellow</_option>
    <_option value="green">green</_option>          <-- these are all␣
↪standardized colors in inkscapeMadeEasy_Draw.color class!
```

(continues on next page)

(continued from previous page)

```
    <_option value="magen">magenta</_option>
    <_option value="white">white</_option>
    <_option value="Lred">Lred</_option>
    <_option value="Lblue">Lblue</_option>
    <_option value="Lyellow">Lyellow</_option>
    <_option value="Lgreen">Lgreen</_option>
    <_option value="Lmagen">Lmagenta</_option>
    <_option value="Dred">Dred</_option>
    <_option value="Dblue">Dblue</_option>
    <_option value="Dyellow">Dyellow</_option>
    <_option value="Dgreen">Dgreen</_option>
    <_option value="Dmagen">Dmagenta</_option>
    <_option value="picker">use color picker</_option>      <-- indicate that␣
↪the color must be taken from the colorPicker attribute
</param>
<param name="myColorPicker" type="color"></param>
```

Then in the .py file

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     inkex.Effect.__init__(self)
>>>     self.OptionParser.add_option("--myColorPicker", action="store", type=
↪"string", dest="myColorPickerVar", default='0')       # parses the input␣
↪parameters
>>>     self.OptionParser.add_option("--myColorOption", action="store", type=
↪"string", dest="myColorOptionVar", default='black')   # parses the input␣
↪parameter
>>>
>>>   def effect(self):
>>>     so = self.options
>>>     [RGBstring,alpha] = inkDraw.color.parseColorPicker(so.
↪myColorOptionVar,so.myColorPickerVar)
```

class inkscapeMadeEasy_Draw.**marker**

Bases: `object`

Class to manipulate markers.

This class is used to create new custom markers. Markers can be used with the lineStyle class to define line types that include start, mid and end markers

This class contains only static methods so that you don't have to inherit this in your class

static createMarker(*ExtensionBaseObj, nameID, markerPath, RenameMode=0, strokeColor='#000000', fillColor='#000000', lineWidth=1.0, markerTransform=None*)

Creates a custom line marker

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects

- **nameID** (*string*) – nameID of the marker
- **markerPath** (*string*) – path definition. Must follow 'd' attribute format. See <https://www.w3.org/TR/SVG/paths.html#PathElement> for further information
- **RenameMode** (*int*) – Renaming behavior mode
  - 0: (default) do not rename the marker. If nameID is already taken, the marker will not be modified.
  - 1: overwrite marker definition if nameID is already taken
  - 2: Create a new unique nameID, adding a suffix number (Please refer to inkscapeMadeEasy.uniqueIdNumber(prefix_id)).
- **strokeColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')
- **fillColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')
- **lineWidth** (*float*) – line width of the marker. Default: 1.0
- **markerTransform** (*string*) – custom transform applied to marker's path. Default: `None`

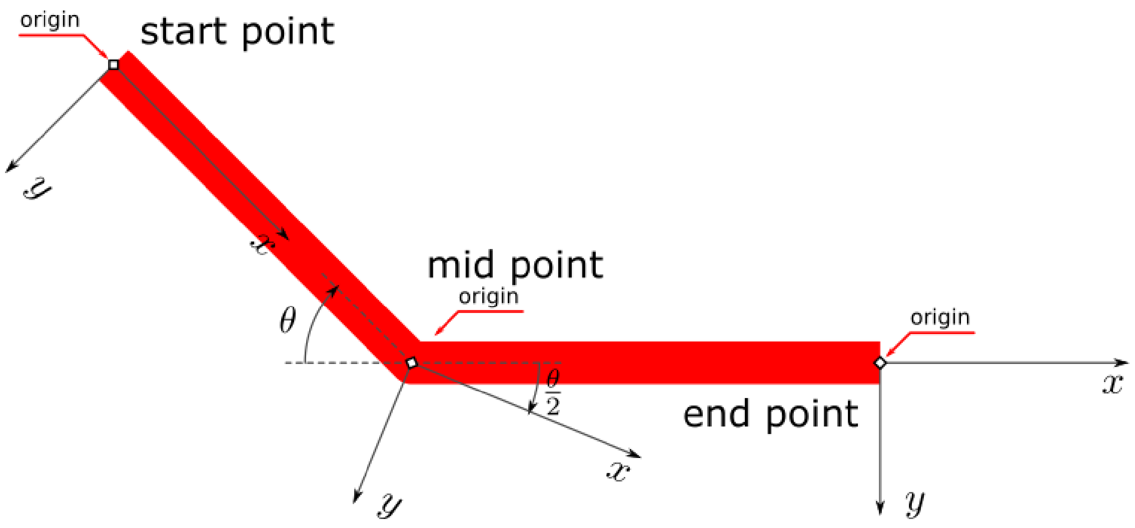
  the transform must follow 'transform' attribute format. See <https://www.w3.org/TR/SVG/coords.html#TransformAttribute> for further information

Returns NameID of the new marker

Return type string

System of coordinates

The system of coordinates of the marker depends on the point under consideration. The following figure presents the coordinate system for all cases



Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
```

(continues on next page)

(continued from previous page)

```
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     nameID='myMarker'
>>>     markerPath='M 3,0 L 0,1 L 0,-1 z'   # defines a path forming an␣
↪triangle with vertices (3,0) (0,1) (0,-1)
>>>     strokeColor=inkDraw.color.defined('red')
>>>     fillColor=None
>>>     RenameMode=1
>>>     width=1
>>>     markerTransform=None
>>>     markerID=inkDraw.marker.createMarker(self,nameID,markerPath,
↪RenameMode,strokeColor,fillColor,width,markerTransform)
>>>     myLineStyle = inkDraw.lineStyle.set(1.0, markerEnd=markerID,
↪lineColor=inkDraw.color.defined('black'))  # see lineStyle class for␣
↪further information on this function
>>>
>>>     #tries to make another marker with the same nameID, changing␣
↪RenameMode
>>>     strokeColor=inkDraw.color.defined('blue')
>>>     RenameMode=0
>>>     markerID=inkDraw.marker.createMarker(self,nameID,RenameMode,scale,
↪strokeColor,fillColor) # this will not modify the marker
>>>     RenameMode=1
>>>     markerID=inkDraw.marker.createMarker(self,nameID,RenameMode,scale,
↪strokeColor,fillColor) # modifies the marker 'myMarker'
>>>     RenameMode=2
>>>     markerID=inkDraw.marker.createMarker(self,nameID,RenameMode,scale,
↪strokeColor,fillColor) # creates a new marker with nameID='myMarker-0001'
```

Note: In next versions, path definition and transformation will be modified to make it easier. =)

static createDotMarker(*ExtensionBaseObj, nameID, RenameMode=0, scale=0.4, strokeColor='#000000', fillColor='#000000'*)

Creates a dotS/M/L marker, exactly like inkscape default markers

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **nameID** (*string*) – nameID of the marker
- **RenameMode** (*int*) – Renaming behavior mode. For more information, see documentation of marker.createMarker(. . .) method.
  - 0: (default) do not rename the marker. If nameID is already taken, the marker will not be modified
  - 1: overwrite marker if nameID is already taken
  - 2: Create a new unique nameID, adding a suffix number (Please refer to inkscapeMadeEasy.uniqueIdNumber(prefix_id)).

- **scale** (*float*) – scale of the marker. To copy exactly inkscape sizes dotS/M/L, use 0.2, 0.4 and 0.8 respectively. Default: 0.4
- **strokeColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')
- **fillColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')

Returns NameID of the new marker

Return type string

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     myMarker=inkDraw.marker.createDotMarker(self,nameID='myDotMarkerA',
↪RenameMode=1,scale=0.5,strokeColor=inkDraw.color.defined('red'),
↪fillColor=None)
>>>     myLineStyle = inkDraw.lineStyle.set(1.0, markerEnd=myMarker,
↪lineColor=inkDraw.color.defined('black'))  # see lineStyle class for␣
↪further information on this function
```

static createCrossMarker(*ExtensionBaseObj*, *nameID*, *RenameMode=0*, *scale=0.4*, *strokeColor='#000000'*, *fillColor='#000000'*)

Creates a cross marker

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **nameID** (*string*) – nameID of the marker
- **RenameMode** (*int*) – Renaming behavior mode. For more information, see documentation of marker.createMarker(. . .) method.
  - 0: (default) do not rename the marker. If nameID is already taken, the marker will not be modified
  - 1: overwrite marker if nameID is already taken
  - 2: Create a new unique nameID, adding a suffix number (Please refer to inkscapeMadeEasy.uniqueIdNumber(prefix_id)).
- **scale** (*float*) – scale of the marker. Default: 0.4
- **strokeColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')
- **fillColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')

Returns NameID of the new marker

Return type string

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     myMarker=inkDraw.marker.createCrossMarker(self,nameID='myDotMarkerA',
↪RenameMode=1,scale=0.5,strokeColor=inkDraw.color.defined('red'),
↪fillColor=None)
>>>     myLineStyle = inkDraw.lineStyle.set(1.0, markerEnd=myMarker,
↪lineColor=inkDraw.color.defined('black'))  # see lineStyle class for␣
↪further information on this function
```

static createArrow1Marker(*ExtensionBaseObj*, *nameID*, *RenameMode=0*, *scale=0.4*, *strokeColor='#000000'*, *fillColor='#000000'*)

Creates a arrowS/M/L arrow markers (both start and end markers), exactly like inkscape

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **nameID** (*string*) – nameID of the marker. Start and End markers will have 'Start' and 'End' suffix respectively
- **RenameMode** (*int*) – Renaming behavior mode. For more information, see documentation of marker.createMarker(. . .) method.
  - 0: (default) do not rename the marker. If nameID is already taken, the marker will not be modified
  - 1: overwrite marker if nameID is already taken
  - 2: Create a new unique nameID, adding a suffix number (Please refer to inkscapeMadeEasy.uniqueIdNumber(prefix_id)).
- **scale** (*float*) – scale of the marker. Default: 0.4
- **strokeColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')
- **fillColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')

Returns a list of strings: [startArrowMarker,endArrowMarker] - startArrowMarker: nameID of start marker - endArrowMarker: nameID of end marker

Return type list

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
```

(continues on next page)

(continued from previous page)

```
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     StartArrowMarker,EndArrowMarker=inkDraw.marker.
↪createArrow1Marker(self,nameID='myArrow',RenameMode=1,scale=0.5,
↪strokeColor=inkDraw.color.defined('red'),fillColor=None)
>>>     myLineStyle = inkDraw.lineStyle.set(1.0, markerStart=StartArrowMarker,
↪markerEnd=EndArrowMarker,lineColor='#000000')  # see lineStyle class for␣
↪further information on this function
```

static createInfLineMarker(*ExtensionBaseObj*, *nameID*, *RenameMode=0*, *scale=1.0*, *strokeColor=None*, *fillColor='#000000'*)

Creates ellipsis markers, both start and end markers.

These markers differ from inkscape's default ellipsis since these markers are made such that the diameter of the dots are equal to the line width.

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **nameID** (*string*) – nameID of the marker. Start and End markers will have 'Start' and 'End' suffix respectively
- **RenameMode** (*int*) – Renaming behavior mode. For more information, see documentation of marker.createMarker(. . .) method.
  - 0: (default) do not rename the marker. If nameID is already taken, the marker will not be modified
  - 1: overwrite marker if nameID is already taken
  - 2: Create a new unique nameID, adding a suffix number (Please refer to inkscapeMadeEasy.uniqueIdNumber(prefix_id)).
- **scale** (*float*) – scale of the marker. Default 1.0
- **strokeColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: `None`
- **fillColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')

Returns a list of strings: [startInfMarker,endInfMarker] - startInfMarker: nameID of start marker - endInfMarker: nameID of end marker

Return type list

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
```

(continues on next page)

(continued from previous page)

```
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     startInfMarker,endInfMarker=inkDraw.marker.createInfLineMarker(self,
↪nameID='myInfMarker',RenameMode=1,scale=1.0,strokeColor=None,fillColor='
↪#00FF00')
>>>     myLineStyle = inkDraw.lineStyle.set(1.0, markerStart=startInfMarker,
↪markerEnd=endInfMarker,lineColor='#000000')  # see lineStyle class for␣
↪further information on this function
```

class `inkscapeMadeEasy_Draw.`**`lineStyle`**

Bases: `object`

Class to create line styles.

This class is used to define line styles. It is capable of setting stroke and filling colors, line width, linejoin and linecap, markers (start, mid, and end) and stroke dash array

This class contains only static methods so that you don't have to inherit this in your class

static set(*lineWidth=1.0*, *lineColor='#000000'*, *fillColor=None*, *lineJoin='round'*, *lineCap='round'*, *markerStart=None*, *markerMid=None*, *markerEnd=None*, *strokeDashArray=None*)

Creates a new line style

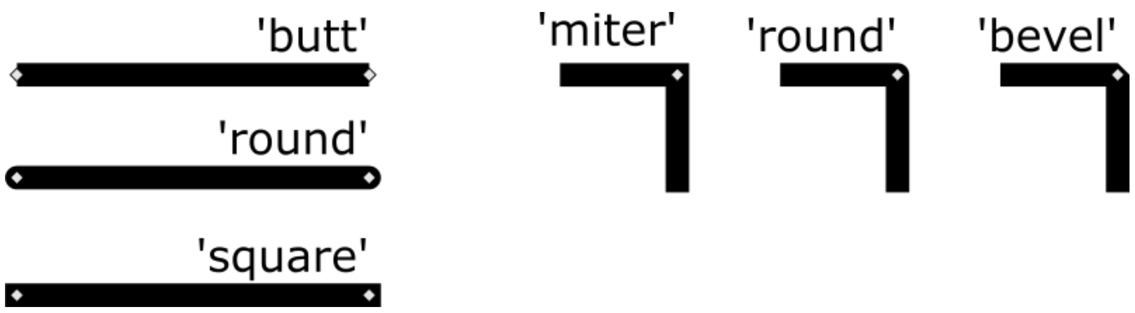
Parameters

- **lineWidth** (*float*) – line width. Default: 1.0
- **lineColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')
- **fillColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: `None`
- **lineJoin** (*string*) – shape of the lines at the joints. Valid values 'miter', 'round', 'bevel'. Default: round
- **lineCap** (*string*) – shape of the lines at the ends. Valid values 'butt', 'square', 'round'. Default: round
- **markerStart** (*string*) – marker at the start node. Default: `None`
- **markerMid** (*string*) – marker at the mid nodes. Default: `None`
- **markerEnd** (*string*) – marker at the end node. Default: `None`
- **strokeDashArray** (*string*) – dashed line pattern definition. Default: `None` See <http://www.w3schools.com/svg/svg_stroking.asp> for further information

Returns line definition following the provided specifications

Return type string

Line node types



Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>
>>>     # creates a line style using a dot marker at its end node
>>>     myMarker=inkDraw.marker.createDotMarker(self,nameID='myMarker',
↪RenameMode=1,scale=0.5,strokeColor=color.defined('red'),fillColor=None)    #␣
↪see marker class for further information on this function
>>>     myLineStyle = inkDraw.lineStyle.set(lineWidth=1.0, markerEnd=myMarker,
↪lineColor=inkDraw.color.defined('black'),fillColor=inkDraw.color('red'))
>>>
>>>     # creates a line style with dashed line (5 units dash , 10 units space
>>>     myDashedStyle = inkDraw.lineStyle.set(lineWidth=1.0,lineColor=inkDraw.
↪color.defined('black'),fillColor=inkDraw.color,strokeDashArray='5,10')
>>>     # creates a line style with a more complex pattern (5 units dash , 10␣
↪units space, 2 units dash, 3 units space
>>>     myDashedStyle = inkDraw.lineStyle.set(lineWidth=1.0,lineColor=inkDraw.
↪color.defined('black'),fillColor=inkDraw.color,strokeDashArray='5,10,2,3')
```

static setSimpleBlack(*lineWidth=1.0*)

Defines a standard black line style.

The only adjustable parameter is its width. The fixed parameters are: lineColor=black, fillColor=None, lineJoin='round', lineCap='round', no markers, no dash pattern

Parameters lineWidth (*float*) – line width. Default: 1.0

Returns line definition following the provided specifications

Return type string

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
```

(continues on next page)

(continued from previous page)

```
>>>     ...
>>>
>>>   def effect(self):
>>>
>>>     mySimpleStyle = inkDraw.lineStyle.setSimpleBlack(lineWidth=2.0)
```

class inkscapeMadeEasy_Draw.**textStyle**

Bases: `object`

Class to create text styles.

This class is used to define text styles. It is capable of setting font size, justification, text color, font family, font style, font weight, line spacing, letter spacing and word spacing

This class contains only static methods so that you don't have to inherit this in your class

static set (*fontSize=10*, *justification='left'*, *textColor='#000000'*, *fontFamily='Sans'*, *fontStyle='normal'*, *fontWeight='normal'*, *lineSpacing='100%'*, *letterSpacing='0px'*, *wordSpacing='0px'*)

Defines a new text style

Parameters

- **fontSize** (*float*) – size of the font in px. Default: 10
- **justification** (*string*) – text justification. `left`, `right`, `center`. Default: `left`
- **textColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')
- **fontFamily** (*string*) – font family name. Default `Sans`
- **fontStyle** (*string*) – `normal` or `italic`. Default: `normal`
- **fontWeight** (*string*) – `normal` or `bold`. Default: `normal`
- **lineSpacing** (*string*) – spacing between lines in percentage. Default: `100%`
- **letterSpacing** (*string*) – extra space between letters. Format: `_px`. Default: `0px`
- **wordSpacing** (*string*) – extra space between words. Format: `_px`. Default: `0px`

Returns text style definition following the provided specifications

Return type string

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>
```

(continues on next page)

(continued from previous page)

```
>>>     myTextStyle=inkDraw.textStyle.set(fontSize=10, justification='left',␣
↪textColor=color.defined('black'), fontFamily='Sans',
>>>                                     fontStyle='normal', fontWeight=
↪'normal', lineSpacing='100%', letterSpacing='0px', wordSpacing='0px')
```

static setSimpleBlack (*fontSize=10*, *justification='left'*)

Defines a standard black text style

The only adjustable parameter are font size and justification. The fixed parameters are: textColor=color.defined('black'), fontFamily='Sans', fontStyle='normal', fontWeight='normal', lineSpacing='100%', letterSpacing='0px', wordSpacing='0px.

Parameters

- **fontSize** (*float*) – size of the font in px. Default: 10
- **justification** (*string*) – text justification. `left`, `right`, `center`. Default: `left`

Returns line definition following the provided specifications

Return type string

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>
>>>     mySimpleStyle = inkDraw.textStyle.setSimpleBlack(fontSize=20,
↪justification='center')
```

static setSimpleColor (*fontSize=10*, *justification='left'*, *textColor='#000000'*)

Defines a standard colored text style

The only adjustable parameter are font size justification and textColor. The fixed parameters are: fontFamily='Sans', fontStyle='normal', fontWeight='normal', lineSpacing='100%', letterSpacing='0px', wordSpacing='0px.

Parameters

- **fontSize** (*float*) – size of the font in px. Default: 10
- **justification** (*string*) – text justification. `left`, `right`, `center`. Default: `left`
- **textColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')

Returns line definition following the provided specifications

Return type string

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>
>>>     mySimpleStyle = inkDraw.textStyle.setSimpleColor(fontSize=20,
↪justification='center',textColor=inkDraw.color.gray(0.5))
```

class `inkscapeMadeEasy_Draw.`**`text`**

Bases: `object`

Class for writing texts. It is possible to add regular inkscape's text elements or LaTeX text. For the later, the excellent 'textext' extension from Pauli Virtanen's <https://pav.iki.fi/software/textext/> is incorporated here. Please refer to *Main Features* section for further instructions

This class contains only static methods so that you don't have to inherit this in your class

Note: LaTeX support is an optional feature, **enabled by default**. Please refer to *Installation and requirements (all users)* on how to disable it.

static write (*ExtensionBaseObj*, *text*, *coords*, *parent*, *textStyle={'fill': '#000000', 'fill-opacity': '1', 'font-family': 'Sans', 'font-size': '10px', 'font-style': 'normal', 'font-weight': 'normal', 'letter-spacing': '0px', 'line-height': '100%', 'stroke': 'none', 'text-align': 'start', 'text-anchor': 'start', 'word-spacing': '0px'}*, *fontSize=None*, *justification=None*, *angleDeg=0.0*)

Adds a text line to the document

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **text** (*string*) – text to be drawn. Use \n in the string to start a new line
- **coords** (*list*) – position [x,y]
- **parent** (*element object*) – parent object
- **textStyle** (*textStyle object*) – text style to be used. See class `textStyle`. Default: textStyle.setSimpleBlack(fontSize=10,justification='left')
- **fontSize** (*float*) – size of the font in px. - `None`: Uses fontSize of textStyle argument (Default) - number: takes precedence over the size on textStyle
- **justification** (*string*) – text justification. `left`, `right`, `center` - `None`: Uses justification of textStyle argument (Default) - `left`, `right`, `center`: takes precedence over the justification set on textStyle
- **angleDeg** (*float*) – angle of the text, counterclockwise, in degrees. Default: 0

Returns the new text object

Return type text Object

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     mySimpleStyle = inkDraw.textStyle.setSimpleBlack(fontSize=20,
↪justification='center')  # creates a simple text style.
>>>
>>>     #adds a two-line text, at the point x=5.0,y=6.0
>>>     #                L1: 'foo bar who-hoo!'
>>>     #                L2: 'second line!'
>>>     myText='foo bar who-hoo!\ntwo lines!'
>>>     inkDraw.text.write(self, text=myText, coords=[5.0,6.0], parent=root_
↪layer, textStyle=mySimpleStyle, fontSize=None, justification=None,␣
↪angleDeg=0.0)
>>>
>>>     # creates a group in root-layer and add text to it
>>>     myGroup = self.createGroup(root_layer,'textGroup')
>>>     #adds a text 'foo bar', rotated 45 degrees, at the point x=0,y=0,␣
↪overriding justification of mySimpleStyle
>>>     inkDraw.text.write(self, text='foo bar', coords=[0.0,0.0],␣
↪parent=myGroup, textStyle=mySimpleStyle, fontSize=None, justification='left
↪', angleDeg=45.0)
```

static latex(*ExtensionBaseObj*, *parent*, *LaTeXtext*, *position*, *fontSize=10*, *refPoint='cc'*, *textColor='#000000'*, *LatexCommands=' '*, *angleDeg=0*, *preambleFile=None*)

Draws a text line using LaTeX. You can use any LaTeX contents here.

Note: Employs the excellent 'textext' extension from Pauli Virtanen's <https://pav.iki.fi/software/textext/> is incorporated here. Please refer to *Main Features* section for further instructions

Note: LaTeX support is an optional feature that requires a few extra packages to be installed outside inkscape. **It is enabled by default**. Please refer to *Installation and requirements (all users)* on how to disable it. If disabled, this function will still work, internally calling the method text.write().

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **parent** (*element object*) – parent object
- **LaTeXtext** (*string*) – text to be drawn. Can contain any latex command
- **position** (*list*) – position of the reference point [x,y]

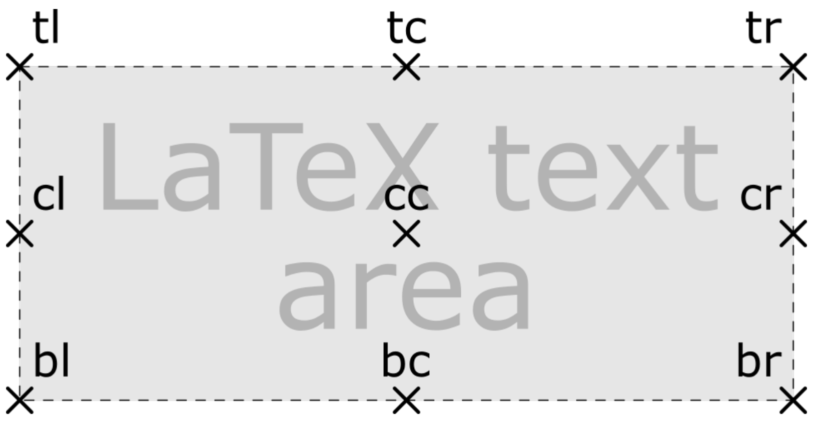
- **fontSize** (*float*) – size of the font. Assume any text of `\normalsize` will have this size. Default: 10
- **refPoint** (*string*) – text reference Point. See figure below for options. Default: `cc`
- **textColor** (*string*) – color in the format `#RRGGBB` (hexadecimal), or `None` for no color. Default: color.defined('black')
- **LatexCommands** (*string*) – commands to be included before LaTeXtext (default: ' '). If LaTeX support is disabled, this parameter has no effect.
- **angleDeg** (*float*) – angle of the text, counterclockwise, in degrees. Default: 0
- **preambleFile** (*string*) – optional preamble file to be included. Default: None. If LaTeX support is disabled, this parameter has no effect.

Returns the new text object

Return type text Object

Note: This function does not use `textStyle` class.

Reference point options



Standard Preamble file

When a preamble file is not provided, inkscapeMadeEasy assumes a standard preamble file located at `./textextLib/basicLatexPackages.tex`. By default, its contents is:

```
\usepackage{amsmath,amsthm,amsbsy,amsfonts,amssymb}
\usepackage[per=slash]{siunitx}
\usepackage{steinmetz}
\usepackage[utf8]{inputenc}
```

You will need these packages installed. This file can be modified to include extra default packages and/or commands.

LaTeX .tex document structure

LaTeX .tex document have the following structure. Note that LatexCommands lies within document environment:

```
\documentclass[landscape,a0]{article}

[contents of Preamble file]
```

(continues on next page)

(continued from previous page)

```
\pagestyle{empty}

\begin{document}
\noindent

[contents of LatexCommands]

[contens of LaTeXtext]

\end{document}
```

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     customCommand = r'\newcommand{\fooBar}{\textbf{Foo Bar Function!␣
↪WhooHoo!}}'   # do not forget the r to avoid backslash escape.
>>>     inkDraw.text.latex(self, root_layer,r'This is one equation \begin
↪{align} x=y^2\end{align} And this is my \fooBar{}',
>>>                        position=[0.0,0.0], fontSize=10, refPoint='cc',␣
↪textColor=inkDraw.color.defined('black'), LatexCommands=customCommand,␣
↪angleDeg=0, preambleFile=None)
```

class inkscapeMadeEasy_Draw.**cubicBezier**

Bases: `object`

This is a class with different methods for drawing cubic bezier lines.

This class contains only static methods so that you don't have to inherit this in your class

static addNode(*NodeList*, *coord=[0, 0]*, *cPbefore=[- 1, 0]*, *cPafter=[1, 0]*, *typeNode='corner'*, *flagAbsCoords=True*)

Add a new node to the list of nodes of the cubic bezier line.

Warning: Keep in mind that Inkscape's y axis is upside down!

Parameters

- **NodeList** (*list*) – list of nodes to be appended with the new node.
- **coord** (*list [x,y]*) – list with the coordinates of the node
- **cPbefore** (*list [x,y]*) – list with the coordinates of the control point before the node.
- **cPafter** (*list [x,y]*) – list with the coordinates of the control point after the node. Used only if 'typeNode' is 'smooth' or 'corner'

- **typeNode** (*string*) – type of node to be added. See image below
  - `corner`: Node without smoothness constraint at the node. The bezier curve can have a sharp edge at the node
  - **smooth: Node with smoothness constraint at the node. The bezier curve will be smooth at the node.** If the control points do not form a straight line, then they are modified to form a straight line. See image below
  - `symmetric`: same as `smooth`, but the control points are forced to be symmetric with respect to the node.
- **flagAbsCoords** (*bool*) – indicate absolute or relative coordinates. See section below on how reference system works. .. warning:: All nodes in a given list must be defined in the same reference system (absolute or relative).

Returns None

Return type

-

Node Types

The image below presents the types of nodes

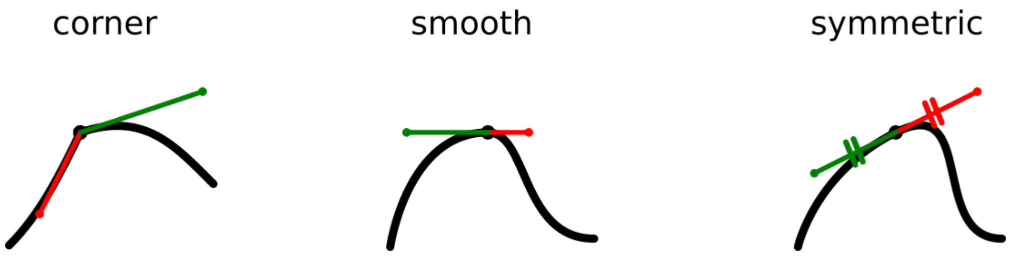
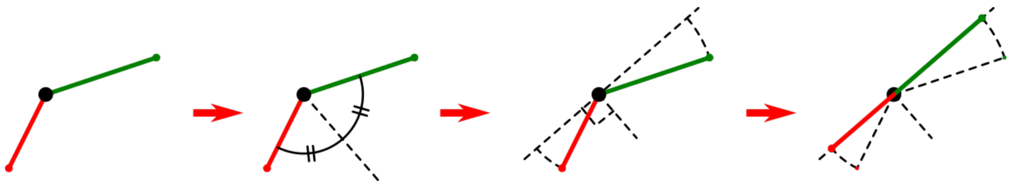
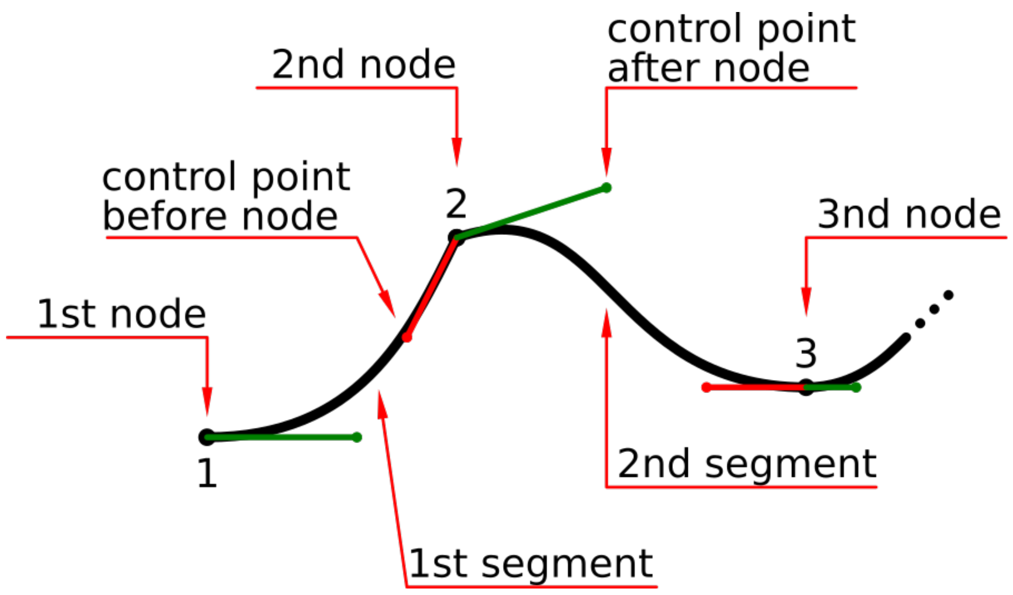


Image below present the process of smoothing control nodes not completely aligned when `smooth` is selected.



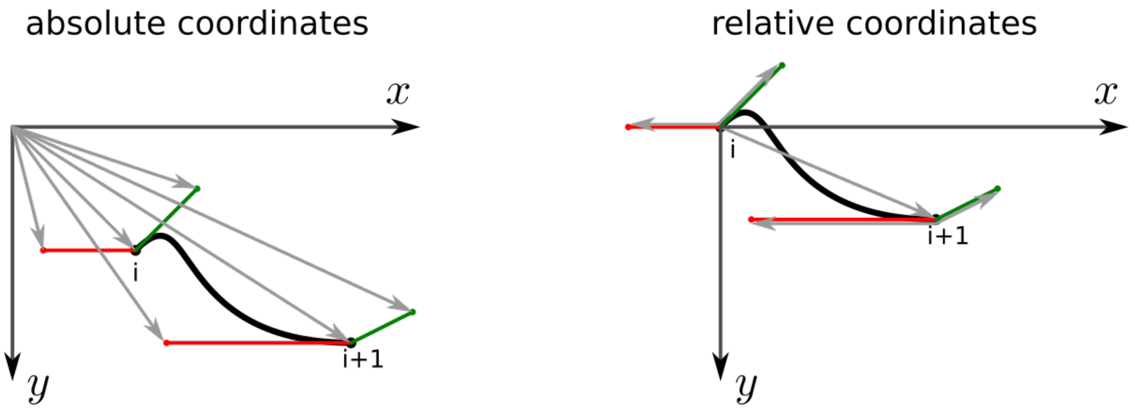
Absolute and relative coordinate systems

cubic bezier curves are composed by segments which are defined by 4 coordinates, two node coordinates and two control points.



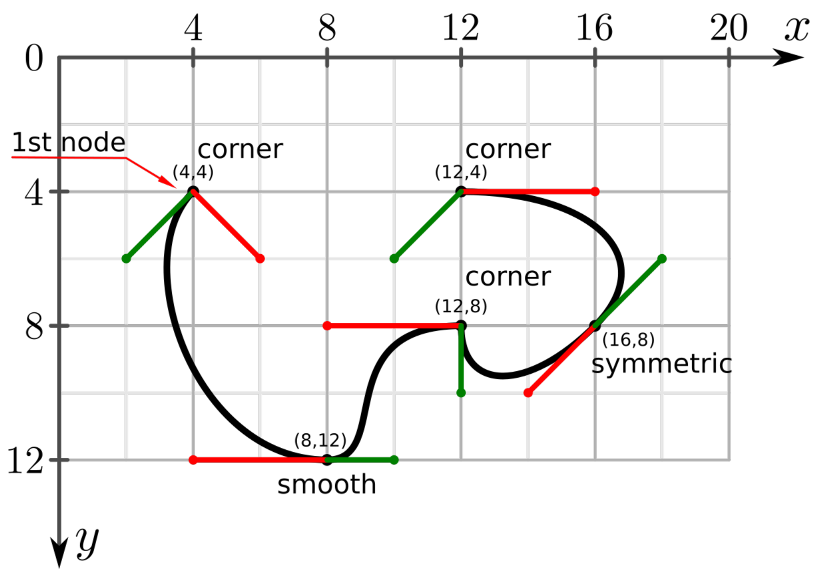
In absolute coordinate system, all node and control point localizations are specified using the origin as reference. In relative coordinate system, control point localizations are specified using its node as reference, and each node use the previous node as reference (the first node use the origin as reference). See image below.

Warning: Keep in mind that Inkscape's y axis is upside down!



Example

Note: In the following example, the control point before the first node and after the last node are important when the bezier curve must be closed. See method `draw`



```
>>>     # create a list of nodes using absolute coordinate system
>>>     nodeListABS=[]
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[4,4], cPbefore=[6,6],␣
↪cPafter=[2,6], typeNode='corner', flagAbsCoords=True)
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[8,12], cPbefore=[4,
↪12], cPafter=[10,12], typeNode='smooth', flagAbsCoords=True)
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[12,8], cPbefore=[8,8],
↪ cPafter=[12,10], typeNode='corner', flagAbsCoords=True)
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[16,8], cPbefore=[14,
↪10], cPafter=None, typeNode='symmetric', flagAbsCoords=True)
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[12,4], cPbefore=[16,
↪4], cPafter=[10,6], typeNode='corner', flagAbsCoords=True)
```

```
>>>     # create a list of nodes using relative coordinate system
>>>     nodeListREL=[]
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[4, 4], cPbefore=[2,2],
↪ cPafter=[-2,2], typeNode='corner', flagAbsCoords=False)
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[4, 8], cPbefore=[-4,
↪0], cPafter=[2,0], typeNode='smooth', flagAbsCoords=False)
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[4, -4], cPbefore=[-4,
↪0], cPafter=[0,2], typeNode='corner', flagAbsCoords=False)
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[4, 0], cPbefore=[-2,
↪2], cPafter=None, typeNode='symmetric', flagAbsCoords=False)
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[-4,-4], cPbefore=[4,
↪0], cPafter=[-2,2], typeNode='corner', flagAbsCoords=False)
```

static draw(*parent*, *NodeList*, *offset=array([0, 0])*, *label='none'*, *lineStyle={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*, *closePath=False*)

draws the bezier line, given a list of nodes, built using `addNode` method

Parameters

- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **NodeList** (*list of nodes*) – list of nodes. See `addNode` method
- **offset** (*list*) – offset coords. Default [0,0]

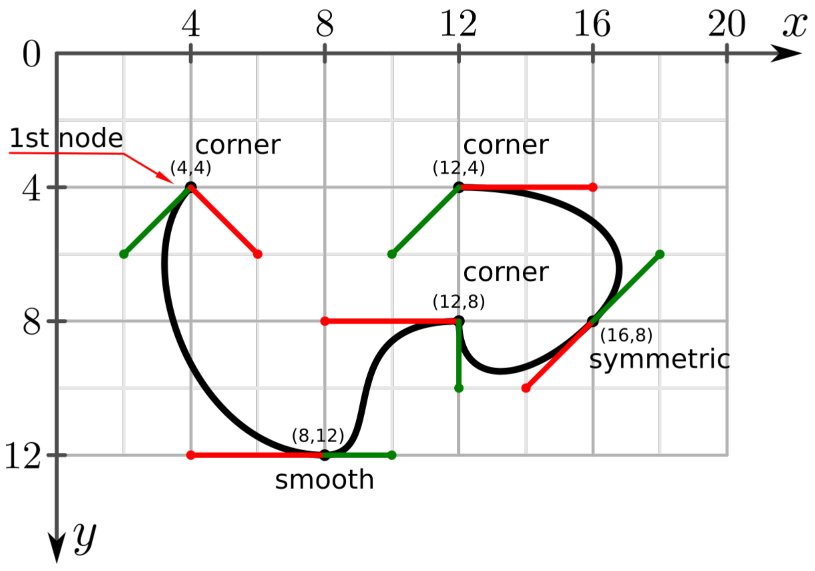
- **label** (*string*) – label of the line. Default 'none'
- **lineStyle** (*lineStyle object*) – line style to be used. See class `lineStyle`. Default: lineStyle=lineStyle.setSimpleBlack()
- **closePath** (*bool*) – Connects the first point to the last. Default: False

Returns the new line object

Return type line Object

Example

Note: In the following example, the control point before the first node and after the last node are important when the bezier curve must be closed.



```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     myLineStyle = set(lineWidth=1.0, lineColor=color.defined('red'))
```

```
>>>     # create a list of nodes using absolute coordinate system
>>>     nodeListABS=[]
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[4,4], cPbefore=[6,6],␣
↪cPafter=[2,6], typeNode='corner', flagAbsCoords=True)
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[8,12], cPbefore=[4,
↪12], cPafter=[10,12], typeNode='smooth', flagAbsCoords=True)
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[12,8], cPbefore=[8,8],
↪ cPafter=[12,10], typeNode='corner', flagAbsCoords=True)
```

(continues on next page)

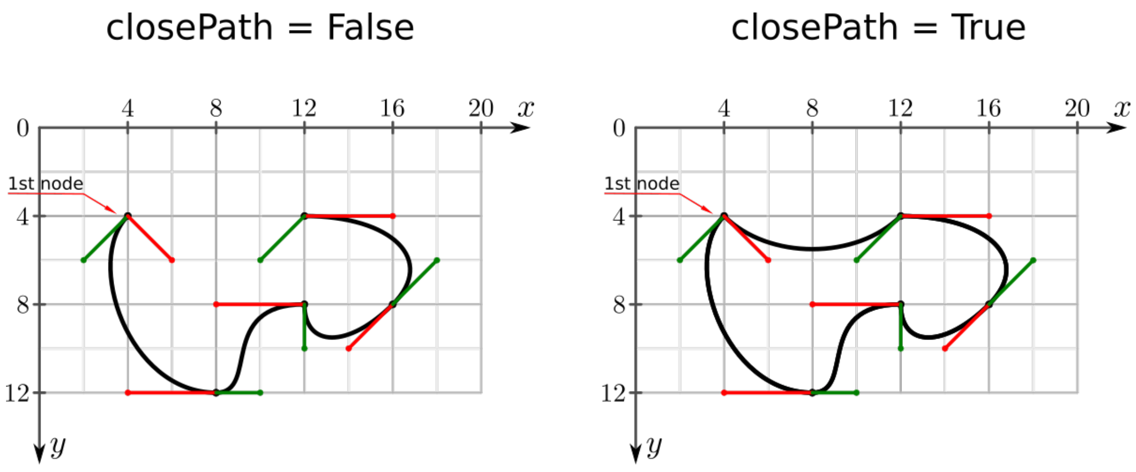
(continued from previous page)

```
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[16,8], cPbefore=[14,
↪10], cPafter=None, typeNode='symmetric', flagAbsCoords=True)
>>>     inkDraw.cubicBezier.addNode(nodeListABS, coord=[12,4], cPbefore=[16,
↪4], cPafter=[10,6], typeNode='corner', flagAbsCoords=True)
```

```
>>>     # create a list of nodes using relative coordinate system
>>>     nodeListREL=[]
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[4, 4], cPbefore=[2,2],
↪ cPafter=[-2,2], typeNode='corner', flagAbsCoords=False)
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[4, 8], cPbefore=[-4,
↪0], cPafter=[2,0], typeNode='smooth', flagAbsCoords=False)
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[4, -4], cPbefore=[-4,
↪0], cPafter=[0,2], typeNode='corner', flagAbsCoords=False)
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[4, 0], cPbefore=[-2,
↪2], cPafter=None, typeNode='symmetric', flagAbsCoords=False)
>>>     inkDraw.cubicBezier.addNode(nodeListREL, coord=[-4,-4], cPbefore=[4,
↪0], cPafter=[-2,2], typeNode='corner', flagAbsCoords=False)
```

```
>>>     C1 = inkDraw.cubicBezier.draw(root_layer,nodeListABS, offset=[0, 0],
↪closePath=False)
>>>     C2 = inkDraw.cubicBezier.draw(root_layer,nodeListABS, offset=[0, 0],
↪closePath=True)
>>>     C3 = inkDraw.cubicBezier.draw(root_layer,nodeListREL, offset=[0, 0],
↪closePath=False)
>>>     C4 = inkDraw.cubicBezier.draw(root_layer,nodeListREL, offset=[0, 0],
↪closePath=True)
```

Result of the example



class inkscapeMadeEasy_Draw.**line**

Bases: `object`

This is a class with different methods for drawing lines.

This class contains only static methods so that you don't have to inherit this in your class

static absCoords(*parent*, *coordsList*, *offset=[0, 0]*, *label='none'*, *lineStyle={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*, *closePath=False*)

Draws a (poly)line based on a list of absolute coordinates

Warning: Keep in mind that Inkscape's y axis is upside down!

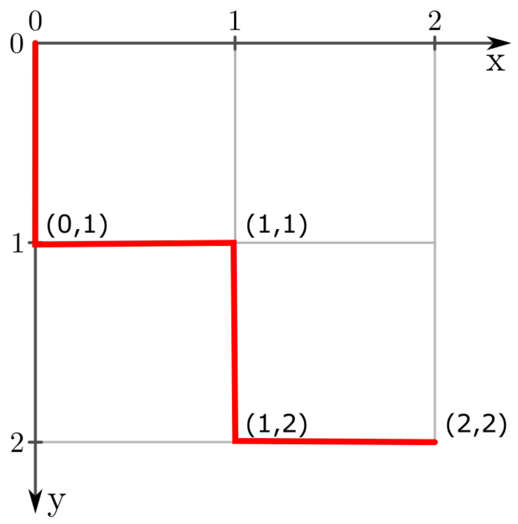
Parameters

- **`parent`** (*`inkscapeMadeEasy object (see example below)`*) – parent object
- **`coordsList`** (*`list of list`*) – list with coords x and y. ex [[x1,y1], . . . , [xN,yN]]
- **`offset`** (*`list`*) – offset coords. Default [0,0]
- **`label`** (*`string`*) – label of the line. Default 'none'
- **`lineStyle`** (*`lineStyle object`*) – line style to be used. See class `lineStyle`. Default: lineStyle=lineStyle.setSimpleBlack()
- **`closePath`** (*`bool`*) – Connects the first point to the last. Default: False

Returns the new line object

Return type line Object

Example



```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     myLineStyle = set(lineWidth=1.0, lineColor=color.defined('red'))
>>>
>>>
>>>     # creates a polyline passing by points (0,0) (0,1) (1,1) (1,2) (2,2)␣
↪using absolute coordinates
```

(continues on next page)

(continued from previous page)

```
>>>     coords=[ [0,0], [0,1], [1,1], [1,2], [2,2] ]
>>>     inkDraw.line.absCoords(root_layer, coordsList=coords, offset=[0, 0],␣
↪label='fooBarLine', lineStyle=myLineStyle)
>>>
>>>     # creates the same polyline translated to point (5,6). Note we just␣
↪have to change the offset
>>>     inkDraw.line.absCoords(root_layer, coordsList=coords, offset=[5, 6],␣
↪label='fooBarLine', lineStyle=myLineStyle)
```

static relCoords(*parent*, *coordsList*, *offset=[0, 0]*, *label='none'*, *lineStyle={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*, *closePath=False*)

Draws a (poly)line based on a list of relative coordinates

Warning: Keep in mind that Inkscape's y axis is upside down!

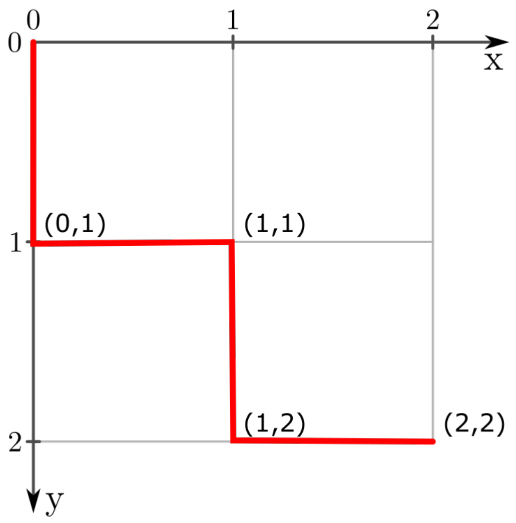
Parameters

- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **coordsList** (*list of list*) – list with distances dx and dy for all points. ex [[dx1,dy1], . . . , [dxN,dyN]]
- **offset** (*list*) – offset coords. Default [0,0]
- **label** (*string*) – label of the line. Default 'none'
- **lineStyle** (*lineStyle object*) – line style to be used. See class `lineStyle`. Default: lineStyle=lineStyle.setSimpleBlack()
- **closePath** (*bool*) – Connects the first point to the last. Default: False

Returns the new line object

Return type line Object

Example



```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     myLineStyle = setSimpleBlack(lineWidth=1.0)
>>>
>>>
>>>     # creates a polyline passing by points (0,0) (0,1) (1,1) (1,2) (2,2)␣
↪using relative coordinates
>>>     coords=[ [0,1], [1,0], [0,1], [1,0] ]
>>>     inkDraw.line.relCoords(root_layer, coordsList=coords, offset=[0, 0],␣
↪label='fooBarLine', lineStyle=myLineStyle)
>>>
>>>     # creates the same polyline translated to point (5,6)
>>>     inkDraw.line.relCoords(root_layer, coordsList=coords, offset=[5, 6],␣
↪label='fooBarLine', lineStyle=myLineStyle)
```

class `inkscapeMadeEasy_Draw.`**`arc`**

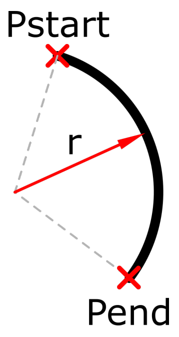
Bases: `object`

This is a class with different methods for drawing arcs.

This class contains only static methods so that you don't have to inherit this in your class

static startEndRadius(*parent*, *Pstart*, *Pend*, *radius*, *offset=[0, 0]*, *label='arc'*, *lineStyle={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*, *flagRightOf=True*, *flagOpen=True*, *largeArc=False*)

Draws a circle arc from `Pstart` to `Pend` with a given radius



Parameters

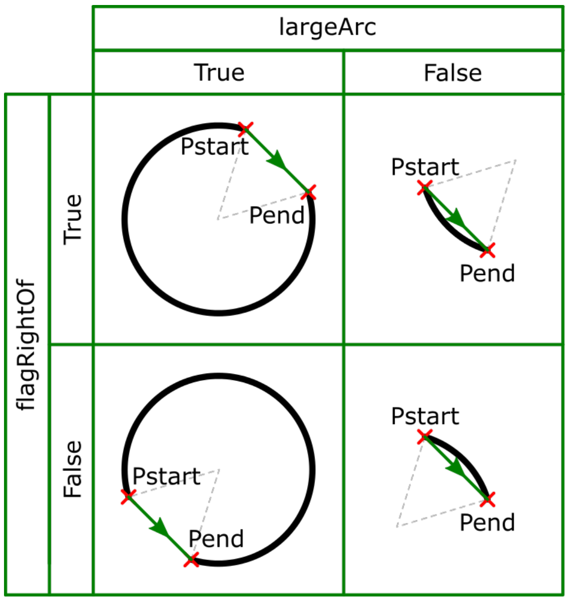
- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **Pstart** (*list*) – start coordinate [x,y]
- **Pend** (*list*) – end coordinate [x,y]
- **radius** (*float*) – arc radius

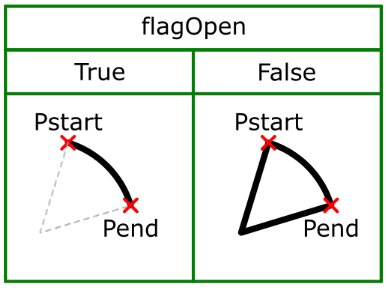
- **offset** (*list*) – extra offset coords [x,y]. Default [0,0]
- **label** (*string*) – label of the line. Default 'arc'
- **lineStyle** (*lineStyle object*) – line style to be used. See class `lineStyle`. Default: lineStyle=lineStyle.setSimpleBlack()
- **flagRightOf** (*bool*) – sets the side of the vector Pend-Pstart which the arc must be drawn. See image below
  - True: Draws the arc to the right (Default)
  - False: Draws the arc to the left
- **flagOpen** (*bool*) – closes the arc. See image below. Default: True
- **largeArc** (*bool*) – Sets the largest arc to be drawn. See image below
  - True: Draws the largest arc
  - False: Draws the smallest arc (Default)

Returns the new arc object

Return type line Object

Arc options





Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
```

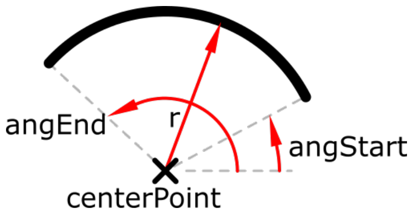
(continues on next page)

(continued from previous page)

```
>>>
>>>     P1=[10.0,0.0]
>>>     P2=[20.0,10.0]
>>>     R=15.0
>>>     myLineStyle=inkDraw.lineStyle.setSimpleBlack()
>>>
>>>     #draws an opened arc
>>>     inkDraw.arc.startEndRadius(parent=root_layer, Pstart=P1, Pend=P2,␣
↪radius=R, offset=[25,0], label='arc1',  lineStyle=myLineStyle,␣
↪flagOpen=True)
>>>
>>>     #draws a closed arc
>>>     inkDraw.arc.startEndRadius(parent=root_layer, Pstart=P1, Pend=P2,␣
↪radius=R, offset=[25,20], label='arc2',  lineStyle=myLineStyle,␣
↪flagOpen=False)
>>>
>>>     #draws arcs with all combinations of flagRightOf and largeArc␣
↪parameters
>>>     inkDraw.arc.startEndRadius(parent=root_layer, Pstart=P1, Pend=P2,␣
↪radius=R, offset=[0,0], label='arc',  lineStyle=myLineStyle,␣
↪flagRightOf=True, largeArc=True)
>>>     inkDraw.arc.startEndRadius(parent=root_layer, Pstart=P1, Pend=P2,␣
↪radius=R, offset=[25,0], label='arc4',  lineStyle=myLineStyle,␣
↪flagRightOf=False, largeArc=True)
>>>     inkDraw.arc.startEndRadius(parent=root_layer, Pstart=P1, Pend=P2,␣
↪radius=R, offset=[0,40], label='arc5',  lineStyle=myLineStyle,␣
↪flagRightOf=True, largeArc=False)
>>>     inkDraw.arc.startEndRadius(parent=root_layer, Pstart=P1, Pend=P2,␣
↪radius=R, offset=[25,40], label='arc6',  lineStyle=myLineStyle,␣
↪flagRightOf=False, largeArc=False)
```

static centerAngStartAngEnd(*parent, centerPoint, radius, angStart, angEnd, offset=[0, 0], label='arc', lineStyle={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}, flagOpen=True, largeArc=False*)

Draws a circle arc given its center and start and end angles



Warning: Keep in mind that Inkscape's y axis is upside down!

Parameters

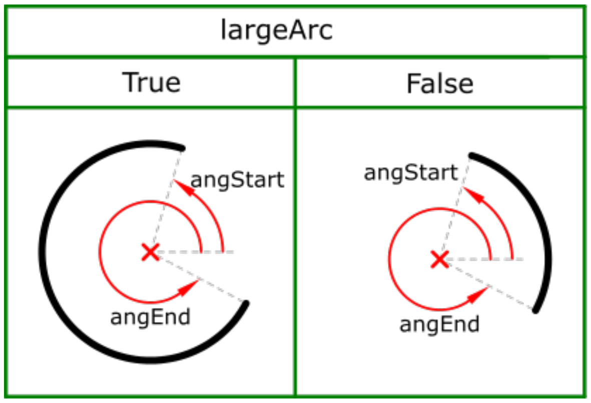
- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **centerPoint** (*list*) – center coordinate [x,y]
- **radius** (*float*) – arc radius
- **angStart** (*float*) – start angle in degrees

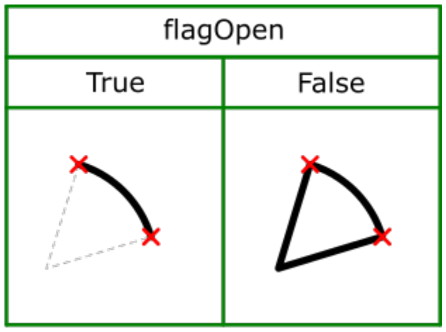
- **angEnd** (*float*) – end angle in degrees
- **offset** (*list*) – extra offset coords [x,y]
- **label** (*string*) – label of the line. Default ‘arc’
- **lineStyle** (*lineStyle object*) – line style to be used. See class `lineStyle`. Default: lineStyle=lineStyle.setSimpleBlack()
- **flagOpen** (*bool*) – closes the arc. See image below. Default: True
- **largeArc** (*bool*) – Sets the largest arc to be drawn. See image below
  - True: Draws the largest arc
  - False: Draws the smallest arc (Default)

Returns the new arc object

Return type line Object

Arc options





Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     myLineStyle=inkDraw.lineStyle.setSimpleBlack()
>>>
>>>     #draws the shortest arc
>>>     inkDraw.arc.centerAngStartAngEnd(parent=root_layer, centerPoint=[0,0],
↪ radius=15.0, angStart=-10, angEnd=90,
>>>                                      offset=[0,0], label='arc1', ␣
↪lineStyle=myLineStyle, flagOpen=True,largeArc=False)
>>>     #draws the longest arc
```

(continues on next page)

(continued from previous page)

```
>>>     inkDraw.arc.centerAngStartAngEnd(parent=root_layer, centerPoint=[0,0],
↪ radius=15.0, angStart=-10, angEnd=90,
>>>                                       offset=[30,0], label='arc1', ␣
↪lineStyle=myLineStyle, flagOpen=True,largeArc=True)
```

class `inkscapeMadeEasy_Draw.`**circle**

Bases: `object`

This is a class with different methods for drawing circles.

This class contains only static methods so that you don't have to inherit this in your class

static centerRadius(*parent*, *centerPoint*, *radius*, *offset=[0, 0]*, *label='circle'*, *lineStyle={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*)

draws a circle given its center point and radius

Warning: Keep in mind that Inkscape's y axis is upside down!

Parameters

- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **centerPoint** (*list*) – center coordinate [x,y]
- **radius** (*float*) – circle's radius
- **offset** (*list*) – extra offset coords [x,y]
- **label** (*string*) – label of the line. Default 'circle'
- **lineStyle** (*lineStyle object*) – line style to be used. See class `lineStyle`. Default: lineStyle=lineStyle.setSimpleBlack()

Returns the new circle object

Return type line Object

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     myLineStyle=inkDraw.lineStyle.setSimpleBlack()
>>>
>>>     #draws the shortest arc
>>>     inkDraw.circle.centerRadius(parent=root_layer, centerPoint=[0,0],␣
↪radius=15.0, offset=[5,1], label='circle1',  lineStyle=myLineStyle)
```

class inkscapeMadeEasy_Draw.**rectangle**

Bases: `object`

This is a class with different methods for drawing rectangles.

This class contains only static methods so that you don't have to inherit this in your class

static widthHeightCenter(*parent*, *centerPoint*, *width*, *height*, *radiusX=None*, *radiusY=None*, *offset=[0, 0]*, *label='rectangle'*, *lineStyle={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*)

draws a rectangle given its center point and dimensions

Warning: Keep in mind that Inkscape's y axis is upside down!

Parameters

- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **centerPoint** (*list*) – center coordinate [x,y]
- **width** (*float*) – dimension in X direction
- **height** (*float*) – dimension in Y direction
- **radiusX** (*float*) – rounding radius in X direction. If this value is `None`, the rectangle will have sharp corners. Default: None
- **radiusY** (*float*) – rounding radius in Y direction. If this value is `None`, then radiusX will also be used in Y direction. If radiusX is also `None`,then the rectangle will have sharp corners. Default: None
- **offset** (*list*) – extra offset coords [x,y]
- **label** (*string*) – label of the line. Default 'circle'
- **lineStyle** (*lineStyle object*) – line style to be used. See class `lineStyle`. Default: lineStyle=lineStyle.setSimpleBlack()

Returns the new rectangle object

Return type rectangle Object

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     myLineStyle=inkDraw.lineStyle.setSimpleBlack()
>>>
```

(continues on next page)

(continued from previous page)

```
>>>     #draws a 50x60 rectangle with radiusX=2.0 and radiusY=3.0
>>>     inkDraw.rectangle.widthHeightCenter(parent=root_layer, centerPoint=[0,
↪0], width=50, height=60, radiusX=2.0,radiusY=3.0, offset=[0,0], label='rect1
↪',  lineStyle=myLineStyle)
```

static corners(*parent, corner1, corner2, radiusX=None, radiusY=None, offset=[0, 0], label='rectangle', lineStyle={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*)

draws a rectangle given the coordinates of two oposite corners

Warning: Keep in mind that Inkscape's y axis is upside down!

Parameters

- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **corner1** (*list*) – coordinates of corner 1 [x,y]
- **corner2** (*list*) – coordinates of corner 1 [x,y]
- **radiusX** (*float*) – rounding radius in X direction. If this value is `None`, the rectangle will have sharp corners. Default: None
- **radiusY** (*float*) – rounding radius in Y direction. If this value is `None`, then radiusX will also be used in Y direction. If radiusX is also `None`, then the rectangle will have sharp corners. Default: None
- **offset** (*list*) – extra offset coords [x,y]
- **label** (*string*) – label of the line. Default 'circle'
- **lineStyle** (*lineStyle object*) – line style to be used. See class `lineStyle`. Default: lineStyle=lineStyle.setSimpleBlack()

Returns the new rectangle object

Return type rectangle Object

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     myLineStyle=inkDraw.lineStyle.setSimpleBlack()
>>>
>>>     #draws a rectangle with corners C1=[1,5] and C2=[6,10], with␣
↪radiusX=2.0 and radiusY=3.0
>>>     inkDraw.rectangle.corners(parent=root_layer, corner1=[1,5],␣
↪corner2=[6,10], radiusX=2.0,radiusY=3.0, offset=[0,0], label='rect1',␣
↪lineStyle=myLineStyle)
```

(continues on next page)

(continued from previous page)

class inkscapeMadeEasy_Draw.**ellipse**

Bases: `object`

This is a class with different methods for drawing ellipses.

This class contains only static methods so that you don't have to inherit this in your class

static centerRadius(*parent, centerPoint, radiusX, radiusY, offset=[0, 0], label='circle', lineStyle={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*)

draws an ellipse given its center point and radius

Warning: Keep in mind that Inkscape's y axis is upside down!

Parameters

- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **centerPoint** (*list*) – center coordinate [x,y]
- **radiusX** (*float*) – circle's radius in x direction
- **radiusY** (*float*) – circle's radius in y direction
- **offset** (*list*) – extra offset coords [x,y]
- **label** (*string*) – label of the line. Default 'circle'
- **lineStyle** (*lineStyle object*) – line style to be used. See class `lineStyle`. Default: lineStyle=lineStyle.setSimpleBlack()

Returns the new ellipse object

Return type line Object

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()      # retrieves the root layer␣
↪of the document
>>>     myLineStyle=inkDraw.lineStyle.setSimpleBlack()
>>>
>>>     #draws the shortest arc
>>>     inkDraw.ellipse.centerRadius(parent=root_layer, centerPoint=[0,0],␣
↪radiusX=15.0, radiusY=25.0, offset=[5,1], label='circle1', ␣
↪lineStyle=myLineStyle)
```

7.3 inkscapeMadeEasy_Plot

`inkscapeMadeEasy_Plot.`**`displayMsg`**`(`*msg*`)`

Displays a message to the user.

Returns nothing

Return type

-

Note: Identical function has been also defined inside inkscapeMadeEasy class

`inkscapeMadeEasy_Plot.`**`Dump`**`(`*obj*, *file='./dump_file.txt'*, *mode='w'*`)`

Function to easily output the result of `str(obj)` to a file

This function was created to help debugging the code while it is running under inkscape. Since inkscape does not possess a terminal as today (2016), this function overcomes partially the issue of sending things to stdout by dumping result of the function `str()` in a text file.

Parameters

- **`obj`** (any, as long as `str(obj)` is implemented (see `__str__()` metaclass definition)) – object to sent to the file. Any type that can be used in `str()`
- **`file`** – file path. Default: `./dump_file.txt`
- **`mode`** (*`string`*) – writing mode of the file. Default: `w` (write)

Returns nothing

Return type

-

Note: Identical function has been also defined inside inkscapeMadeEasy class

Example

```
>>> vector1=[1,2,3,4,5,6]
>>> Dump(vector1,file='~/temporary.txt',mode='w')   # writes the list to a file
>>> vector2=[7,8,9,10]
>>> Dump(vector2,file='~/temporary.txt',mode='a')   # append the list to a file
```

`class` `inkscapeMadeEasy_Plot.`**`axis`**

Bases: `object`

This is a class with different methods for making plot axes.

This class contains only static methods so that you don't have to inherit this in your class

Note: This class uses LaTeX in labels and tick marks if LaTeX support is enabled. This is an optional feature, **enabled by default**. Please refer to *Installation and requirements (all users)* on how to disable it.

static cartesian(*ExtensionBaseObj*, *parent*, *xLim*, *yLim*, *position=[0, 0]*, *xLabel="*, *yLabel="*, *xlog10scale=False*, *ylog10scale=False*, *xTicks=True*, *yTicks=True*, *xTickStep=1.0*, *yTickStep=1.0*, *xScale=20*, *yScale=20*, *xAxisUnitFactor="*, *yAxisUnitFactor="*, *xGrid=False*, *yGrid=False*, *forceTextSize=0*, *forceLineWidth=0*, *drawAxis=True*, *ExtraLenghtAxisX=0.0*, *ExtraLenghtAxisY=0.0*)

Creates the axes for cartesian plot

Note: This method uses LaTeX in labels and tick marks if LaTeX support is enabled. This is an optional feature, **enabled by default**. Please refer to *Installation and requirements (all users)* on how to disable it.

Parameters

- **ExtensionBaseObj** (`inkscapeMadeEasy object (see example below)`) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **parent** (`inkscapeMadeEasy object (see example below)`) – parent object
- **xLim** (`list`) – limits of the X axis [x_min,x_max]. If the axis is in log10 scale, then the limits will be rounded to complete one decade.
- **yLim** (`list`) – limits of the Y axis [y_min,y_max]. If the axis is in log10 scale, then the limits will be rounded to complete one decade.
- **position** (`list`) – position of the point where x and y axis cross [x0,y0]. The point where the axis cross depend on the limits.
  - If xLimits comprises the origin x=0, then the Y axis crosses the X axis at x=0.
  - If xLimits contains only negative numbers, then the Y axis crosses the X axis at x_max.
  - If xLimits contains only positive numbers, then the Y axis crosses the X axis at x_min.
  - The same rule applies to y direction.
- **xLabel** (`string`) – Label of the X axis. Default: ''

  The text can contain any LaTeX command. If you want to write mathematical text, you can enclose it between dollar signs $...$. If LaTeX support is disabled, do not use $.
- **yLabel** (`string`) – Label of the Y axis. Default: ''

  The text can contain any LaTeX command. If you want to write mathematical text, you can enclose it between dollar signs $...$. If LaTeX support is disabled, do not use $.
- **xlog10scale** (`bool`) – sets X axis to log10 scale if True. Default: False
- **ylog10scale** (`bool`) – sets Y axis to log10 scale if True. Default: False
- **xTicks** (`bool`) – Adds axis ticks to the X axis if True. Default: True
- **yTicks** (`bool`) – Adds axis ticks to the Y axis if True. Default: True
- **xTickStep** (`float`) – Value interval between two consecutive ticks on X axis. (Not used if X axis is in log10 scale). Default:1.0
- **yTickStep** (`float`) – Value interval between two consecutive ticks on Y axis. (Not used if Y axis is in log10 scale). Default:1.0
- **xScale** (`float`) – Distance between each xTickStep in svg units. Default: 20
  - If axis is linear, then xScale is the size in svg units of each tick

  - If axis is log10, the xScale is the size in svg units of one decade
- **yScale** (*float*) – Distance between each xTickStep in svg units. Default: 20
  - If axis is linear, then xScale is the size in svg units of each tick
  - If axis is log10, the xScale is the size in svg units of one decade
- **xAxisUnitFactor** (*string*) – extra text to be added to Ticks in both x and y. Default: ‘’

  This is useful when we want to represent interval with different units. example pi, 2pi 3pi, etc. The text can be any LaTeX text. Keep in mind that this text will be inserted within a mathematical environment \$. . . \$, therefore no \$ is needed here.
- **yAxisUnitFactor** (*string*) – extra text to be added to the ticks in Y axis. Default: ‘’

  This is useful when we want to represent interval with different units. example pi, 2pi 3pi, etc. The text can be any LaTeX text. Keep in mind that this text will be inserted within a mathematical environment \$. . . \$, therefore no \$ is needed here.
- **xGrid** (*bool*) – adds grid lines to X axis if true. Default: False
- **yGrid** (*bool*) – adds grid lines to Y axis if true. Default: False
- **forceTextSize** (*float*) – Size of the text. If this parameter is 0.0 then the method will compute an appropriate size. Default: 0.0
- **forceLineWidth** (*float*) – Width of the lines. If this parameter is 0.0 then the method will compute an appropriate size. Default: 0.0
- **drawAxis** (*bool*) – control flag of the axis method
  - True: draws axis normally
  - False: returns the limits and origin position without drawing the axis itself
- **ExtraLenghtAxisX** (*float*) – extra length left near the arrow pointer of X axis. Default 0.0
- **ExtraLenghtAxisY** (*float*) – extra length left near the arrow pointer of Y axis. Default 0.0

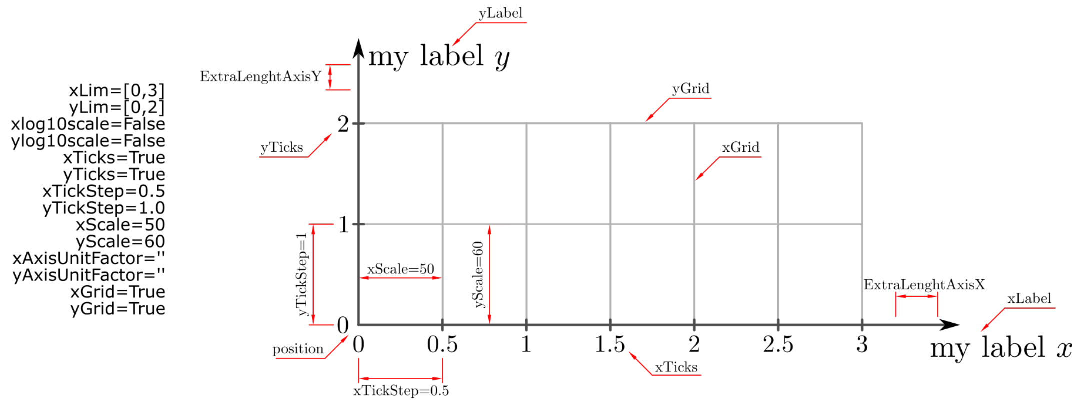
Returns

[GroupPlot, outputLimits, axisOrigin]

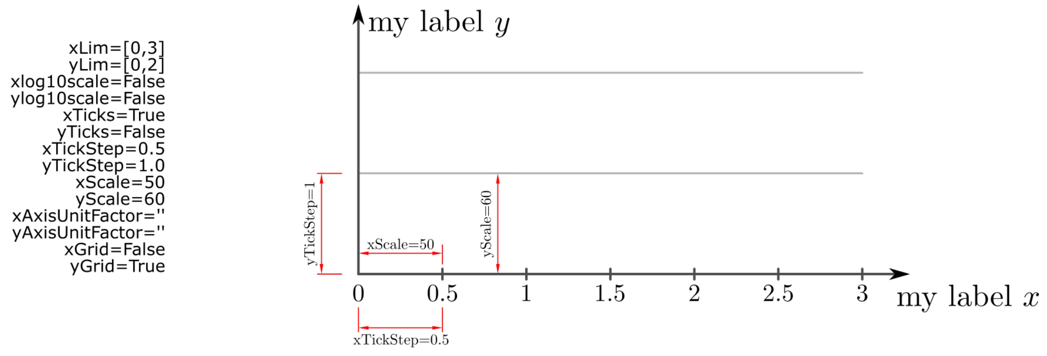
- GroupPlot: the axis area object (if drawAxis=False, this output is `None`)
- outputLimits: a list with tuples:[(x_min,xPos_min),(x_max,xPos_max),(y_min,yPos_min),(y_max,yPos_max)]
  - x_min, x_max, y_min, y_max: The limits of the axis object
  - xPos_min, xPos_max, yPos_min, yPos_max: The positions of the limits of the axis object, considering the scaling and units
  - axisOrigin [X0,Y0]: A list with the coordinates of the point where the axes cross.

Return type list

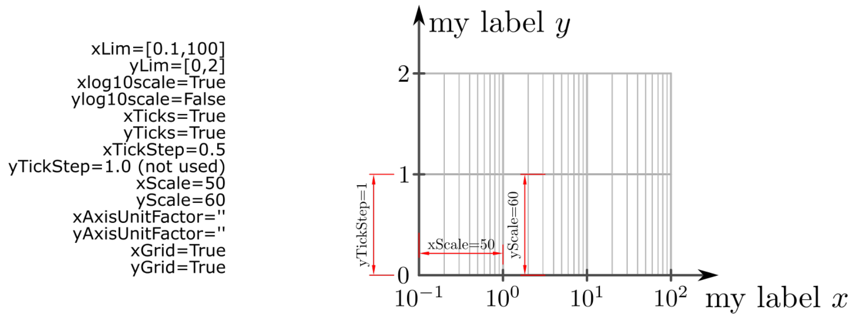
Examples



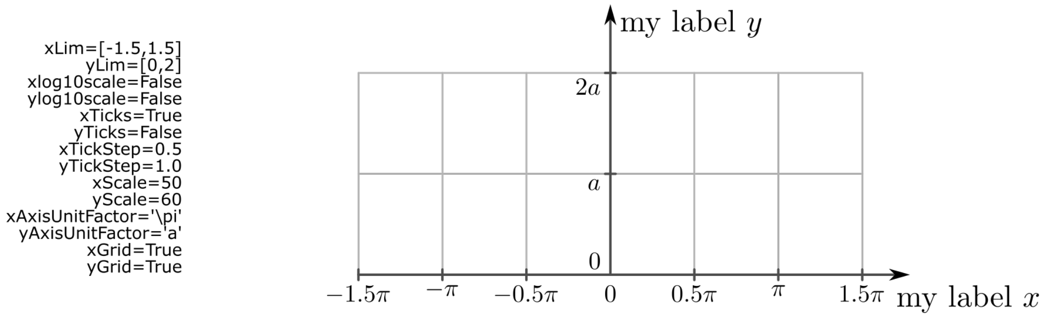
```
xLim=[0,3]
yLim=[0,2]
xlog10scale=False
ylog10scale=False
xTicks=True
yTicks=True
xTickStep=0.5
yTickStep=1.0
xScale=50
yScale=60
xAxisUnitFactor=''
yAxisUnitFactor=''
xGrid=True
yGrid=True
```



```
xLim=[0,3]
yLim=[0,2]
xlog10scale=False
ylog10scale=False
xTicks=True
yTicks=False
xTickStep=0.5
yTickStep=1.0
xScale=50
yScale=60
xAxisUnitFactor=''
yAxisUnitFactor=''
xGrid=False
yGrid=True
```



```
xLim=[0.1,100]
yLim=[0,2]
xlog10scale=True
ylog10scale=False
xTicks=True
yTicks=True
xTickStep=0.5
yTickStep=1.0 (not used)
xScale=50
yScale=60
xAxisUnitFactor=''
yAxisUnitFactor=''
xGrid=True
yGrid=True
```



```
xLim=[-1.5,1.5]
yLim=[0,2]
xlog10scale=False
ylog10scale=False
xTicks=True
yTicks=False
xTickStep=0.5
yTickStep=1.0
xScale=50
yScale=60
xAxisUnitFactor='\pi'
yAxisUnitFactor='a'
xGrid=True
yGrid=True
```

static polar (*ExtensionBaseObj*, *parent*, *rLim*, *tLim=[0.0, 360.0]*, *position=[0.0, 0.0]*, *rLabel=''*, *rlog10scale=False*, *rTicks=True*, *tTicks=True*, *rTickStep=1.0*, *tTickStep=45.0*, *rScale=20*, *rAxisUnitFactor=''*, *rGrid=False*, *tGrid=False*, *forceTextSize=0*, *forceLineWidth=0*, *drawAxis=True*, *ExtraLenghtAxisR=0.0*)

Creates the axes for polar plot

Note: This method uses LaTeX in labels and tick marks if LaTeS support is enabled. This is an optional feature, **enabled by default**. Please refer to *Installation and requirements (all users)* on how to disable it.

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **rLim** (*list*) – limits of the R axis [r_min,r_max]. If the axis is in log10 scale, then the limits will be rounded to complete one decade.
- **tLim** (*list*) – limits of the theta axis [t_min,t_max]. Values in degrees. Default: [0,360]
- **position** (*list*) – position of the center [x0,y0].
- **rLabel** (*string*) – Label of the R axis. Default: ''

  The text can contain any LaTeX command. If you want to write mathematical text, you can enclose it between dollar signs $. . . $. If LaTeX support is disabled, do not use $.
- **rlog10scale** (*bool*) – sets R axis to log10 scale if True. Default: False
  - If rlog10scale=True, then the lower limit of rLim must be >=1
- **rTicks** (*bool*) – Adds axis ticks to the R axis if True. Default: True
- **tTicks** (*bool*) – Adds axis ticks to the theta axis if True. Default: True
- **rTickStep** (*float*) – Value interval between two consecutive ticks on R axis. (Not used if R axis is in log10 scale). Default:1.0
- **tTickStep** (*float*) – Value interval between two consecutive ticks on theta axis. Default:45.0
- **rScale** (*float*) – Distance between each rTickStep in svg units. Default: 20
  - If axis is linear, then rScale is the size in svg units of each tick
  - If axis is log10, the rScale is the size in svg units of one decade
- **rAxisUnitFactor** (*string*) – extra text to be added to Ticks in both x and y. Default: ''

  This is useful when we want to represent interval with different units. example pi, 2pi 3pi, etc. The text can be any LaTeX text. Keep in mind that this text will be inserted within a mathematical environment $. . . $, therefore no $ is needed here.
- **rGrid** (*bool*) – adds grid lines to R axis if true. Default: False
- **tGrid** (*bool*) – adds grid lines to theta axis if true. Default: False
- **forceTextSize** (*float*) – Size of the text. If this parameter is 0.0 then the method will compute an appropriate size. Default: 0.0
- **forceLineWidth** (*float*) – Width of the lines. If this parameter is 0.0 then the method will compute an appropriate size. Default: 0.0
- **drawAxis** (*bool*) – control flag of the axis method
  - True: draws axis normally

  - False: returns the limits and origin position without drawing the axis itself

- **ExtraLenghtAxisR** (*float*) – extra length between the R axis and its label. Default 0.0

Returns

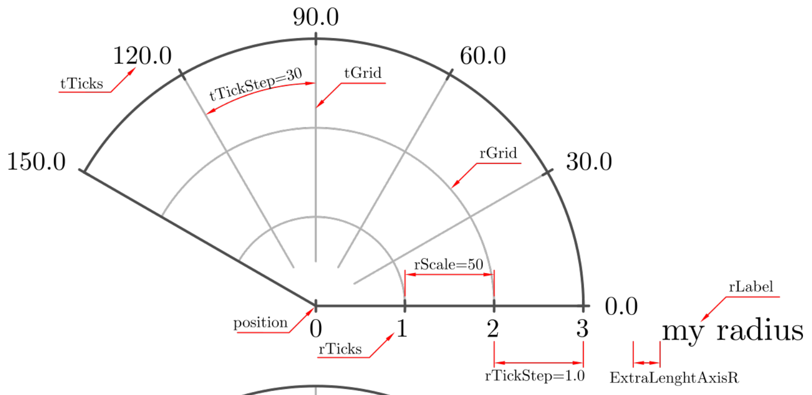
[GroupPlot, outputRLimits, axisOrigin]

- GroupPlot: the axis area object (if drawAxis=False, this output is `None`)
- outputRLimits: a list with tuples:[(r_min,rPos_min),(r_max,rPos_max)]
  - r_min, r_max : The limits of the axis object
  - rPos_min, rPos_max : The positions of the limits of the axis object, considering the scaling and units
  - axisOrigin [X0,Y0] : A list with the coordinates of the point where the axes cross.

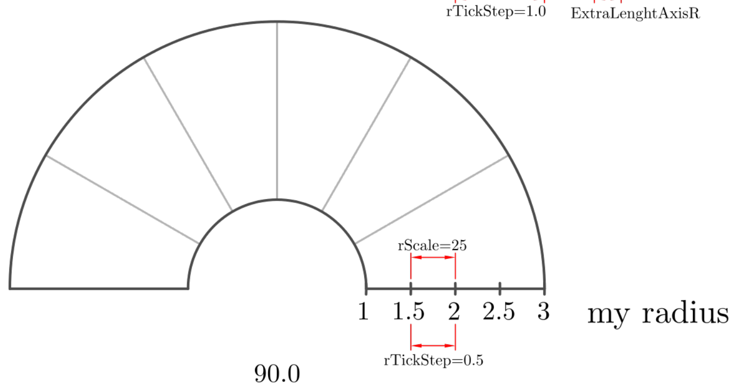
Return type list

Examples

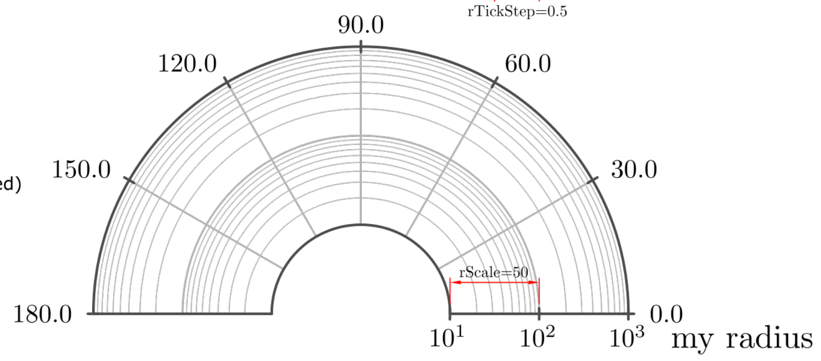
```
rLim=[0,3]
tLim=[0,150]
rlog10scale=False
rTicks=True
tTicks=True
rTickStep=1.0
tTickStep=30
rScale=50
rAxisUnitFactor=''
rGrid=True
tGrid=True
```



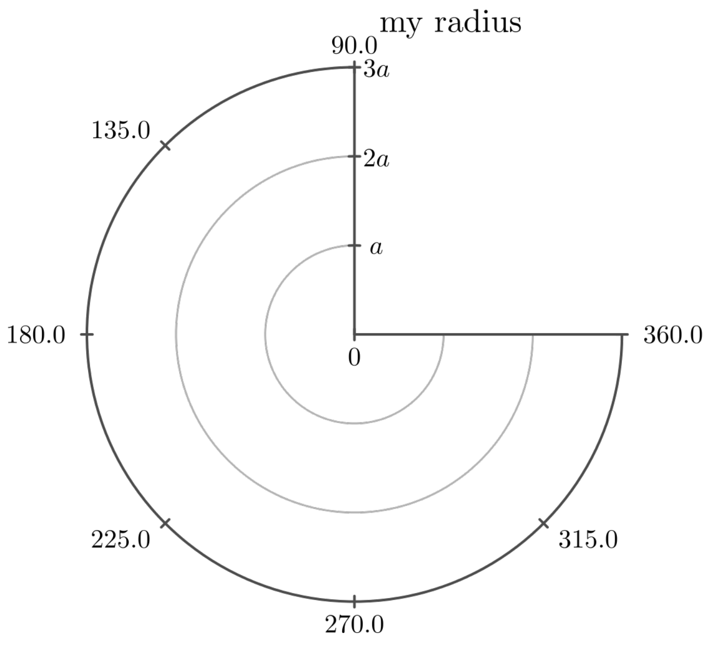
```
rLim=[1,3]
tLim=[0,180]
rlog10scale=False
rTicks=True
tTicks=False
rTickStep=0.5
tTickStep=30
rScale=25
rAxisUnitFactor=''
rGrid=False
tGrid=True
```



```
rLim=[10,1000]
tLim=[0,180]
rlog10scale=True
rTicks=True
tTicks=True
rTickStep=1.0 (not used)
tTickStep=30
rScale=50
rAxisUnitFactor=''
rGrid=True
tGrid=True
```



```
rLim=[0,3]
tLim=[90,360]
rlog10scale=False
rTicks=True
tTicks=True
rTickStep=1.0
tTickStep=45
rScale=50
rAxisUnitFactor='a'
rGrid=True
tGrid=F4alse
```



class `inkscapeMadeEasy_Plot.`**`plot`**

Bases: `object`

This is a class with different methods for making plots.

This class contains only static methods so that you don't have to inherit this in your class

Note: This class uses LaTeX in labels and tick marks if LaTeX support is enabled. This is an optional feature, **enabled by default**. Please refer to *Installation and requirements (all users)* on how to disable it.

static cartesian(*ExtensionBaseObj*, *parent*, *xData*, *yData*, *position=[0, 0]*, *xLabel=''*, *yLabel=''*, *xlog10scale=False*, *ylog10scale=False*, *xTicks=True*, *yTicks=True*, *xTickStep=1.0*, *yTickStep=1.0*, *xScale=20*, *yScale=20*, *xExtraText=''*, *yExtraText=''*, *xGrid=False*, *yGrid=False*, *generalAspectFactorAxis=1.0*, *lineStylePlot={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*, *forceXlim=None*, *forceYlim=None*, *drawAxis=True*, *ExtraLenghtAxisX=0.0*, *ExtraLenghtAxisY=0.0*)

Cartesian Plot

Note: This method uses LaTeX in labels and tick marks if the support is enabled. This is an optional feature, **enabled by default**. Please refer to *Installation and requirements (all users)* on how to disable it.

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **xData** (*list*) – list of x data
- **yData** (*list*) – list of y data
- **position** (*list*) – position of the point where x and y axis cross [x0,y0]. The point where the axis cross depend on the limits.
  - If xLimits comprises the origin x=0, then the Y axis crosses the X axis at x=0.
  - If xLimits contains only negative numbers, then the Y axis crosses the X axis at x_max.
  - If xLimits contains only positive numbers, then the Y axis crosses the X axis at x_min.
- **xLabel** (*string*) – Label of the X axis. Default: ''

  The text can contain any LaTeX command. If you want to write mathematical text, you can enclose it between dollar signs \$...\$. If LaTeX support is disabled, do not use \$.
- **yLabel** (*string*) – Label of the Y axis. Default: ''

  The text can contain any LaTeX command. If you want to write mathematical text, you can enclose it between dollar signs \$...\$. If LaTeX support is disabled, do not use \$.
- **xlog10scale** (*bool*) – sets X axis to log10 scale if True. Default: False
- **ylog10scale** (*bool*) – sets Y axis to log10 scale if True. Default: False
- **xTicks** (*bool*) – Adds axis ticks to the X axis if True. Default: True

- **yTicks** (*bool*) – Adds axis ticks to the Y axis if True. Default: True
- **xTickStep** (*float*) – Value interval between two consecutive ticks on X axis. (Not used if X axis is in log10 scale). Default:1.0
- **yTickStep** (*float*) – Value interval between two consecutive ticks on Y axis. (Not used if Y axis is in log10 scale). Default:1.0
- **xScale** (*float*) – Distance between each xTickStep in svg units. Default: 20
  - If axis is linear, then xScale is the size in svg units of each tick
  - If axis is log10, the xScale is the size in svg units of one decade
- **yScale** (*float*) – Distance between each xTickStep in svg units. Default: 20
  - If axis is linear, then yScale is the size in svg units of each tick
  - If axis is log10, the yScale is the size in svg units of one decade
- **xExtraText** (*string*) – extra text to be added to Ticks in both x and y. Default: ‘’

  This is useful when we want to represent interval with different units. example pi, 2pi 3pi, etc. The text can be any LaTeX text. Keep in mind that this text will be inserted within a mathematical environment $…$, therefore no $ is needed here.
- **yExtraText** (*string*) – extra text to be added to the ticks in Y axis. Default: ‘’

  This is useful when we want to represent interval with different units. example pi, 2pi 3pi, etc. The text can be any LaTeX text. Keep in mind that this text will be inserted within a mathematical environment $…$, therefore no $ is needed here.
- **xGrid** (*bool*) – adds grid lines to X axis if true. Default: False
- **yGrid** (*bool*) – adds grid lines to Y axis if true. Default: False
- **generalAspectFactorAxis** (*float*) – regulates the general aspect ratio between grid lines, text and Ticks separations. Default: 1.0
- **lineStylePlot** (*lineStyle object*) – line style to be used to plot the data. See class `inkscapeMadeEasy_Draw.lineStyle`. Default: lineStylePlot=inkDraw.lineStyle.setSimpleBlack()
- **forceXlim** (*list*) – forces limits of X axis to these limits. These limits affect the axis only, that is, all xData is plotted despite of these limits. Obs: for logarithmic scale, the limits are always adjusted to complete the decade. Usually you don’t need this for logarithmic scale
  - if forceXlim=None Limits will be defined by min and max of xData (Default)
  - if forceXlim=[xMin,xMax] then these limits will be used.
- **forceYlim** (*list*) – forces limits of Y axis to these limits. These limits affect the axis only, that is, all yData is plotted despite of these limits. Obs: for logarithmic scale, the limits are always adjusted to complete the decade. Usually you don’t need this for logarithmic scale
  - if forceYlim=None Limits will be defined by min and max of yData (Default)
  - if forceYlim=[yMin,yMax] then these limits will be used.
- **drawAxis** (*bool*) – control flag of the axis method
  - True: draws axis normally
  - False: returns the limits and origin position without drawing the axis itself

- **ExtraLenghtAxisX** (*float*) – extra length left near the arrow pointer of X axis. Default 0.0
- **ExtraLenghtAxisY** (*float*) – extra length left near the arrow pointer of Y axis. Default 0.0

Returns

[GroupPlot, outputLimits, axisOrigin]

- GroupPlot: the plot object
- outputLimits: a list with tuples:[(x_min,xPos_min),(x_max,xPos_max),(y_min,yPos_min),(y_max,yPos_max)]
  - x_min, x_max, y_min, y_max: The limits of the axis object
  - xPos_min, xPos_max, yPos_min, yPos_max: The positions of the limits of the axis object, considering the scaling and units
  - axisOrigin [X0,Y0]: A list with the coordinates of the point where the axes cross.

Return type list

Note: If any of the axis are log10, then the method ignores any pairs of (x,y) data with invalid coordinates, that is, if xData and/or yData is less than or equal to 0.0 (they would result in complex log10… =P). The method will create a text object alongside your plot warning this.

Note: If any of the axis are linear, the method will ignore any value greater than 10.000 (in absolute value). This avoids plotting too big numbers (basically inf =)). The method will create a text object alongside your plot warning this.

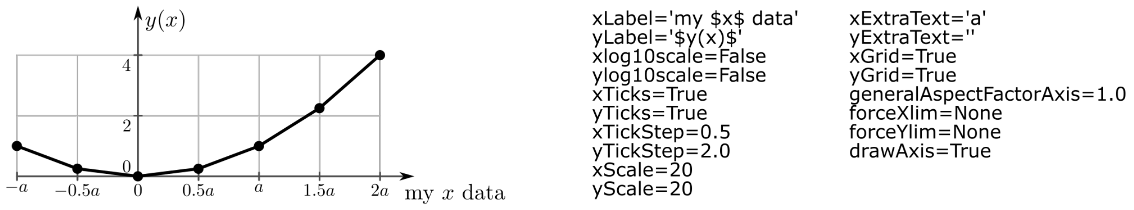
Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>> import inkscapeMadeEasy_Plot as inkPlot
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()      # retrieves the root layer␣
↪of the document
>>>
>>>    xData=[-1,-0.5,0,0.5,1.0,1.5,2]
>>>    yData=[x*x for x in xData]     # computes y=x*x
>>>
>>>    myMarkerDot=inkDraw.marker.createDotMarker(self,'DotM',RenameMode=2,
↪scale=0.3,strokeColor=inkDraw.color.defined('black'),fillColor=inkDraw.
↪color.defined('black'))
>>>    lineStyleDiscrete = inkDraw.lineStyle.set(lineWidth=1.0,␣
↪markerStart=myMarkerDot,markerMid=myMarkerDot,markerEnd=myMarkerDot)
>>>
>>>    inkPlot.plot.cartesian(self,root_layer,xData,yData,position=[0,0],
```

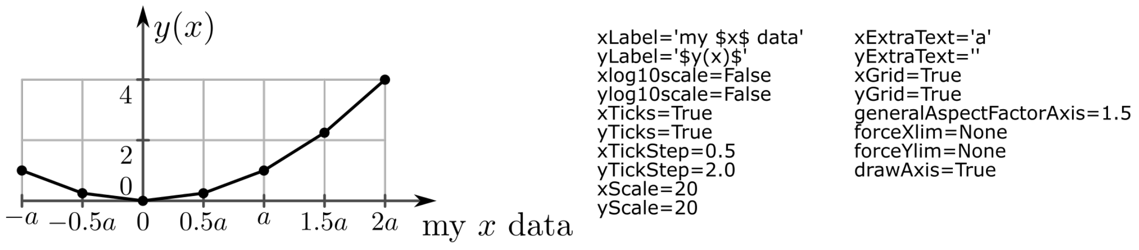
(continues on next page)

(continued from previous page)

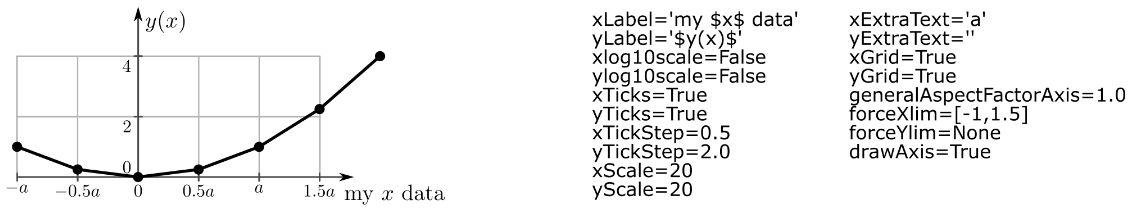
```
>>>                         xLabel='my $x$ data',yLabel='$y(x)$',
↪xlog10scale=False,ylog10scale=False,
>>>                         xTicks=True,yTicks=True,xTickStep=0.5,yTickStep=2.
↪0,
>>>                         xScale=20,yScale=10,xExtraText='a',yExtraText='',
>>>                         xGrid=True,yGrid=True,generalAspectFactorAxis=1.0,
↪lineStylePlot=lineStyleDiscrete,
>>>                         forceXlim=None,forceYlim=None,drawAxis=True)
```



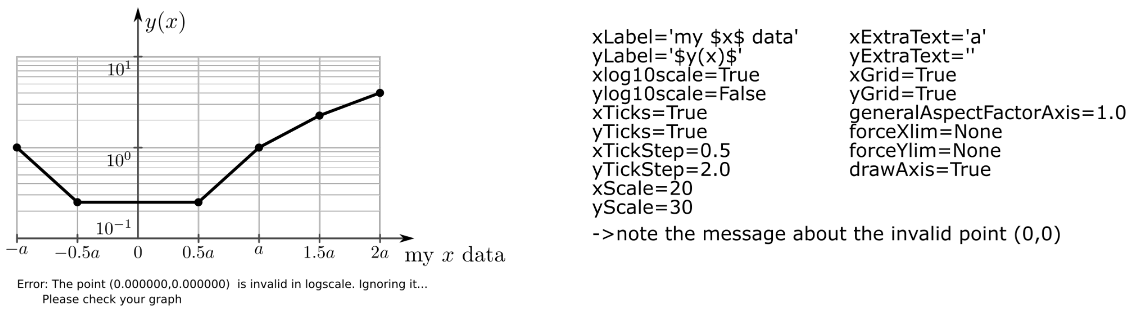
```
xLabel='my $x$ data'       xExtraText='a'
yLabel='$y(x)$'            yExtraText=''
xlog10scale=False          xGrid=True
ylog10scale=False          yGrid=True
xTicks=True                generalAspectFactorAxis=1.0
yTicks=True                forceXlim=None
xTickStep=0.5              forceYlim=None
yTickStep=2.0              drawAxis=True
xScale=20
yScale=20
```



```
xLabel='my $x$ data'       xExtraText='a'
yLabel='$y(x)$'            yExtraText=''
xlog10scale=False          xGrid=True
ylog10scale=False          yGrid=True
xTicks=True                generalAspectFactorAxis=1.5
yTicks=True                forceXlim=None
xTickStep=0.5              forceYlim=None
yTickStep=2.0              drawAxis=True
xScale=20
yScale=20
```



```
xLabel='my $x$ data'       xExtraText='a'
yLabel='$y(x)$'            yExtraText=''
xlog10scale=False          xGrid=True
ylog10scale=False          yGrid=True
xTicks=True                generalAspectFactorAxis=1.0
yTicks=True                forceXlim=[-1,1.5]
xTickStep=0.5              forceYlim=None
yTickStep=2.0              drawAxis=True
xScale=20
yScale=20
```



```
xLabel='my $x$ data'       xExtraText='a'
yLabel='$y(x)$'            yExtraText=''
xlog10scale=True           xGrid=True
ylog10scale=False          yGrid=True
xTicks=True                generalAspectFactorAxis=1.0
yTicks=True                forceXlim=None
xTickStep=0.5              forceYlim=None
yTickStep=2.0              drawAxis=True
xScale=20
yScale=30
->note the message about the invalid point (0,0)
```

```
Error: The point (0.000000,0.000000)  is invalid in logscale. Ignoring it...
      Please check your graph
```

static polar(*ExtensionBaseObj, parent, rData, tData, position=[0, 0], rLabel='', rlog10scale=False, rTicks=True, tTicks=True, rTickStep=1.0, tTickStep=45.0, rScale=20, rExtraText='', rGrid=False, tGrid=False, generalAspectFactorAxis=1.0, lineStylePlot={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}, forceRlim=None, forceTlim=None, drawAxis=True, ExtraLenghtAxisR=0.0*)

Polar Plot

Note: This method uses LaTeX in labels and tick marks if the support is enabled. This is an optional feature, **enabled by default**. Please refer to *Installation and requirements (all users)* on how to disable it.

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **rData** (*list*) – list of R data
- **tData** (*list*) – list of Theta data
- **position** (*list*) – position of the point where x and y axis cross [x0,y0]. The point where the axis cross depend on the limits.
  - If xLimits comprises the origin x=0, then the Y axis crosses the X axis at x=0.
  - If xLimits contains only negative numbers, then the Y axis crosses the X axis at x_max.
  - If xLimits contains only positive numbers, then the Y axis crosses the X axis at x_min.
- **rLabel** (*string*) – Label of the X axis. Default: ''

  The text can contain any LaTeX command. If you want to write mathematical text, you can enclose it between dollar signs $...$. If LaTeX support is disabled, do not use $.
- **rlog10scale** (*bool*) – sets X axis to log10 scale if True. Default: False
- **rTicks** (*bool*) – Adds axis ticks to the X axis if True. Default: True
- **tTicks** (*bool*) – Adds axis ticks to the Y axis if True. Default: True
- **rTickStep** (*float*) – Value interval between two consecutive ticks on X axis. (Not used if X axis is in log10 scale). Default:1.0
- **tTickStep** (*float*) – Value interval between two consecutive ticks on Y axis.
- **rScale** (*float*) – Distance between each rTickStep in svg units. Default: 20
  - If axis is linear, then rScale is the size in svg units of each tick
  - If axis is log10, the rScale is the size in svg units of one decade
- **rExtraText** (*string*) – extra text to be added to Ticks in both x and y. Default: ''

  This is useful when we want to represent interval with different units. example pi, 2pi 3pi, etc. The text can be any LaTeX text. Keep in mind that this text will be inserted within a mathematical environment $...$, therefore no $ is needed here.
- **rGrid** (*bool*) – adds grid lines to X axis if true. Default: False
- **tGrid** (*bool*) – adds grid lines to Y axis if true. Default: False
- **generalAspectFactorAxis** (*float*) – regulates the general aspect ratio between grid lines, text and Ticks separations. Default: 1.0
- **lineStylePlot** (*lineStyle object*) – line style to be used to plot the data. See class `inkscapeMadeEasy_Draw.lineStyle`. Default: lineStylePlot=inkDraw.lineStyle.setSimpleBlack()
- **forceRlim** (*list*) – forces limits of X axis to these limits. These limits affect the axis only, that is, all rData is plotted despite of these limits. Obs: for logarithmic scale, the limits are always adjusted to complete the decade. Usually you don't need this for logarithmic scale
  - if forceRlim=None Limits will be defined by min and max of rData (Default)

  - if forceRlim=[xMin,xMax] then these limits will be used.
- **forceTlim** (*list*) – forces limits of Y axis to these limits. These limits affect the axis only, that is, all tData is plotted despite of these limits. Obs: for logarithmic scale, the limits are always adjusted to complete the decade. Usually you don't need this for logarithmic scale
  - if forceTlim=None Limits will be defined by min and max of tData (Default)
  - if forceTlim=[yMin,yMax] then these limits will be used.
- **drawAxis** (*bool*) – control flag of the axis method
  - True: draws axis normally
  - False: returns the limits and origin position without drawing the axis itself
- **ExtraLenghtAxisR** (*float*) – extra length left near the arrow pointer of X axis. Default 0.0

Returns

[GroupPlot, outputLimits, axisOrigin]

- GroupPlot: the plot object
- outputLimits: a list with tuples:[(x_min,xPos_min),(x_max,xPos_max),(y_min,yPos_min),(y_max,yPos_max)]
  - x_min, x_max, y_min, y_max: The limits of the axis object
  - xPos_min, xPos_max, yPos_min, yPos_max: The positions of the limits of the axis object, considering the scaling and units
  - axisOrigin [X0,Y0]: A list with the coordinates of the point where the axes cross.

Return type list

Note: If any of the axis are log10, then the method ignores any pairs of (x,y) data with invalid coordinates, that is, if rData and/or tData is less than or equal to 0.0 (they would result in complex log10… =P). The method will create a text object alongside your plot warning this.

Note: If any of the axis are linear, the method will ignore any value greater than 10.000 (in absolute value). This avoids plotting too big numbers (basically inf =)). The method will create a text object alongside your plot warning this.

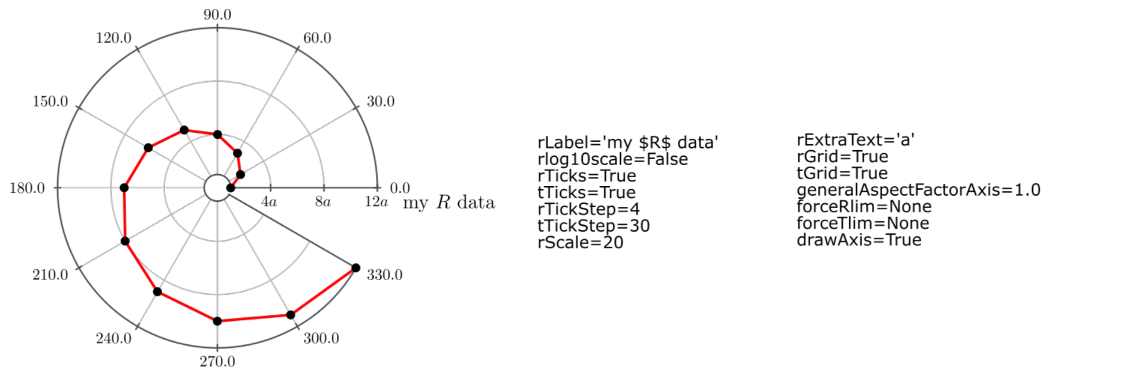
Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>> import inkscapeMadeEasy_Plot as inkPlot
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
```

(continues on next page)

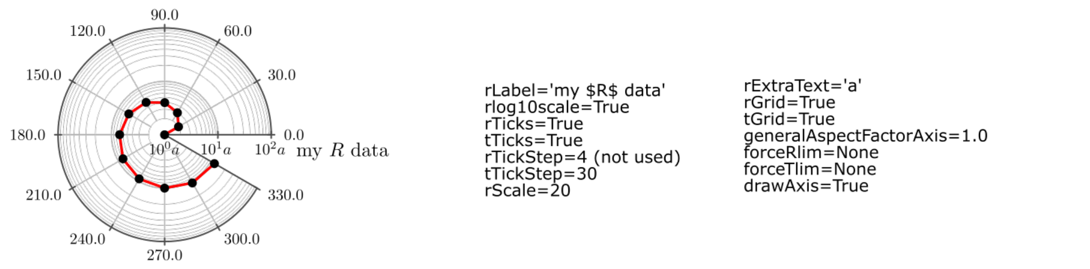
(continued from previous page)

```
>>>
>>>
>>>     rData=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11,12]
>>>     tData=[30*x for x in range(12)]
>>>
>>>     myMarkerDot=inkDraw.marker.createDotMarker(self,'DotM',RenameMode=2,
↪scale=0.3,strokeColor=inkDraw.color.defined('black'),fillColor=inkDraw.
↪color.defined('black'))
>>>     lineStyleDiscrete = inkDraw.lineStyle.set(lineWidth=1.0,
↪linecolor=inkDraw.color.defined('red'), markerStart=myMarkerDot,
↪markerMid=myMarkerDot,markerEnd=myMarkerDot)
>>>
>>>     inkPlot.plot.polar(self,root_layer,rData,tData,position=[0,0],
>>>                       rLabel='my $R$ data',rlog10scale=False,
>>>                       rTicks=True,tTicks=True,rTickStep=2,tTickStep=30,
>>>                       rScale=20,rExtraText='a',
>>>                       rGrid=True,tGrid=True,generalAspectFactorAxis=1.0,
↪lineStylePlot=lineStyleDiscrete,
>>>                       forceRlim=None,forceTlim=None,drawAxis=True)
>>>
>>>     # another spiral, comprising two turns
>>>     tData=[2*x for x in range(360)]
>>>     rData=[x/180.0 for x in tData]
>>>
>>>     inkPlot.plot.polar(self,root_layer,rData,tData,position=[0,0],
>>>                         rLabel='my $R$ data',rlog10scale=False,
>>>                         rTicks=True,tTicks=True,rTickStep=2,tTickStep=30,
>>>                         rScale=20,rExtraText='',
>>>                         rGrid=True,tGrid=True,generalAspectFactorAxis=1.0,
>>>                         forceRlim=None,forceTlim=None,drawAxis=True)
```



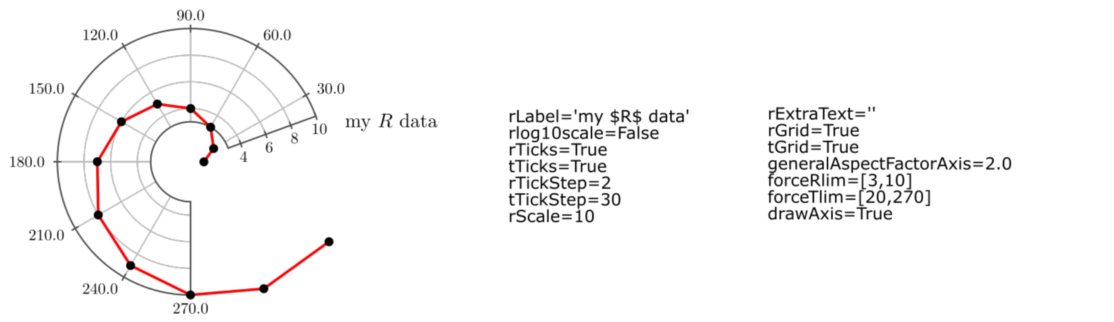
rLabel='my R data'
rlog10scale=False
rTicks=True
tTicks=True
rTickStep=4
tTickStep=30
rScale=20

rExtraText='a'
rGrid=True
tGrid=True
generalAspectFactorAxis=1.0
forceRlim=None
forceTlim=None
drawAxis=True



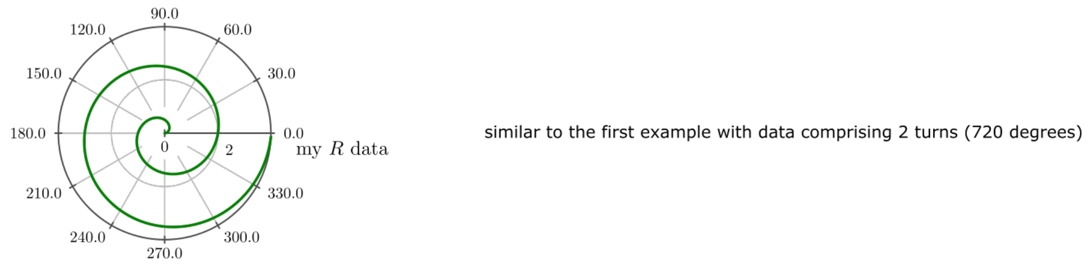
rLabel='my R data'
rlog10scale=True
rTicks=True
tTicks=True
rTickStep=4 (not used)
tTickStep=30
rScale=20

rExtraText='a'
rGrid=True
tGrid=True
generalAspectFactorAxis=1.0
forceRlim=None
forceTlim=None
drawAxis=True



rLabel='my R data'
rlog10scale=False
rTicks=True
tTicks=True
rTickStep=2
tTickStep=30
rScale=10

rExtraText=''
rGrid=True
tGrid=True
generalAspectFactorAxis=2.0
forceRlim=[3,10]
forceTlim=[20,270]
drawAxis=True



similar to the first example with data comprising 2 turns (720 degrees)

static stem(*ExtensionBaseObj*, *parent*, *xData*, *yData*, *position=[0, 0]*, *xLabel=''*, *yLabel=''*, *ylog10scale=False*, *xTicks=True*, *yTicks=True*, *xTickStep=1.0*, *yTickStep=1.0*, *xScale=20*, *yScale=20*, *xExtraText=''*, *yExtraText=''*, *xGrid=False*, *yGrid=False*, *generalAspectFactorAxis=1.0*, *lineStylePlot={'fill': 'none', 'stroke': '#000000', 'stroke-dasharray': 'none', 'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'stroke-width': '1.0'}*, *forceXlim=None*, *forceYlim=None*, *drawAxis=True*, *ExtraLenghtAxisX=0.0*, *ExtraLenghtAxisY=0.0*)

Stem plot in Cartesian axis

Note: This method uses LaTeX in labels and tick marks if the support is enabled. This is an optional feature, **enabled by default**. Please refer to *Installation and requirements (all users)* on how to disable it.

Parameters

- **ExtensionBaseObj** (*inkscapeMadeEasy object (see example below)*) – Most of the times you have to use 'self' from inkscapeMadeEasy related objects
- **parent** (*inkscapeMadeEasy object (see example below)*) – parent object
- **xData** (*list*) – list of x data
- **yData** (*list*) – list of y data
- **position** (*list*) – position of the point where x and y axis cross [x0,y0]. The point where the axis cross depend on the limits.
  - If xLimits comprises the origin x=0, then the Y axis crosses the X axis at x=0.
  - If xLimits contains only negative numbers, then the Y axis crosses the X axis at x_max.
  - If xLimits contains only positive numbers, then the Y axis crosses the X axis at x_min.
- **xLabel** (*string*) – Label of the X axis. Default: ''

  The text can contain any LaTeX command. If you want to write mathematical text, you can enclose it between dollar signs $…$. If LaTeX support is disabled, do not use $.
- **yLabel** (*string*) – Label of the Y axis. Default: ''

  The text can contain any LaTeX command. If you want to write mathematical text, you can enclose it between dollar signs $…$. If LaTeX support is disabled, do not use $.
- **ylog10scale** (*bool*) – sets Y axis to log10 scale if True. Default: False
- **xTicks** (*bool*) – Adds axis ticks to the X axis if True. Default: True
- **yTicks** (*bool*) – Adds axis ticks to the Y axis if True. Default: True
- **xTickStep** (*float*) – Value interval between two consecutive ticks on X axis. (Not used if X axis is in log10 scale). Default:1.0
- **yTickStep** (*float*) – Value interval between two consecutive ticks on Y axis. (Not used if Y axis is in log10 scale). Default:1.0
- **xScale** (*float*) – Distance between each xTickStep in svg units. Default: 20
  - If axis is linear, then xScale is the size in svg units of each tick
  - If axis is log10, the xScale is the size in svg units of one decade
- **yScale** (*float*) – Distance between each xTickStep in svg units. Default: 20
  - If axis is linear, then xScale is the size in svg units of each tick
  - If axis is log10, the xScale is the size in svg units of one decade
- **xExtraText** (*string*) – extra text to be added to Ticks in both x and y. Default: ''

  This is useful when we want to represent interval with different units. example pi, 2pi 3pi, etc. The text can be any LaTeX text. Keep in mind that this text will be inserted within a mathematical environment $…$, therefore no $ is needed here.
- **yExtraText** (*string*) – extra text to be added to the ticks in Y axis. Default: ''

  This is useful when we want to represent interval with different units. example pi, 2pi 3pi, etc. The text can be any LaTeX text. Keep in mind that this text will be inserted within a mathematical environment $…$, therefore no $ is needed here.

- **xGrid** (*bool*) – adds grid lines to X axis if true. Default: False
- **yGrid** (*bool*) – adds grid lines to Y axis if true. Default: False
- **generalAspectFactorAxis** (*float*) – regulates the general aspect ratio between grid lines, text and Ticks separations. Default: 1.0
- **lineStylePlot** (*lineStyle object*) – line style to be used to plot the data. See class `inkscapeMadeEasy_Draw.lineStyle`. Default: lineStylePlot=inkDraw.lineStyle.setSimpleBlack()
- **forceXlim** (*list*) – forces limits of X axis to these limits. These limits affect the axis only, that is, all xData is plotted despite of these limits. Obs: for logarithmic scale, the limits are always adjusted to complete the decade. Usually you don't need this for logarithmic scale
  - if forceXlim=None Limits will be defined by min and max of xData (Default)
  - if forceXlim=[xMin,xMax] then these limits will be used.
- **forceYlim** (*list*) – forces limits of Y axis to these limits. These limits affect the axis only, that is, all yData is plotted despite of these limits. Obs: for logarithmic scale, the limits are always adjusted to complete the decade. Usually you don't need this for logarithmic scale
  - if forceYlim=None Limits will be defined by min and max of yData (Default)
  - if forceYlim=[yMin,yMax] then these limits will be used.
- **drawAxis** (*bool*) – control flag of the axis method
  - True: draws axis normally
  - False: returns the limits and origin position without drawing the axis itself
- **ExtraLenghtAxisX** (*float*) – extra length left near the arrow pointer of X axis. Default 0.0
- **ExtraLenghtAxisY** (*float*) – extra length left near the arrow pointer of Y axis. Default 0.0

Returns

[GroupPlot, outputLimits, axisOrigin]

- GroupPlot: the plot object
- outputLimits: a list with tuples:[(x_min,xPos_min),(x_max,xPos_max),(y_min,yPos_min),(y_max,yPos_max)]
  - x_min, x_max, y_min, y_max: The limits of the axis object
  - xPos_min, xPos_max, yPos_min, yPos_max: The positions of the limits of the axis object, considering the scaling and units
  - axisOrigin [X0,Y0]: A list with the coordinates of the point where the axes cross.

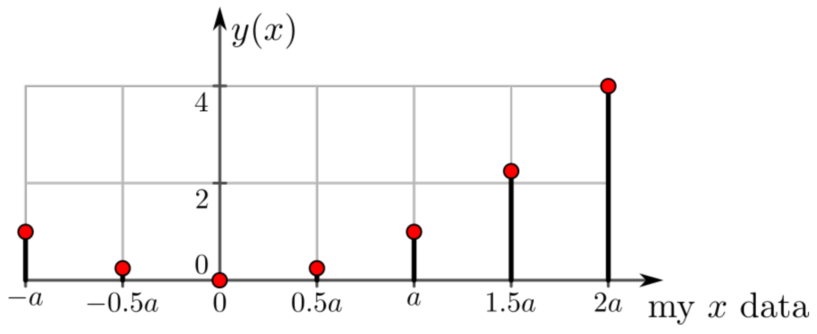
Return type list

Note: If any of the axis are log10, then the method ignores any pairs of (x,y) data with invalid coordinates, that is, if xData and/or yData is less than or equal to 0.0 (they would result in complex log10… =P). The method will create a text object alongside your plot warning this.

Note: If any of the axis are linear, the method will ignore any value greater than 10.000. this avoids plotting too big numbers (basically inf =)). The method will create a text object alongside your plot warning this.

Example

```
>>> import inkex
>>> import inkscapeMadeEasy_Base as inkBase
>>> import inkscapeMadeEasy_Draw as inkDraw
>>> import inkscapeMadeEasy_Plot as inkPlot
>>>
>>> class myExtension(inkBase.inkscapeMadeEasy):
>>>   def __init__(self):
>>>     ...
>>>     ...
>>>
>>>   def effect(self):
>>>     root_layer = self.document.getroot()     # retrieves the root layer␣
↪of the document
>>>     myLineStyle=inkDraw.lineStyle.setSimpleBlack()
>>>
>>>    xData=[-1,-0.5,0,0.5,1.0,1.5,2]
>>>    yData=[x*x for x in xData]    # computes y=x*x
>>>
>>>    # creates a line style with a dot marker for the stem plot
>>>    myMarkerDot=inkDraw.marker.createDotMarker(self,'DotMDiscreteTime',
↪RenameMode=2,scale=0.3,strokeColor=inkDraw.color.defined('black'),
↪fillColor=inkDraw.color.defined('red'))
>>>    lineStyleDiscrete = inkDraw.lineStyle.set(lineWidth=1.0,␣
↪markerEnd=myMarkerDot)
>>>
>>>    inkPlot.plot.stem(self,root_layer,xData,yData,position=[0,0],
>>>                      xLabel='my $x$ data',yLabel='$y(x)$',
↪ylog10scale=False,
>>>                      xTicks=True,yTicks=True,xTickStep=0.5,yTickStep=2.
↪0,
>>>                      xScale=20,yScale=20,xExtraText='a',yExtraText='',
>>>                      xGrid=True,yGrid=True,generalAspectFactorAxis=1.0,
↪lineStylePlot=lineStyleDiscrete,
>>>                      forceXlim=None,forceYlim=None,drawAxis=True)
```



7.4 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX