

API Structure: matplotlib

matplotlib	54
matplotlib.ExecutableNotFoundError	55
matplotlib.MatplotlibDeprecationWarning	56
matplotlib.RcParams	56
matplotlib.RcParams.copy	62
matplotlib.RcParams.find_all	62
matplotlib.animation	62
matplotlib.animation.AbstractMovieWriter	62
matplotlib.animation.Animation	63
matplotlib.animation.ArtistAnimation	63
matplotlib.animation.FFMpegBase	64
matplotlib.animation.FFMpegFileWriter	64
matplotlib.animation.FFMpegWriter	64
matplotlib.animation.FileMovieWriter	65
matplotlib.animation.FuncAnimation	65
matplotlib.animation.HTMLWriter	67
matplotlib.animation.ImageMagickBase	67
matplotlib.animation.ImageMagickFileWriter	67
matplotlib.animation.ImageMagickWriter	67
matplotlib.animation.MovieWriter	67
matplotlib.animation.MovieWriterRegistry	68
matplotlib.animation.PillowWriter	68
matplotlib.animation.TimedAnimation	68
matplotlib.animation.adjusted_figsize	69

matplotlib.artist	69
matplotlib.artist.Artist	69
matplotlib.artist.ArtistInspector	69
matplotlib.artist.Bbox	69
matplotlib.artist.BboxBase	71
matplotlib.artist.IdentityTransform	71
matplotlib.artist.Path	71
matplotlib.artist.Transform	72
matplotlib.artist.TransformedBbox	73
matplotlib.artist.TransformedPatchPath	73
matplotlib.artist.TransformedPath	73
matplotlib.artist.allow_rasterization	73
matplotlib.artist.get	74
matplotlib.artist.getp	74
matplotlib.artist.kwdoc	75
matplotlib.artist.setp	75
matplotlib.axes	76
matplotlib.axes.Axes	76
matplotlib.axes.Subplot	77
matplotlib.axes.SubplotBase	77
matplotlib.axes.subplot_class_factory	77
matplotlib.axis	77
matplotlib.axis.Axis	77
matplotlib.axis.Tick	78
matplotlib.axis.Ticker	79
matplotlib.axis.XAxis	79

matplotlib.axis.XTick	80
matplotlib.axis.YAxis	80
matplotlib.axis.YTick	81
matplotlib.backend_bases	81
matplotlib.backend_bases.Affine2D	81
matplotlib.backend_bases.CapStyle	82
matplotlib.backend_bases.CloseEvent	82
matplotlib.backend_bases.ConstrainedLayoutEngine	82
matplotlib.backend_bases.DrawEvent	82
matplotlib.backend_bases.Event	83
matplotlib.backend_bases.FigureCanvasBase	83
matplotlib.backend_bases.FigureManagerBase	83
matplotlib.backend_bases.Gcf	85
matplotlib.backend_bases.GraphicsContextBase	85
matplotlib.backend_bases.JoinStyle	85
matplotlib.backend_bases.KeyEvent	86
matplotlib.backend_bases.LocationEvent	87
matplotlib.backend_bases.MouseButton	87
matplotlib.backend_bases.MouseEvent	87
matplotlib.backend_bases.NavigationToolbar2	89
matplotlib.backend_bases.NonGuiException	89
matplotlib.backend_bases.Path	89
matplotlib.backend_bases.PickEvent	90
matplotlib.backend_bases.RendererBase	91
matplotlib.backend_bases.ResizeEvent	92

matplotlib.backend_bases.ShowBase	92
matplotlib.backend_bases.TexManager	92
matplotlib.backend_bases.TimerBase	92
matplotlib.backend_bases.ToolContainerBase	93
matplotlib.backend_bases.ToolManager	93
matplotlib.backend_bases.button_press_handler	93
matplotlib.backend_bases.cursors	93
matplotlib.backend_bases.get_registered_canvas_class	93
matplotlib.backend_bases.is_interactive	94
matplotlib.backend_bases.key_press_handler	94
matplotlib.backend_bases.register_backend	94
matplotlib.backend_managers	94
matplotlib.backend_managers.ToolEvent	95
matplotlib.backend_managers.ToolManager	95
matplotlib.backend_managers.ToolManagerMessageEvent	95
matplotlib.backend_managers.ToolTriggerEvent	95
matplotlib.backend_tools	95
matplotlib.backend_tools.AxisScaleBase	95
matplotlib.backend_tools.ConfigureSubplotsBase	96
matplotlib.backend_tools.Cursors	96
matplotlib.backend_tools.Gcf	96
matplotlib.backend_tools.RubberbandBase	96
matplotlib.backend_tools.SaveFigureBase	96
matplotlib.backend_tools.ToolBack	96
matplotlib.backend_tools.ToolBase	96
matplotlib.backend_tools.ToolCopyToClipboardBase	97

matplotlib.backend_tools.ToolCursorPosition	97
matplotlib.backend_tools.ToolForward	97
matplotlib.backend_tools.ToolFullScreen	97
matplotlib.backend_tools.ToolGrid	97
matplotlib.backend_tools.ToolHelpBase	97
matplotlib.backend_tools.ToolHome	97
matplotlib.backend_tools.ToolMinorGrid	98
matplotlib.backend_tools.ToolPan	98
matplotlib.backend_tools.ToolQuit	98
matplotlib.backend_tools.ToolQuitAll	98
matplotlib.backend_tools.ToolSetCursor	98
matplotlib.backend_tools.ToolToggleBase	98
matplotlib.backend_tools.ToolViewsPositions	98
matplotlib.backend_tools.ToolXScale	99
matplotlib.backend_tools.ToolYScale	99
matplotlib.backend_tools.ToolZoom	99
matplotlib.backend_tools.ViewsPositionsBase	99
matplotlib.backend_tools.ZoomPanBase	99
matplotlib.backend_tools.add_tools_to_container	99
matplotlib.backend_tools.add_tools_to_manager	100
matplotlib.backend_tools.cursors	100
matplotlib.backends	100
matplotlib.backends.BackendFilter	100
matplotlib.backends.backend_agg	100
matplotlib.backends.backend_cairo	101

matplotlib.backends.backend_mixed	101
matplotlib.backends.backend_nbagg	101
matplotlib.backends.backend_pdf	101
matplotlib.backends.backend_pgf	101
matplotlib.backends.backend_ps	101
matplotlib.backends.backend_qt	101
matplotlib.backends.backend_qt5	102
matplotlib.backends.backend_qt5agg	102
matplotlib.backends.backend_qt5cairo	102
matplotlib.backends.backend_qtagg	102
matplotlib.backends.backend_qtcairo	102
matplotlib.backends.backend_svg	102
matplotlib.backends.backend_template	102
matplotlib.backends.backend_tkagg	103
matplotlib.backends.backend_tkcairo	103
matplotlib.backends.backend_webagg	103
matplotlib.backends.backend_webagg_core	103
matplotlib.backends.qt_compat	103
matplotlib.backends.qt_editor	104
matplotlib.backends.registry	104
matplotlib.bezier	104
matplotlib.bezier.BezierSegment	104
matplotlib.bezier.NonIntersectingPathException	104
matplotlib.bezier.check_if_parallel	104
matplotlib.bezier.find_bezier_t_intersecting_with_closedpath	105
matplotlib.bezier.find_control_points	105

matplotlib.bezier.get_cos_sin	105
matplotlib.bezier.get_intersection	105
matplotlib.bezier.get_normal_points	106
matplotlib.bezier.get_parallels	106
matplotlib.bezier.inside_circle	106
matplotlib.bezier.make_wedged_bezier2	106
matplotlib.bezier.split_bezier_intersecting_with_closedpath	106
matplotlib.bezier.split_de_casteljau	107
matplotlib.bezier.split_path_inout	107
matplotlib.category	107
matplotlib.category.StrCategoryConverter	107
matplotlib.category.StrCategoryFormatter	107
matplotlib.category.StrCategoryLocator	107
matplotlib.category.UnitData	107
matplotlib.cbook	108
matplotlib.cbook.CallbackRegistry	108
matplotlib.cbook.Grouper	109
matplotlib.cbook.GrouperView	109
matplotlib.cbook.boxplot_stats	110
matplotlib.cbook.contiguous_regions	111
matplotlib.cbook.delete_masked_points	111
matplotlib.cbook.file_requires_unicode	112
matplotlib.cbook.flatten	112
matplotlib.cbook.get_sample_data	112
matplotlib.cbook.index_of	112

matplotlib.cbook.is_math_text	113
matplotlib.cbook.is_scalar_or_string	113
matplotlib.cbook.is_writable_file_like	113
matplotlib.cbook.normalize_kwargs	113
matplotlib.cbook.open_file_cm	114
matplotlib.cbook.print_cycles	114
matplotlib.cbook.pts_to_midstep	114
matplotlib.cbook.pts_to_poststep	115
matplotlib.cbook.pts_to_prestep	115
matplotlib.cbook.safe_first_element	116
matplotlib.cbook.safe_masked_invalid	116
matplotlib.cbook.sanitize_sequence	116
matplotlib.cbook.silent_list	116
matplotlib.cbook.simple_linear_interpolation	117
matplotlib.cbook.strip_math	117
matplotlib.cbook.to_filehandle	117
matplotlib.cbook.violin_stats	118
matplotlib.cm	119
matplotlib.cm.ColormapRegistry	119
matplotlib.cm.ScalarMappable	120
matplotlib.cm.get_cmap	120
matplotlib.collections	120
matplotlib.collections.AsteriskPolygonCollection	120
matplotlib.collections.CapStyle	121
matplotlib.collections.CircleCollection	121
matplotlib.collections.Collection	121

matplotlib.collections.EllipseCollection	122
matplotlib.collections.EventCollection	122
matplotlib.collections.FillBetweenPolyCollection	122
matplotlib.collections.JoinStyle	122
matplotlib.collections.LineCollection	123
matplotlib.collections.PatchCollection	124
matplotlib.collections.PathCollection	124
matplotlib.collections.PolyCollection	124
matplotlib.collections.PolyQuadMesh	124
matplotlib.collections.QuadMesh	125
matplotlib.collections.RegularPolyCollection	125
matplotlib.collections.StarPolygonCollection	126
matplotlib.collections.TriMesh	126
matplotlib.colorbar	126
matplotlib.colorbar.Colorbar	126
matplotlib.colorbar.ColorbarBase	127
matplotlib.colorbar.make_axes	127
matplotlib.colorbar.make_axes_gridspec	128
matplotlib.colorizer	129
matplotlib.colorizer.Colorizer	130
matplotlib.colorizer.ColorizingArtist	130
matplotlib.colors	130
matplotlib.colors.AsinhNorm	131
matplotlib.colors.BivarColormap	131
matplotlib.colors.BivarColormapFromImage	132

matplotlib.colors.BoundaryNorm	132
matplotlib.colors.CenteredNorm	132
matplotlib.colors.ColorConverter	133
matplotlib.colors.ColorSequenceRegistry	133
matplotlib.colors.Colormap	133
matplotlib.colors.FuncNorm	133
matplotlib.colors.LightSource	134
matplotlib.colors.LinearSegmentedColormap	134
matplotlib.colors.ListedColormap	134
matplotlib.colors.LogNorm	135
matplotlib.colors.MultivarColormap	135
matplotlib.colors.NoNorm	135
matplotlib.colors.Normalize	135
matplotlib.colors.PowerNorm	136
matplotlib.colors.SegmentedBivarColormap	137
matplotlib.colors.SymLogNorm	137
matplotlib.colors.TwoSlopeNorm	138
matplotlib.colors.from_levels_and_colors	138
matplotlib.colors.get_named_colors_mapping	138
matplotlib.colors.hex2color	139
matplotlib.colors.hsv_to_rgb	139
matplotlib.colors.is_color_like	139
matplotlib.colors.make_norm_from_scale	139
matplotlib.colors.rgb2hex	140
matplotlib.colors.rgb_to_hsv	140
matplotlib.colors.same_color	140

matplotlib.colors.to_hex	140
matplotlib.colors.to_rgb	141
matplotlib.colors.to_rgba	141
matplotlib.colors.to_rgba_array	141
matplotlib.container	142
matplotlib.container.Artist	142
matplotlib.container.BarContainer	142
matplotlib.container.Container	143
matplotlib.container.ErrorbarContainer	143
matplotlib.container.StemContainer	143
matplotlib.contour	144
matplotlib.contour.ContourLabeler	144
matplotlib.contour.ContourSet	144
matplotlib.contour.Line2D	145
matplotlib.contour.MouseButton	145
matplotlib.contour.Path	145
matplotlib.contour.QuadContourSet	146
matplotlib.contour.Text	147
matplotlib.cycler	147
matplotlib.dates	148
matplotlib.dates.AutoDateFormatter	151
matplotlib.dates.AutoDateLocator	152
matplotlib.dates.ConciseDateConverter	153
matplotlib.dates.ConciseDateFormatter	153
matplotlib.dates.DateConverter	154

matplotlib.dates.DateFormatter	154
matplotlib.dates.DateLocator	155
matplotlib.dates.DayLocator	155
matplotlib.dates.HourLocator	155
matplotlib.dates.MicrosecondLocator	155
matplotlib.dates.MinuteLocator	155
matplotlib.dates.MonthLocator	155
matplotlib.dates.RRRuleLocator	156
matplotlib.dates.SecondLocator	156
matplotlib.dates.WeekdayLocator	156
matplotlib.dates.YearLocator	156
matplotlib.dates.date2num	156
matplotlib.dates.datestr2num	157
matplotlib.dates.drange	157
matplotlib.dates.get_epoch	157
matplotlib.dates.num2date	157
matplotlib.dates.num2timedelta	158
matplotlib.dates.rrulewrapper	158
matplotlib.dates.set_epoch	158
matplotlib.dviread	159
matplotlib.dviread.Box	159
matplotlib.dviread.Dvi	159
matplotlib.dviread.DviFont	160
matplotlib.dviread.Page	160
matplotlib.dviread.PsFont	160
matplotlib.dviread.PsfontsMap	161

matplotlib.dviread.Text	161
matplotlib.dviread.Tfm	162
matplotlib.dviread.Vf	162
matplotlib.figure	163
matplotlib.figure.Affine2D	163
matplotlib.figure.Artist	163
matplotlib.figure.Axes	163
matplotlib.figure.Bbox	164
matplotlib.figure.BboxTransformTo	165
matplotlib.figure.ConstrainedLayoutEngine	165
matplotlib.figure.DrawEvent	166
matplotlib.figure.Figure	166
matplotlib.figure.FigureBase	166
matplotlib.figure.FigureCanvasBase	166
matplotlib.figure.GridSpec	167
matplotlib.figure.LayoutEngine	167
matplotlib.figure.MouseButton	167
matplotlib.figure.NonGuiException	168
matplotlib.figure.PlaceHolderLayoutEngine	168
matplotlib.figure.Rectangle	168
matplotlib.figure.SubFigure	168
matplotlib.figure.SubplotParams	169
matplotlib.figure.Text	169
matplotlib.figure.TightLayoutEngine	169
matplotlib.figure.TransformedBbox	169

matplotlib.figure.allow_rasterization	169
matplotlib.figure.figspect	169
matplotlib.font_manager	170
matplotlib.font_manager.FontEntry	171
matplotlib.font_manager.FontManager	171
matplotlib.font_manager.FontProperties	171
matplotlib.font_manager.afmFontProperty	173
matplotlib.font_manager.findSystemFonts	173
matplotlib.font_manager.generate_fontconfig_pattern	173
matplotlib.font_manager.get_font	173
matplotlib.font_manager.get_fonttext_synonyms	174
matplotlib.font_manager.json_dump	174
matplotlib.font_manager.json_load	174
matplotlib.font_manager.list_fonts	174
matplotlib.font_manager.ttfFontProperty	175
matplotlib.font_manager.win32FontDirectory	175
matplotlib.ft2font	175
matplotlib.ft2font.FT2Font	175
matplotlib.ft2font.FT2Image	176
matplotlib.ft2font.FaceFlags	176
matplotlib.ft2font.Glyph	176
matplotlib.ft2font.Kerning	176
matplotlib.ft2font.LoadFlags	176
matplotlib.ft2font.StyleFlags	177
matplotlib.get_backend	177
matplotlib.get_cachedir	177

matplotlib.get_configdir	177
matplotlib.get_data_path	178
matplotlib.gridspec	178
matplotlib.gridspec.Bbox	178
matplotlib.gridspec.GridSpec	180
matplotlib.gridspec.GridSpecBase	180
matplotlib.gridspec.GridSpecFromSubplotSpec	180
matplotlib.gridspec.SubplotParams	180
matplotlib.gridspec.SubplotSpec	180
matplotlib.hatch	181
matplotlib.hatch.Circles	181
matplotlib.hatch.HatchPatternBase	181
matplotlib.hatch.HorizontalHatch	181
matplotlib.hatch.LargeCircles	181
matplotlib.hatch.NorthEastHatch	181
matplotlib.hatch.Path	181
matplotlib.hatch.Shapes	182
matplotlib.hatch.SmallCircles	182
matplotlib.hatch.SmallFilledCircles	182
matplotlib.hatch.SouthEastHatch	183
matplotlib.hatch.Stars	183
matplotlib.hatch.VerticalHatch	183
matplotlib.hatch.get_path	183
matplotlib.image	183
matplotlib.image.Affine2D	183

matplotlib.image.AxesImage	183
matplotlib.image.Bbox	184
matplotlib.image.BboxBase	186
matplotlib.image.BboxImage	186
matplotlib.image.BboxTransform	186
matplotlib.image.BboxTransformTo	186
matplotlib.image.FigureCanvasBase	186
matplotlib.image.FigureImage	187
matplotlib.image.IdentityTransform	187
matplotlib.image.NonUniformImage	187
matplotlib.image.PcolorImage	187
matplotlib.image.TransformedBbox	187
matplotlib.image.composite_images	187
matplotlib.image.imread	188
matplotlib.image.imsave	189
matplotlib.image.pil_to_array	190
matplotlib.image.thumbnail	190
matplotlib.inset	191
matplotlib.inset.ConnectionPatch	191
matplotlib.inset.InsetIndicator	191
matplotlib.inset.Path	191
matplotlib.inset.PathPatch	192
matplotlib.inset.Rectangle	192
matplotlib.interactive	193
matplotlib.is_interactive	193
matplotlib.layout_engine	193

matplotlib.layout_engine.ConstrainedLayoutEngine	193
matplotlib.layout_engine.LayoutEngine	193
matplotlib.layout_engine.PlaceHolderLayoutEngine	194
matplotlib.layout_engine.TightLayoutEngine	194
matplotlib.layout_engine.do_constrained_layout	194
matplotlib.layout_engine.get_subplotspec_list	195
matplotlib.layout_engine.get_tight_layout_figure	195
matplotlib.legend	196
matplotlib.legend.AnchoredOffsetbox	196
matplotlib.legend.Artist	197
matplotlib.legend.BarContainer	197
matplotlib.legend.Bbox	197
matplotlib.legend.BboxBase	199
matplotlib.legend.BboxTransformFrom	199
matplotlib.legend.BboxTransformTo	199
matplotlib.legend.CircleCollection	199
matplotlib.legend.Collection	199
matplotlib.legend.DraggableLegend	200
matplotlib.legend.DraggableOffsetBox	200
matplotlib.legend.DrawingArea	201
matplotlib.legend.ErrorbarContainer	201
matplotlib.legend.FancyBboxPatch	202
matplotlib.legend.FontProperties	202
matplotlib.legend.HPacker	203
matplotlib.legend.Legend	203

matplotlib.legend.Line2D	204
matplotlib.legend.LineCollection	204
matplotlib.legend.Patch	204
matplotlib.legend.PathCollection	204
matplotlib.legend.PolyCollection	204
matplotlib.legend.Rectangle	205
matplotlib.legend.RegularPolyCollection	205
matplotlib.legend.Shadow	205
matplotlib.legend.StemContainer	205
matplotlib.legend.StepPatch	206
matplotlib.legend.Text	206
matplotlib.legend.TextArea	206
matplotlib.legend.TransformedBbox	206
matplotlib.legend.VPacker	206
matplotlib.legend.allow_rasterization	206
matplotlib.legend.silent_list	207
matplotlib.legend_handler	207
matplotlib.legend_handler.HandlerBase	208
matplotlib.legend_handler.HandlerCircleCollection	208
matplotlib.legend_handler.HandlerErrorbar	208
matplotlib.legend_handler.HandlerLine2D	208
matplotlib.legend_handler.HandlerLine2DCompound	208
matplotlib.legend_handler.HandlerLineCollection	208
matplotlib.legend_handler.HandlerNpoints	209
matplotlib.legend_handler.HandlerNpointsYoffsets	209
matplotlib.legend_handler.HandlerPatch	209

matplotlib.legend_handler.HandlerPathCollection	209
matplotlib.legend_handler.HandlerPolyCollection	209
matplotlib.legend_handler.HandlerRegularPolyCollection	209
matplotlib.legend_handler.HandlerStem	209
matplotlib.legend_handler.HandlerStepPatch	209
matplotlib.legend_handler.HandlerTuple	210
matplotlib.legend_handler.Line2D	210
matplotlib.legend_handler.Rectangle	210
matplotlib.legend_handler.update_from_first_child	210
matplotlib.lines	210
matplotlib.lines.Artist	210
matplotlib.lines.AxLine	211
matplotlib.lines.Bbox	211
matplotlib.lines.BboxTransformTo	212
matplotlib.lines.CapStyle	212
matplotlib.lines.JoinStyle	213
matplotlib.lines.Line2D	214
matplotlib.lines.MarkerStyle	214
matplotlib.lines.Path	215
matplotlib.lines.TransformedPath	216
matplotlib.lines.VertexSelector	216
matplotlib.lines.allow_rasterization	216
matplotlib.lines.segment_hits	217
matplotlib.markers	217
matplotlib.markers.Affine2D	219

matplotlib.markers.CapStyle	219
matplotlib.markers.IdentityTransform	220
matplotlib.markers.JoinStyle	220
matplotlib.markers.MarkerStyle	221
matplotlib.markers.Path	221
matplotlib.mathtext	222
matplotlib.mathtext.FontProperties	223
matplotlib.mathtext.LoadFlags	224
matplotlib.mathtext.MathTextParser	224
matplotlib.mathtext.RasterParse	224
matplotlib.mathtext.VectorParse	225
matplotlib.mathtext.get_unicode_index	225
matplotlib.mathtext.math_to_image	225
matplotlib.matplotlib_fname	226
matplotlib.mlab	226
matplotlib.mlab.GaussianKDE	227
matplotlib.mlab.cohere	228
matplotlib.mlab.csd	229
matplotlib.mlab.detrend	231
matplotlib.mlab.detrend_linear	231
matplotlib.mlab.detrend_mean	232
matplotlib.mlab.detrend_none	232
matplotlib.mlab.psd	232
matplotlib.mlab.specgram	234
matplotlib.mlab.window_hanning	236
matplotlib.mlab.window_none	236

matplotlib.offsetbox	236
matplotlib.offsetbox.AnchoredOffsetbox	237
matplotlib.offsetbox.AnchoredText	237
matplotlib.offsetbox.AnnotationBbox	237
matplotlib.offsetbox.AuxTransformBox	237
matplotlib.offsetbox.Bbox	238
matplotlib.offsetbox.BboxBase	239
matplotlib.offsetbox.BboxImage	239
matplotlib.offsetbox.DraggableAnnotation	240
matplotlib.offsetbox.DraggableBase	240
matplotlib.offsetbox.DraggableOffsetBox	241
matplotlib.offsetbox.DrawingArea	241
matplotlib.offsetbox.FancyArrowPatch	241
matplotlib.offsetbox.FancyBboxPatch	242
matplotlib.offsetbox.FontProperties	242
matplotlib.offsetbox.HPacker	243
matplotlib.offsetbox.OffsetBox	243
matplotlib.offsetbox.OffsetImage	244
matplotlib.offsetbox.PackerBase	244
matplotlib.offsetbox.PaddedBox	244
matplotlib.offsetbox.TextArea	245
matplotlib.offsetbox.TransformedBbox	245
matplotlib.offsetbox.VPacker	245
matplotlib.offsetbox.mbbox_artist	245
matplotlib.patches	245

matplotlib.patches.Affine2D	245
matplotlib.patches.Annulus	246
matplotlib.patches.Arc	246
matplotlib.patches.Arrow	246
matplotlib.patches.ArrowStyle	246
matplotlib.patches.BoxStyle	247
matplotlib.patches.CapStyle	248
matplotlib.patches.Circle	249
matplotlib.patches.CirclePolygon	249
matplotlib.patches.ConnectionPatch	249
matplotlib.patches.ConnectionStyle	249
matplotlib.patches.Ellipse	250
matplotlib.patches.FancyArrow	250
matplotlib.patches.FancyArrowPatch	250
matplotlib.patches.FancyBboxPatch	251
matplotlib.patches.JoinStyle	251
matplotlib.patches.NonIntersectingPathException	252
matplotlib.patches.Patch	252
matplotlib.patches.Path	252
matplotlib.patches.PathPatch	253
matplotlib.patches.Polygon	253
matplotlib.patches.Rectangle	253
matplotlib.patches.RegularPolygon	254
matplotlib.patches.Shadow	254
matplotlib.patches.StepPatch	254
matplotlib.patches.Wedge	254

matplotlib.patches.bbox_artist	254
matplotlib.patches.draw_bbox	254
matplotlib.patches.get_cos_sin	255
matplotlib.patches.get_intersection	255
matplotlib.patches.get_parallel	255
matplotlib.patches.inside_circle	255
matplotlib.patches.make_wedged_bezier2	255
matplotlib.patches.split_bezier_intersecting_with_closedpath	255
matplotlib.patches.split_path_inout	256
matplotlib.path	256
matplotlib.path.BezierSegment	256
matplotlib.path.Path	256
matplotlib.path.get_path_collection_extents	257
matplotlib.path.simple_linear_interpolation	258
matplotlib.path_effects	258
matplotlib.path_effects.AbstractPathEffect	258
matplotlib.path_effects.Normal	259
matplotlib.path_effects.Path	259
matplotlib.path_effects.PathEffectRenderer	260
matplotlib.path_effects.PathPatchEffect	260
matplotlib.path_effects.RendererBase	260
matplotlib.path_effects.SimpleLineShadow	261
matplotlib.path_effects.SimplePatchShadow	261
matplotlib.path_effects.Stroke	261
matplotlib.path_effects.TickedStroke	261

matplotlib.patheffects.withSimplePatchShadow	261
matplotlib.patheffects.withStroke	262
matplotlib.patheffects.withTickedStroke	262
matplotlib.projections	262
matplotlib.projections.AitoffAxes	263
matplotlib.projections.HammerAxes	263
matplotlib.projections.LambertAxes	263
matplotlib.projections.MollweideAxes	264
matplotlib.projections.PolarAxes	264
matplotlib.projections.ProjectionRegistry	264
matplotlib.projections.geo	264
matplotlib.projections.get_projection_class	264
matplotlib.projections.polar	264
matplotlib.projections.register_projection	264
matplotlib.pylab	264
matplotlib.pylab.Annotation	265
matplotlib.pylab.Arrow	265
matplotlib.pylab.Artist	265
matplotlib.pylab.AutoLocator	265
matplotlib.pylab.AxLine	266
matplotlib.pylab.Axes	266
matplotlib.pylab.BackendFilter	266
matplotlib.pylab.Button	266
matplotlib.pylab.Circle	267
matplotlib.pylab.Colorizer	267
matplotlib.pylab.ColorizingArtist	267

matplotlib.pyplot.Colormap	267
matplotlib.pyplot.DateFormatter	267
matplotlib.pyplot.DateLocator	268
matplotlib.pyplot.DayLocator	268
matplotlib.pyplot.Figure	268
matplotlib.pyplot.FigureBase	268
matplotlib.pyplot.FigureCanvasBase	268
matplotlib.pyplot.FigureManagerBase	269
matplotlib.pyplot.FixedFormatter	270
matplotlib.pyplot.FixedLocator	270
matplotlib.pyplot.FormatStrFormatter	270
matplotlib.pyplot.Formatter	270
matplotlib.pyplot.FuncFormatter	271
matplotlib.pyplot.GridSpec	271
matplotlib.pyplot.HourLocator	271
matplotlib.pyplot.IndexLocator	271
matplotlib.pyplot.Line2D	271
matplotlib.pyplot.LinearLocator	271
matplotlib.pyplot.Locator	272
matplotlib.pyplot.LogFormatter	272
matplotlib.pyplot.LogFormatterExponent	273
matplotlib.pyplot.LogFormatterMathtext	273
matplotlib.pyplot.LogLocator	273
matplotlib.pyplot.MaxNLocator	273
matplotlib.pyplot.MinuteLocator	273

matplotlib.pylab.MonthLocator	274
matplotlib.pylab.MouseButton	274
matplotlib.pylab.MultipleLocator	274
matplotlib.pylab.Normalize	274
matplotlib.pylab.NullFormatter	274
matplotlib.pylab.NullLocator	274
matplotlib.pylab.PolarAxes	275
matplotlib.pylab.Polygon	275
matplotlib.pylab.RRRuleLocator	275
matplotlib.pylab.Rectangle	275
matplotlib.pylab.ScalarFormatter	275
matplotlib.pylab.SecondLocator	276
matplotlib.pylab.Slider	276
matplotlib.pylab.Subplot	277
matplotlib.pylab.SubplotSpec	277
matplotlib.pylab.Text	278
matplotlib.pylab.TickHelper	278
matplotlib.pylab.WeekdayLocator	278
matplotlib.pylab.Widget	278
matplotlib.pylab.YearLocator	278
matplotlib.pylab.acorr	278
matplotlib.pylab.angle_spectrum	280
matplotlib.pylab.annotate	282
matplotlib.pylab.arrow	285
matplotlib.pylab.autoscale	287
matplotlib.pylab.autumn	288

matplotlib.pyplot.axes	288
matplotlib.pyplot.axhline	290
matplotlib.pyplot.axhspan	293
matplotlib.pyplot.axis	294
matplotlib.pyplot.axline	295
matplotlib.pyplot.axvline	297
matplotlib.pyplot.axvspan	300
matplotlib.pyplot.bar	301
matplotlib.pyplot.bar_label	304
matplotlib.pyplot.barbs	306
matplotlib.pyplot.barh	309
matplotlib.pyplot.bone	312
matplotlib.pyplot.box	312
matplotlib.pyplot.boxplot	313
matplotlib.pyplot.broken_barh	317
matplotlib.pyplot.cla	319
matplotlib.pyplot.clabel	319
matplotlib.pyplot.clf	320
matplotlib.pyplot.clim	320
matplotlib.pyplot.close	320
matplotlib.pyplot.cohere	320
matplotlib.pyplot.colorbar	323
matplotlib.pyplot.connect	326
matplotlib.pyplot.contour	327
matplotlib.pyplot.contourf	333

matplotlib.pylab.cool	338
matplotlib.pylab.copper	338
matplotlib.pylab.csd	338
matplotlib.pylab.date2num	341
matplotlib.pylab.datestr2num	342
matplotlib.pylab.delaxes	342
matplotlib.pylab.detrend	342
matplotlib.pylab.detrend_linear	343
matplotlib.pylab.detrend_mean	343
matplotlib.pylab.detrend_none	343
matplotlib.pylab.disconnect	344
matplotlib.pylab.drange	344
matplotlib.pylab.draw	344
matplotlib.pylab.ecdf	345
matplotlib.pylab.errorbar	347
matplotlib.pylab.eventplot	350
matplotlib.pylab.fgaspect	353
matplotlib.pylab.figimage	353
matplotlib.pylab.figlegend	355
matplotlib.pylab.fignum_exists	361
matplotlib.pylab.figtext	361
matplotlib.pylab.figure	363
matplotlib.pylab.fill	365
matplotlib.pylab.fill_between	366
matplotlib.pylab.fill_betweenx	369
matplotlib.pylab.findobj	371

matplotlib.pylab.flag	372
matplotlib.pylab.flatten	372
matplotlib.pylab.gca	372
matplotlib.pylab.gcf	372
matplotlib.pylab.gci	373
matplotlib.pylab.get	373
matplotlib.pylab.get_backend	374
matplotlib.pylab.get_cmap	374
matplotlib.pylab.get_current_fig_manager	374
matplotlib.pylab.get_figlabels	375
matplotlib.pylab.get_fignums	375
matplotlib.pylab.get_plot_commands	375
matplotlib.pylab.get_scale_names	375
matplotlib.pylab.getp	375
matplotlib.pylab.ginput	376
matplotlib.pylab.gray	377
matplotlib.pylab.grid	377
matplotlib.pylab.hexbin	379
matplotlib.pylab.hist	383
matplotlib.pylab.hist2d	387
matplotlib.pylab.hlines	389
matplotlib.pylab.hot	390
matplotlib.pylab.hsv	390
matplotlib.pylab.imread	390
matplotlib.pylab.imsave	391

matplotlib.pylab.imshow	392
matplotlib.pylab.inferno	397
matplotlib.pylab.install_repl_displayhook	397
matplotlib.pylab.interactive	397
matplotlib.pylab.ioff	397
matplotlib.pylab.ion	398
matplotlib.pylab.isinteractive	399
matplotlib.pylab.jet	399
matplotlib.pylab.legend	399
matplotlib.pylab.locator_params	405
matplotlib.pylab.loglog	406
matplotlib.pylab.magma	407
matplotlib.pylab.magnitude_spectrum	407
matplotlib.pylab.margins	410
matplotlib.pylab.matshow	411
matplotlib.pylab.minorticks_off	412
matplotlib.pylab.minorticks_on	412
matplotlib.pylab.new_figure_manager	412
matplotlib.pylab.nipy_spectral	412
matplotlib.pylab.num2date	413
matplotlib.pylab.pause	413
matplotlib.pylab.pcolor	413
matplotlib.pylab.pcolormesh	418
matplotlib.pylab.phase_spectrum	422
matplotlib.pylab.pie	425
matplotlib.pylab.pink	427

matplotlib.pyplot.plasma	427
matplotlib.pyplot.plot	427
matplotlib.pyplot.plot_date	433
matplotlib.pyplot.polar	435
matplotlib.pyplot.prism	435
matplotlib.pyplot.psd	436
matplotlib.pyplot.quiver	439
matplotlib.pyplot.quiverkey	444
matplotlib.pyplot.rc	445
matplotlib.pyplot.rc_context	446
matplotlib.pyplot.rcdefaults	447
matplotlib.pyplot.rgrids	448
matplotlib.pyplot.savefig	449
matplotlib.pyplot.sca	451
matplotlib.pyplot.scatter	451
matplotlib.pyplot.sci	455
matplotlib.pyplot.semilogx	455
matplotlib.pyplot.semilogy	456
matplotlib.pyplot.set_cmap	457
matplotlib.pyplot.set_loglevel	457
matplotlib.pyplot.setp	458
matplotlib.pyplot.show	459
matplotlib.pyplot.silent_list	460
matplotlib.pyplot.specgram	461
matplotlib.pyplot.spring	463

matplotlib.pylab.spy	464
matplotlib.pylab.stackplot	466
matplotlib.pylab.stairs	467
matplotlib.pylab.stem	468
matplotlib.pylab.step	470
matplotlib.pylab.streamplot	471
matplotlib.pylab.subplot	473
matplotlib.pylab.subplot2grid	476
matplotlib.pylab.subplot_mosaic	477
matplotlib.pylab.subplot_tool	479
matplotlib.pylab.subplots	479
matplotlib.pylab.subplots_adjust	482
matplotlib.pylab.summer	483
matplotlib.pylab.suptitle	483
matplotlib.pylab.switch_backend	484
matplotlib.pylab.table	484
matplotlib.pylab.text	486
matplotlib.pylab.thetagrids	488
matplotlib.pylab.tick_params	489
matplotlib.pylab.ticklabel_format	491
matplotlib.pylab.tight_layout	492
matplotlib.pylab.title	492
matplotlib.pylab.tricontour	494
matplotlib.pylab.tricontourf	497
matplotlib.pylab.tripcolor	500
matplotlib.pylab.triplot	503

matplotlib.pylab.twinx	503
matplotlib.pylab.twiny	504
matplotlib.pylab.uninstall_repl_displayhook	504
matplotlib.pylab.violinplot	504
matplotlib.pylab.viridis	506
matplotlib.pylab.vlines	506
matplotlib.pylab.waitforbuttonpress	507
matplotlib.pylab.window_hanning	508
matplotlib.pylab.window_none	508
matplotlib.pylab.winter	508
matplotlib.pylab.xcorr	508
matplotlib.pylab.xkcd	510
matplotlib.pylab.xlabel	510
matplotlib.pylab.xlim	511
matplotlib.pylab.xscale	512
matplotlib.pylab.xticks	512
matplotlib.pylab.ylabel	513
matplotlib.pylab.ylim	514
matplotlib.pylab.yscale	514
matplotlib.pylab.yticks	515
matplotlib.pyplot	516
matplotlib.pyplot.Annotation	517
matplotlib.pyplot.Arrow	517
matplotlib.pyplot.Artist	517
matplotlib.pyplot.AutoLocator	517

matplotlib.pyplot.AxLine	517
matplotlib.pyplot.Axes	518
matplotlib.pyplot.BackendFilter	518
matplotlib.pyplot.Button	518
matplotlib.pyplot.Circle	519
matplotlib.pyplot.Colorizer	519
matplotlib.pyplot.ColorizingArtist	519
matplotlib.pyplot.Colormap	519
matplotlib.pyplot.Figure	519
matplotlib.pyplot.FigureBase	520
matplotlib.pyplot.FigureCanvasBase	520
matplotlib.pyplot.FigureManagerBase	520
matplotlib.pyplot.FixedFormatter	521
matplotlib.pyplot.FixedLocator	521
matplotlib.pyplot.FormatStrFormatter	522
matplotlib.pyplot.Formatter	522
matplotlib.pyplot.FuncFormatter	522
matplotlib.pyplot.GridSpec	522
matplotlib.pyplot.IndexLocator	522
matplotlib.pyplot.Line2D	523
matplotlib.pyplot.LinearLocator	523
matplotlib.pyplot.Locator	523
matplotlib.pyplot.LogFormatter	523
matplotlib.pyplot.LogFormatterExponent	524
matplotlib.pyplot.LogFormatterMathtext	524
matplotlib.pyplot.LogLocator	525

matplotlib.pyplot.MaxNLocator	525
matplotlib.pyplot.MouseButton	525
matplotlib.pyplot.MultipleLocator	525
matplotlib.pyplot.Normalize	525
matplotlib.pyplot.NullFormatter	525
matplotlib.pyplot.NullLocator	526
matplotlib.pyplot.PolarAxes	526
matplotlib.pyplot.Polygon	526
matplotlib.pyplot.Rectangle	526
matplotlib.pyplot.ScalarFormatter	526
matplotlib.pyplot.Slider	527
matplotlib.pyplot.Subplot	528
matplotlib.pyplot.SubplotSpec	528
matplotlib.pyplot.Text	528
matplotlib.pyplot.TickHelper	529
matplotlib.pyplot.Widget	529
matplotlib.pyplot.acorr	529
matplotlib.pyplot.angle_spectrum	530
matplotlib.pyplot.annotate	533
matplotlib.pyplot.arrow	536
matplotlib.pyplot.autoscale	538
matplotlib.pyplot.autumn	538
matplotlib.pyplot.axes	539
matplotlib.pyplot.axhline	541
matplotlib.pyplot.axhspan	543

matplotlib.pyplot.axis	545
matplotlib.pyplot.axline	546
matplotlib.pyplot.axvline	548
matplotlib.pyplot.axvspan	550
matplotlib.pyplot.bar	552
matplotlib.pyplot.bar_label	555
matplotlib.pyplot.barbs	556
matplotlib.pyplot.barh	560
matplotlib.pyplot.bone	563
matplotlib.pyplot.box	563
matplotlib.pyplot.boxplot	563
matplotlib.pyplot.broken_barh	568
matplotlib.pyplot.cla	569
matplotlib.pyplot.clabel	570
matplotlib.pyplot.clf	570
matplotlib.pyplot.clim	570
matplotlib.pyplot.close	570
matplotlib.pyplot.cohere	571
matplotlib.pyplot.colorbar	574
matplotlib.pyplot.connect	577
matplotlib.pyplot.contour	578
matplotlib.pyplot.contourf	583
matplotlib.pyplot.cool	588
matplotlib.pyplot.copper	588
matplotlib.pyplot.csd	588
matplotlib.pyplot.delaxes	592

matplotlib.pyplot.disconnect	592
matplotlib.pyplot.draw	592
matplotlib.pyplot.ecdf	592
matplotlib.pyplot.errorbar	595
matplotlib.pyplot.eventplot	598
matplotlib.pyplot.fgaspect	600
matplotlib.pyplot.figimage	601
matplotlib.pyplot.figlegend	603
matplotlib.pyplot.fignum_exists	609
matplotlib.pyplot.figtext	609
matplotlib.pyplot.figure	611
matplotlib.pyplot.fill	613
matplotlib.pyplot.fill_between	614
matplotlib.pyplot.fill_betweenx	616
matplotlib.pyplot.findobj	619
matplotlib.pyplot.flag	619
matplotlib.pyplot.gca	620
matplotlib.pyplot.gcf	620
matplotlib.pyplot.gci	620
matplotlib.pyplot.get	620
matplotlib.pyplot.get_backend	621
matplotlib.pyplot.get_cmap	622
matplotlib.pyplot.get_current_fig_manager	622
matplotlib.pyplot.get_figlabels	622
matplotlib.pyplot.get_fignums	622

matplotlib.pyplot.get_plot_commands	622
matplotlib.pyplot.get_scale_names	623
matplotlib.pyplot.getp	623
matplotlib.pyplot.ginput	623
matplotlib.pyplot.gray	624
matplotlib.pyplot.grid	625
matplotlib.pyplot.hexbin	626
matplotlib.pyplot.hist	631
matplotlib.pyplot.hist2d	634
matplotlib.pyplot.hlines	637
matplotlib.pyplot.hot	638
matplotlib.pyplot.hsv	638
matplotlib.pyplot.imread	638
matplotlib.pyplot.imsave	639
matplotlib.pyplot.imshow	640
matplotlib.pyplot.inferno	644
matplotlib.pyplot.install_repl_displayhook	644
matplotlib.pyplot.interactive	645
matplotlib.pyplot.ioff	645
matplotlib.pyplot.ion	645
matplotlib.pyplot.isinteractive	646
matplotlib.pyplot.jet	647
matplotlib.pyplot.legend	647
matplotlib.pyplot.locator_params	653
matplotlib.pyplot.loglog	653
matplotlib.pyplot.magma	654

matplotlib.pyplot.magnitude_spectrum	655
matplotlib.pyplot.margins	657
matplotlib.pyplot.matshow	658
matplotlib.pyplot.minorticks_off	659
matplotlib.pyplot.minorticks_on	659
matplotlib.pyplot.new_figure_manager	660
matplotlib.pyplot.nipy_spectral	660
matplotlib.pyplot.pause	660
matplotlib.pyplot.pcolor	660
matplotlib.pyplot.pcolormesh	664
matplotlib.pyplot.phase_spectrum	669
matplotlib.pyplot.pie	671
matplotlib.pyplot.pink	674
matplotlib.pyplot.plasma	674
matplotlib.pyplot.plot	674
matplotlib.pyplot.plot_date	680
matplotlib.pyplot.polar	682
matplotlib.pyplot.prism	682
matplotlib.pyplot.psd	682
matplotlib.pyplot.quiver	686
matplotlib.pyplot.quiverkey	691
matplotlib.pyplot.rc	692
matplotlib.pyplot.rc_context	693
matplotlib.pyplot.rcdefaults	694
matplotlib.pyplot.rgrids	695

matplotlib.pyplot.savefig	696
matplotlib.pyplot.sca	698
matplotlib.pyplot.scatter	698
matplotlib.pyplot.sci	702
matplotlib.pyplot.semilogx	702
matplotlib.pyplot.semilogy	703
matplotlib.pyplot.set_cmap	704
matplotlib.pyplot.set_loglevel	704
matplotlib.pyplot.setp	705
matplotlib.pyplot.show	706
matplotlib.pyplot.specgram	707
matplotlib.pyplot.spring	710
matplotlib.pyplot.spy	710
matplotlib.pyplot.stackplot	712
matplotlib.pyplot.stairs	714
matplotlib.pyplot.stem	715
matplotlib.pyplot.step	717
matplotlib.pyplot.streamplot	718
matplotlib.pyplot.subplot	719
matplotlib.pyplot.subplot2grid	723
matplotlib.pyplot.subplot_mosaic	723
matplotlib.pyplot.subplot_tool	725
matplotlib.pyplot.subplots	726
matplotlib.pyplot.subplots_adjust	728
matplotlib.pyplot.summer	729
matplotlib.pyplot.suptitle	729

matplotlib.pyplot.switch_backend	730
matplotlib.pyplot.table	730
matplotlib.pyplot.text	732
matplotlib.pyplot.thetagrids	735
matplotlib.pyplot.tick_params	736
matplotlib.pyplot.ticklabel_format	737
matplotlib.pyplot.tight_layout	738
matplotlib.pyplot.title	739
matplotlib.pyplot.tricontour	740
matplotlib.pyplot.tricontourf	743
matplotlib.pyplot.tripcolor	747
matplotlib.pyplot.triplot	749
matplotlib.pyplot.twinx	750
matplotlib.pyplot.twiny	750
matplotlib.pyplot.uninstall_repl_displayhook	750
matplotlib.pyplot.violinplot	750
matplotlib.pyplot.viridis	753
matplotlib.pyplot.vlines	753
matplotlib.pyplot.waitforbuttonpress	754
matplotlib.pyplot.winter	754
matplotlib.pyplot.xcorr	754
matplotlib.pyplot.xkcd	756
matplotlib.pyplot.xlabel	756
matplotlib.pyplot.xlim	757
matplotlib.pyplot.xscale	758

matplotlib.pyplot.xticks	758
matplotlib.pyplot.ylabel	759
matplotlib.pyplot.ylim	760
matplotlib.pyplot.yscale	761
matplotlib.pyplot.yticks	761
matplotlib.quiver	762
matplotlib.quiver.Barbs	762
matplotlib.quiver.CirclePolygon	763
matplotlib.quiver.Quiver	763
matplotlib.quiver.QuiverKey	763
matplotlib.rc	763
matplotlib.rc_context	764
matplotlib.rc_file	765
matplotlib.rc_file_defaults	766
matplotlib.rc_params	766
matplotlib.rc_params_from_file	766
matplotlib.rcdefaults	766
matplotlib.rcsetup	766
matplotlib.rcsetup.BackendFilter	767
matplotlib.rcsetup.CapStyle	767
matplotlib.rcsetup.Colormap	768
matplotlib.rcsetup.JoinStyle	768
matplotlib.rcsetup.ValidateInStrings	769
matplotlib.rcsetup.cycler	769
matplotlib.rcsetup.is_color_like	770
matplotlib.rcsetup.validate_any	770

matplotlib.rcsetup.validate_anylist	770
matplotlib.rcsetup.validate_aspect	770
matplotlib.rcsetup.validate_axisbelow	770
matplotlib.rcsetup.validate_backend	770
matplotlib.rcsetup.validate_bbox	771
matplotlib.rcsetup.validate_bool	771
matplotlib.rcsetup.validate_color	771
matplotlib.rcsetup.validate_color_for_prop_cycle	771
matplotlib.rcsetup.validate_color_or_auto	771
matplotlib.rcsetup.validate_color_or_inherit	771
matplotlib.rcsetup.validate_colorlist	771
matplotlib.rcsetup.validate_cycler	771
matplotlib.rcsetup.validate_dashlist	771
matplotlib.rcsetup.validate_dpi	772
matplotlib.rcsetup.validate_fillstylelist	772
matplotlib.rcsetup.validate_float	772
matplotlib.rcsetup.validate_float_or_None	772
matplotlib.rcsetup.validate_floatlist	772
matplotlib.rcsetup.validate_font_properties	772
matplotlib.rcsetup.validate_fontsize	772
matplotlib.rcsetup.validate_fontsize_None	772
matplotlib.rcsetup.validate_fontsizelist	772
matplotlib.rcsetup.validate_fontstretch	773
matplotlib.rcsetup.validate_fonttype	773
matplotlib.rcsetup.validate_fontweight	773

matplotlib.rcsetup.validate_hatch	773
matplotlib.rcsetup.validate_hatchlist	773
matplotlib.rcsetup.validate_hist_bins	773
matplotlib.rcsetup.validate_int	773
matplotlib.rcsetup.validate_int_or_None	773
matplotlib.rcsetup.validate_markevery	774
matplotlib.rcsetup.validate_markeverylist	774
matplotlib.rcsetup.validate_ps_distiller	774
matplotlib.rcsetup.validate_sketch	774
matplotlib.rcsetup.validate_string	774
matplotlib.rcsetup.validate_string_or_None	774
matplotlib.rcsetup.validate_stringlist	774
matplotlib.rcsetup.validate_whiskers	775
matplotlib.sanitize_sequence	775
matplotlib.sankey	775
matplotlib.sankey.Affine2D	775
matplotlib.sankey.Path	775
matplotlib.sankey.PathPatch	776
matplotlib.sankey.Sankey	776
matplotlib.scale	776
matplotlib.scale.AsinhLocator	777
matplotlib.scale.AsinhScale	777
matplotlib.scale.AsinhTransform	778
matplotlib.scale.AutoLocator	778
matplotlib.scale.AutoMinorLocator	778
matplotlib.scale.FuncScale	778

matplotlib.scale.FuncScaleLog	778
matplotlib.scale.FuncTransform	779
matplotlib.scale.IdentityTransform	779
matplotlib.scale.InvertedAsinhTransform	779
matplotlib.scale.InvertedLogTransform	779
matplotlib.scale.InvertedSymmetricalLogTransform	779
matplotlib.scale.LinearScale	780
matplotlib.scale.LogFormatterSciNotation	780
matplotlib.scale.LogLocator	780
matplotlib.scale.LogScale	780
matplotlib.scale.LogTransform	781
matplotlib.scale.LogisticTransform	781
matplotlib.scale.LogitFormatter	782
matplotlib.scale.LogitLocator	782
matplotlib.scale.LogitScale	782
matplotlib.scale.LogitTransform	782
matplotlib.scale.NullFormatter	783
matplotlib.scale.NullLocator	783
matplotlib.scale.ScalarFormatter	783
matplotlib.scale.ScaleBase	784
matplotlib.scale.SymmetricalLogLocator	784
matplotlib.scale.SymmetricalLogScale	784
matplotlib.scale.SymmetricalLogTransform	785
matplotlib.scale.Transform	786
matplotlib.scale.get_scale_names	786

matplotlib.scale.register_scale	786
matplotlib.scale.scale_factory	787
matplotlib.set_loglevel	787
matplotlib.sphinxext	787
matplotlib.sphinxext.figmpl_directive	787
matplotlib.sphinxext.mathmpl	788
matplotlib.sphinxext.plot_directive	789
matplotlib.sphinxext.roles	792
matplotlib.spines	793
matplotlib.spines.Spine	793
matplotlib.spines.Spines	793
matplotlib.spines.SpinesProxy	794
matplotlib.spines.allow_rasterization	794
matplotlib.stackplot	794
matplotlib.stackplot.stackplot	794
matplotlib.streamplot	796
matplotlib.streamplot.DomainMap	796
matplotlib.streamplot.Grid	796
matplotlib.streamplot.InvalidIndexError	796
matplotlib.streamplot.OutOfBounds	796
matplotlib.streamplot.StreamMask	796
matplotlib.streamplot.StreamplotSet	797
matplotlib.streamplot.TerminateTrajectory	797
matplotlib.streamplot.interpgrid	797
matplotlib.streamplot.streamplot	797
matplotlib.style	798

matplotlib.style.context	798
matplotlib.style.core	799
matplotlib.style.reload_library	799
matplotlib.style.use	799
matplotlib.table	801
matplotlib.table.Artist	801
matplotlib.table.Bbox	801
matplotlib.table.Cell	803
matplotlib.table.CustomCell	803
matplotlib.table.Path	803
matplotlib.table.Rectangle	804
matplotlib.table.Table	804
matplotlib.table.Text	805
matplotlib.table.allow_rasterization	805
matplotlib.table.table	805
matplotlib.testing	807
matplotlib.testing.compare	807
matplotlib.testing.decorators	807
matplotlib.testing.exceptions	807
matplotlib.testing.ipython_in_subprocess	807
matplotlib.testing.is_ci_environment	808
matplotlib.testing.jpl_units	808
matplotlib.testing.set_font_settings_for_testing	808
matplotlib.testing.set_reproducibility_for_testing	808
matplotlib.testing.setup	808

matplotlib.testing.subprocess_run_for_testing	809
matplotlib.testing.subprocess_run_helper	809
matplotlib.testing.widgets	810
matplotlib.texmanager	810
matplotlib.texmanager.TextManager	810
matplotlib.text	810
matplotlib.text.Affine2D	810
matplotlib.text.Annotation	811
matplotlib.text.Artist	811
matplotlib.text.Bbox	811
matplotlib.text.BboxBase	812
matplotlib.text.BboxTransformTo	813
matplotlib.text.FancyArrowPatch	813
matplotlib.text.FancyBboxPatch	813
matplotlib.text.FontProperties	813
matplotlib.text.IdentityTransform	815
matplotlib.text.OffsetFrom	815
matplotlib.text.Rectangle	815
matplotlib.text.Text	815
matplotlib.text.TextPath	815
matplotlib.text.TextToPath	815
matplotlib.text.Transform	816
matplotlib.ticker	816
matplotlib.ticker.AsinhLocator	819
matplotlib.ticker.AutoLocator	819
matplotlib.ticker.AutoMinorLocator	819

matplotlib.ticker.EngFormatter	819
matplotlib.ticker.FixedFormatter	819
matplotlib.ticker.FixedLocator	820
matplotlib.ticker.FormatStrFormatter	820
matplotlib.ticker.Formatter	820
matplotlib.ticker.FuncFormatter	820
matplotlib.ticker.IndexLocator	820
matplotlib.ticker.LinearLocator	820
matplotlib.ticker.Locator	821
matplotlib.ticker.LogFormatter	821
matplotlib.ticker.LogFormatterExponent	822
matplotlib.ticker.LogFormatterMathtext	822
matplotlib.ticker.LogFormatterSciNotation	822
matplotlib.ticker.LogLocator	822
matplotlib.ticker.LogitFormatter	822
matplotlib.ticker.LogitLocator	823
matplotlib.ticker.MaxNLocator	823
matplotlib.ticker.MultipleLocator	823
matplotlib.ticker.NullFormatter	823
matplotlib.ticker.NullLocator	823
matplotlib.ticker.PercentFormatter	823
matplotlib.ticker.ScalarFormatter	824
matplotlib.ticker.StrMethodFormatter	825
matplotlib.ticker.SymmetricalLogLocator	825
matplotlib.ticker.TickHelper	825

matplotlib.ticker.scale_range	825
matplotlib.transforms	825
matplotlib.transforms.Affine2D	826
matplotlib.transforms.Affine2DBase	826
matplotlib.transforms.AffineBase	826
matplotlib.transforms.AffineDeltaTransform	826
matplotlib.transforms.Bbox	827
matplotlib.transforms.BboxBase	828
matplotlib.transforms.BboxTransform	829
matplotlib.transforms.BboxTransformFrom	829
matplotlib.transforms.BboxTransformTo	829
matplotlib.transforms.BboxTransformToMaxOnly	829
matplotlib.transforms.BlendedAffine2D	829
matplotlib.transforms.BlendedGenericTransform	829
matplotlib.transforms.CompositeAffine2D	829
matplotlib.transforms.CompositeGenericTransform	830
matplotlib.transforms.IdentityTransform	830
matplotlib.transforms.LockableBbox	830
matplotlib.transforms.Path	830
matplotlib.transforms.ScaledTranslation	831
matplotlib.transforms.Transform	831
matplotlib.transforms.TransformNode	832
matplotlib.transforms.TransformWrapper	832
matplotlib.transforms.TransformedBbox	832
matplotlib.transforms.TransformedPatchPath	832
matplotlib.transforms.TransformedPath	833

matplotlib.transforms.blended_transform_factory	833
matplotlib.transforms.composite_transform_factory	833
matplotlib.transforms.interval_contains	833
matplotlib.transforms.interval_contains_open	834
matplotlib.transforms.nonsingular	834
matplotlib.transforms.offset_copy	835
matplotlib.tri	835
matplotlib.tri.CubicTriInterpolator	835
matplotlib.tri.LinearTriInterpolator	837
matplotlib.tri.TrapezoidMapTriFinder	837
matplotlib.tri.TriAnalyzer	837
matplotlib.tri.TriContourSet	838
matplotlib.tri.TriFinder	838
matplotlib.tri.TriInterpolator	838
matplotlib.tri.TriRefiner	839
matplotlib.tri.Triangulation	839
matplotlib.tri.UniformTriRefiner	840
matplotlib.tri.tricontour	840
matplotlib.tri.tricontourf	843
matplotlib.tri.tripcolor	847
matplotlib.tri.triplot	849
matplotlib.typing	850
matplotlib.typing.Artist	850
matplotlib.typing.Bbox	850
matplotlib.typing.CapStyle	852

matplotlib.typing.JoinStyle	852
matplotlib.typing.MarkerStyle	853
matplotlib.typing.RendererBase	854
matplotlib.typing.Transform	854
matplotlib.units	854
matplotlib.units.AxisInfo	855
matplotlib.units.ConversionError	855
matplotlib.units.ConversionInterface	856
matplotlib.units.DecimalConverter	856
matplotlib.units.Registry	856
matplotlib.use	856
matplotlib.validate_backend	857
matplotlib.widgets	857
matplotlib.widgets.Affine2D	857
matplotlib.widgets.AxesWidget	857
matplotlib.widgets.Button	858
matplotlib.widgets.CheckButtons	858
matplotlib.widgets.Cursor	858
matplotlib.widgets.Ellipse	859
matplotlib.widgets.EllipseSelector	859
matplotlib.widgets.Lasso	861
matplotlib.widgets.LassoSelector	862
matplotlib.widgets.Line2D	862
matplotlib.widgets.LockDraw	863
matplotlib.widgets.MultiCursor	863
matplotlib.widgets.Polygon	863

matplotlib.widgets.PolygonSelector	864
matplotlib.widgets.RadioButtons	865
matplotlib.widgets.RangeSlider	865
matplotlib.widgets.Rectangle	866
matplotlib.widgets.RectangleSelector	866
matplotlib.widgets.Slider	868
matplotlib.widgets.SliderBase	869
matplotlib.widgets.SpanSelector	869
matplotlib.widgets.SubplotTool	870
matplotlib.widgets.TextBox	871
matplotlib.widgets.ToolHandles	871
matplotlib.widgets.ToolLineHandles	871
matplotlib.widgets.TransformedPatchPath	872
matplotlib.widgets.Widget	872

matplotlib

```
matplotlib(...)
```

An object-oriented plotting library.

A procedural interface is provided by the companion pyplot module, which may be imported directly, e.g.::

```
import matplotlib.pyplot as plt
```

or using ipython::

```
ipython
```

at your terminal, followed by::

```
In [1]: %matplotlib
```

```
In [2]: import matplotlib.pyplot as plt
```

at the ipython shell prompt.

For the most part, direct use of the explicit object-oriented library is encouraged when programming; the implicit pyplot interface is primarily for working interactively. The exceptions to this suggestion are the pyplot functions ``matplotlib.pyplot.figure``, ``matplotlib.pyplot.subplot``, ``matplotlib.pyplot.subplots``, and ``matplotlib.pyplot.savefig``, which can greatly simplify scripting. See :ref:`api_interfaces` for an explanation of the tradeoffs between the implicit and explicit interfaces.

Modules include:

:mod:`matplotlib.axes`

The ``matplotlib.axes.Axes`` class. Most pyplot functions are wrappers for ``matplotlib.axes.Axes`` methods. The axes module is the highest level of OO access to the library.

:mod:`matplotlib.figure`

The ``matplotlib.figure.Figure`` class.

:mod:`matplotlib.artist`

The ``matplotlib.artist.Artist`` base class for all classes that draw things.

:mod:`matplotlib.lines`

The ``matplotlib.lines.Line2D`` class for drawing lines and markers.

:mod:`matplotlib.patches`

Classes for drawing polygons.

:mod:`matplotlib.text`

The ``matplotlib.text.Text`` and ``matplotlib.text.Annotation`` classes.

:mod:`matplotlib.image`

The ``matplotlib.image.AxesImage`` and ``matplotlib.image.FigureImage`` classes.

:mod:`matplotlib.collections`

Classes for efficient drawing of groups of lines or polygons.

`:mod:`matplotlib.colors``

Color specifications and making colormaps.

`:mod:`matplotlib.cm``

Colormaps, and the `.ScalarMappable`` mixin class for providing color mapping functionality to other classes.

`:mod:`matplotlib.ticker``

Calculation of tick mark locations and formatting of tick labels.

`:mod:`matplotlib.backends``

A subpackage with modules for various GUI libraries and output formats.

The base matplotlib namespace includes:

``~matplotlib.rcParams``

Default configuration settings; their defaults may be overridden using a `:file:`matplotlibrc`` file.

``~matplotlib.use``

Setting the Matplotlib backend. This should be called before any figure is created, because it is not possible to switch between different GUI backends after that.

The following environment variables can be used to customize the behavior:

`:envvar:`MPLBACKEND``

This optional variable can be set to choose the Matplotlib backend. See `:ref:`what-is-a-backend``.

`:envvar:`MPLCONFIGDIR``

This is the directory used to store user customizations to Matplotlib, as well as some caches to improve performance. If `:envvar:`MPLCONFIGDIR`` is not defined, `:file:`{HOME}/.config/matplotlib`` and `:file:`{HOME}/.cache/matplotlib`` are used on Linux, and `:file:`{HOME}/.matplotlib`` on other platforms, if they are writable. Otherwise, the Python standard library's ``tempfile.gettempdir`` is used to find a base directory in which the `:file:`matplotlib`` subdirectory is created.

Matplotlib was initially written by John D. Hunter (1968-2012) and is now developed and maintained by a host of others.

Occasionally the internal documentation (python docstrings) will refer to MATLAB®, a registered trademark of The MathWorks, Inc.

matplotlib.ExecutableNotFoundError

```
ExecutableNotFoundError(...)
```

Error raised when an executable that Matplotlib optionally depends on can't be found.

matplotlib.MatplotlibDeprecationWarning

```
MatplotlibDeprecationWarning(...)
```

A class for issuing deprecation warnings for Matplotlib users.

matplotlib.RcParams

```
RcParams(*args, **kwargs)
```

A dict-like key-value store for config parameters, including validation.

Validating functions are defined and associated with rc parameters in `:mod:`matplotlib.rcsetup``.

The list of rcParams is:

- `_internal.classic_mode`
- `agg.path.chunksize`
- `animation.bitrate`
- `animation.codec`
- `animation.convert_args`
- `animation.convert_path`
- `animation.embed_limit`
- `animation.ffmpeg_args`
- `animation.ffmpeg_path`
- `animation.frame_format`
- `animation.html`
- `animation.writer`
- `axes.autolimit_mode`
- `axes.axisbelow`
- `axes.edgecolor`
- `axes.facecolor`
- `axes.formatter.limits`
- `axes.formatter.min_exponent`
- `axes.formatter.offset_threshold`
- `axes.formatter.use_locale`
- `axes.formatter.use_mathtext`
- `axes.formatter.useoffset`
- `axes.grid`
- `axes.grid.axis`
- `axes.grid.which`
- `axes.labelcolor`
- `axes.labelpad`
- `axes.labelsize`
- `axes.labelweight`
- `axes.linewidth`
- `axes.prop_cycle`
- `axes.spines.bottom`
- `axes.spines.left`
- `axes.spines.right`
- `axes.spines.top`
- `axes.titlecolor`
- `axes.titlelocation`
- `axes.titlepad`

- axes.titlesize
- axes.titleweight
- axes.titley
- axes.unicode_minus
- axes.xmargin
- axes.ymargin
- axes.zmargin
- axes3d.automargin
- axes3d.grid
- axes3d.mouserotationstyle
- axes3d.trackballborder
- axes3d.trackballsize
- axes3d.xaxis.panecolor
- axes3d.yaxis.panecolor
- axes3d.zaxis.panecolor
- backend
- backend_fallback
- boxplot.bootstrap
- boxplot.boxprops.color
- boxplot.boxprops.linestyle
- boxplot.boxprops.linewidth
- boxplot.capprops.color
- boxplot.capprops.linestyle
- boxplot.capprops.linewidth
- boxplot.flierprops.color
- boxplot.flierprops.linestyle
- boxplot.flierprops.linewidth
- boxplot.flierprops.marker
- boxplot.flierprops.markeredgecolor
- boxplot.flierprops.markeredgewidth
- boxplot.flierprops.markerfacecolor
- boxplot.flierprops.markersize
- boxplot.meanline
- boxplot.meanprops.color
- boxplot.meanprops.linestyle
- boxplot.meanprops.linewidth
- boxplot.meanprops.marker
- boxplot.meanprops.markeredgecolor
- boxplot.meanprops.markerfacecolor
- boxplot.meanprops.markersize
- boxplot.medianprops.color
- boxplot.medianprops.linestyle
- boxplot.medianprops.linewidth
- boxplot.notch
- boxplot.patchartist
- boxplot.showbox
- boxplot.showcaps
- boxplot.showfliers
- boxplot.showmeans
- boxplot.vertical
- boxplot.whiskerprops.color
- boxplot.whiskerprops.linestyle
- boxplot.whiskerprops.linewidth
- boxplot.whiskers
- contour.algorithm

- contour.corner_mask
- contour.linewidth
- contour.negative_linestyle
- date.autoformatter.day
- date.autoformatter.hour
- date.autoformatter.microsecond
- date.autoformatter.minute
- date.autoformatter.month
- date.autoformatter.second
- date.autoformatter.year
- date.converter
- date.epoch
- date.interval_multiples
- docstring.hardcopy
- errorbar.capsize
- figure.autolayout
- figure.constrained_layout.h_pad
- figure.constrained_layout.hspace
- figure.constrained_layout.use
- figure.constrained_layout.w_pad
- figure.constrained_layout.wspace
- figure.dpi
- figure.edgecolor
- figure.facecolor
- figure.figsize
- figure.frameon
- figure.hooks
- figure.labelsize
- figure.labelweight
- figure.max_open_warning
- figure.raise_window
- figure.subplot.bottom
- figure.subplot.hspace
- figure.subplot.left
- figure.subplot.right
- figure.subplot.top
- figure.subplot.wspace
- figure.titlesize
- figure.titleweight
- font.cursive
- font.family
- font.fantasy
- font.monospace
- font.sans-serif
- font.serif
- font.size
- font.stretch
- font.style
- font.variant
- font.weight
- grid.alpha
- grid.color
- grid.linestyle
- grid.linewidth
- hatch.color

- hatch.linewidth
- hist.bins
- image.aspect
- image.cmap
- image.composite_image
- image.interpolation
- image.interpolation_stage
- image.lut
- image.origin
- image.resample
- interactive
- keymap.back
- keymap.copy
- keymap.forward
- keymap.fullscreen
- keymap.grid
- keymap.grid_minor
- keymap.help
- keymap.home
- keymap.pan
- keymap.quit
- keymap.quit_all
- keymap.save
- keymap.xscale
- keymap.yscale
- keymap.zoom
- legend.borderaxespad
- legend.borderpad
- legend.columnspacing
- legend.edgecolor
- legend.facecolor
- legend.fancybox
- legend.fontsize
- legend.framealpha
- legend.frameon
- legend.handleheight
- legend.handlelength
- legend.handletextpad
- legend.labelcolor
- legend.labelspadding
- legend.loc
- legend.markerscale
- legend.numpoints
- legend.scatterpoints
- legend.shadow
- legend.title_fontsize
- lines.antialiased
- lines.color
- lines.dash_capstyle
- lines.dash_joinstyle
- lines.dashdot_pattern
- lines.dashed_pattern
- lines.dotted_pattern
- lines.linestyle
- lines.linewidth

- lines.marker
- lines.markeredgewidth
- lines.markeredgewidth
- lines.markerfacecolor
- lines.markersize
- lines.scale_dashes
- lines.solid_capstyle
- lines.solid_joinstyle
- macosx.window_mode
- markers.fillstyle
- mathtext.bf
- mathtext.bfit
- mathtext.cal
- mathtext.default
- mathtext.fallback
- mathtext.fontset
- mathtext.it
- mathtext.rm
- mathtext.sf
- mathtext.tt
- patch.antialiased
- patch.edgecolor
- patch.facecolor
- patch.force_edgecolor
- patch.linewidth
- path.effects
- path.simplify
- path.simplify_threshold
- path.sketch
- path.snap
- pcolor.shading
- pcolormesh.snap
- pdf.compression
- pdf.fonttype
- pdf.inheritcolor
- pdf.use14corefonts
- pgf.preamble
- pgf.rcfonts
- pgf.texsystem
- polaraxes.grid
- ps.distiller.res
- ps.fonttype
- ps.papersize
- ps.useafm
- ps.usedistiller
- savefig.bbox
- savefig.directory
- savefig.dpi
- savefig.edgecolor
- savefig.facecolor
- savefig.format
- savefig.orientation
- savefig.pad_inches
- savefig.transparent
- scatter.edgecolors

- scatter.marker
- svg.fonttype
- svg.hashsalt
- svg.id
- svg.image_inline
- text.antialiased
- text.color
- text.hinting
- text.hinting_factor
- text.kerning_factor
- text.latex.preamble
- text.parse_math
- text.usetex
- timezone
- tk.window_focus
- toolbar
- webagg.address
- webagg.open_in_browser
- webagg.port
- webagg.port_retries
- xaxis.labellocation
- xtick.alignment
- xtick.bottom
- xtick.color
- xtick.direction
- xtick.labelbottom
- xtick.labelcolor
- xtick.labelsize
- xtick.labeltop
- xtick.major.bottom
- xtick.major.pad
- xtick.major.size
- xtick.major.top
- xtick.major.width
- xtick.minor.bottom
- xtick.minor.ndivs
- xtick.minor.pad
- xtick.minor.size
- xtick.minor.top
- xtick.minor.visible
- xtick.minor.width
- xtick.top
- yaxis.labellocation
- ytick.alignment
- ytick.color
- ytick.direction
- ytick.labelcolor
- ytick.labelleft
- ytick.labelright
- ytick.labelsize
- ytick.left
- ytick.major.left
- ytick.major.pad
- ytick.major.right
- ytick.major.size

- ytick.major.width
- ytick.minor.left
- ytick.minor.ndivs
- ytick.minor.pad
- ytick.minor.right
- ytick.minor.size
- ytick.minor.visible
- ytick.minor.width
- ytick.right

See Also

:ref:`customizing-with-matplotlibrc-files`

matplotlib.RcParams.copy

```
copy(self)
```

Copy this RcParams instance.

matplotlib.RcParams.find_all

```
find_all(self, pattern)
```

Return the subset of this RcParams dictionary whose keys match, using `:func:`re.search``, the given `pattern`.

.. note::

Changes to the returned dictionary are **not** propagated to the parent RcParams dictionary.

matplotlib.animation

```
animation(...)
```

No description available.

matplotlib.animation.AbstractMovieWriter

```
AbstractMovieWriter(fps=5, metadata=None, codec=None, bitrate=None)
```

Abstract base class for writing movies, providing a way to grab frames by calling `~AbstractMovieWriter.grab_frame`.

``setup`` is called to start the process and ``finish`` is called afterwards.

``saving`` is provided as a context manager to facilitate this process as ::

```
with moviewriter.saving(fig, outfile='myfile.mp4', dpi=100):
```

```
# Iterate over frames
```

```
moviewriter.grab_frame(**savefig_kwargs)
```

The use of the context manager ensures that ``setup`` and ``finish`` are performed as necessary.

An instance of a concrete subclass of this class can be given as the ``writer`` argument of `Animation.save()`.

matplotlib.animation.Animation

```
Animation(fig, event_source=None, blit=False)
```

A base class for Animations.

This class is not usable as is, and should be subclassed to provide needed behavior.

.. note::

You must store the created Animation in a variable that lives as long as the animation should run. Otherwise, the Animation object will be garbage-collected and the animation stops.

Parameters

`fig` : `~matplotlib.figure.Figure`

The figure object used to get needed events, such as draw or resize.

`event_source` : object, optional

A class that can run a callback when desired events are generated, as well as be stopped and started.

Examples include timers (see `TimedAnimation`) and file system notifications.

`blit` : bool, default: False

Whether blitting is used to optimize drawing. If the backend does not support blitting, then this parameter has no effect.

See Also

`FuncAnimation`, `ArtistAnimation`

matplotlib.animation.ArtistAnimation

```
ArtistAnimation(fig, artists, *args, **kwargs)
```

`TimedAnimation` subclass that creates an animation by using a fixed set of `.Artist` objects.

Before creating an instance, all plotting should have taken place and the relevant artists saved.

.. note::

You must store the created Animation in a variable that lives as long as the animation should run. Otherwise, the Animation object will be garbage-collected and the animation stops.

Parameters

`fig` : `~matplotlib.figure.Figure``

The figure object used to get needed events, such as draw or resize.

`artists` : list

Each list entry is a collection of `.Artist`` objects that are made visible on the corresponding frame. Other artists are made invisible.

`interval` : int, default: 200

Delay between frames in milliseconds.

`repeat_delay` : int, default: 0

The delay in milliseconds between consecutive animation runs, if `*repeat*` is True.

`repeat` : bool, default: True

Whether the animation repeats when the sequence of frames is completed.

`blit` : bool, default: False

Whether blitting is used to optimize drawing.

matplotlib.animation.FFMpegBase

```
FFMpegBase()
```

Mixin class for FFMpeg output.

This is a base class for the concrete ``FFMpegWriter`` and ``FFMpegFileWriter`` classes.

matplotlib.animation.FFMpegFileWriter

```
FFMpegFileWriter(*args, **kwargs)
```

File-based ffmpeg writer.

Frames are written to temporary files on disk and then stitched together at the end.

This effectively works as a slideshow input to ffmpeg with the fps passed as ```-framerate```, so see also ``their notes on frame rates`_` for further details.

.. `_`their notes on frame rates: <https://trac.ffmpeg.org/wiki/Slideshow#Framerates>

matplotlib.animation.FFMpegWriter

```
FFMpegWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
```

Pipe-based ffmpeg writer.

Frames are streamed directly to ffmpeg via a pipe and written in a single pass.

This effectively works as a slideshow input to ffmpeg with the fps passed as ```-framerate```, so see also ``their notes on frame rates`_` for further details.

.. `_`their notes on frame rates: <https://trac.ffmpeg.org/wiki/Slideshow#Framerates>

matplotlib.animation.FileMovieWriter

```
FileMovieWriter(*args, **kwargs)
```

`MovieWriter` for writing to individual files and stitching at the end.

This must be sub-classed to be useful.

matplotlib.animation.FuncAnimation

```
FuncAnimation(fig, func, frames=None, init_func=None, fargs=None, save_count=None, *,
               cache_frame_data=True, **kwargs)
```

`TimedAnimation` subclass that makes an animation by repeatedly calling a function `*func*`.

.. note::

You must store the created Animation in a variable that lives as long as the animation should run. Otherwise, the Animation object will be garbage-collected and the animation stops.

Parameters

`fig` : `~matplotlib.figure.Figure``

The figure object used to get needed events, such as draw or resize.

`func` : callable

The function to call at each frame. The first argument will be the next value in `*frames*`. Any additional positional arguments can be supplied using ``functools.partial`` or via the `*fargs*` parameter.

The required signature is::

```
def func(frame, *fargs) -> iterable_of_artists
```

It is often more convenient to provide the arguments using ``functools.partial``. In this way it is also possible to pass keyword arguments. To pass a function with both positional and keyword arguments, set all arguments as keyword arguments, just leaving the `*frame*` argument unset::

```
def func(frame, art, *, y=None):
```

```
...
```

```
ani = FuncAnimation(fig, partial(func, art=ln, y='foo'))
```

If ``blit == True``, `*func*` must return an iterable of all artists that were modified or created. This information is used by the blitting algorithm to determine which parts of the figure have to be updated. The return value is unused if ``blit == False`` and may be omitted in that case.

`frames` : iterable, int, generator function, or None, optional
Source of data to pass `*func*` and each frame of the animation

- If an iterable, then simply use the values provided. If the iterable has a length, it will override the `*save_count*` kwarg.

- If an integer, then equivalent to passing `range(frames)`

- If a generator function, then must have the signature::

```
def gen_function() -> obj
```

- If `*None*`, then equivalent to passing `itertools.count`.

In all of these cases, the values in `*frames*` is simply passed through to the user-supplied `*func*` and thus can be of any type.

`init_func` : callable, optional

A function used to draw a clear frame. If not given, the results of drawing from the first item in the frames sequence will be used. This function will be called once before the first frame.

The required signature is::

```
def init_func() -> iterable_of_artists
```

If `blit == True`, `*init_func*` must return an iterable of artists to be re-drawn. This information is used by the blitting algorithm to determine which parts of the figure have to be updated. The return value is unused if `blit == False` and may be omitted in that case.

`fargs` : tuple or None, optional

Additional arguments to pass to each call to `*func*`. Note: the use of `functools.partial` is preferred over `*fargs*`. See `*func*` for details.

`save_count` : int, optional

Fallback for the number of values from `*frames*` to cache. This is only used if the number of frames cannot be inferred from `*frames*`, i.e. when it's an iterator without length or a generator.

`interval` : int, default: 200

Delay between frames in milliseconds.

`repeat_delay` : int, default: 0

The delay in milliseconds between consecutive animation runs, if `*repeat*` is True.

`repeat` : bool, default: True

Whether the animation repeats when the sequence of frames is completed.

`blit` : bool, default: False

Whether blitting is used to optimize drawing. Note: when using blitting, any animated artists will be drawn according to their zorder; however, they will be drawn on top of any previous artists, regardless of their zorder.

`cache_frame_data` : bool, default: True

Whether frame data is cached. Disabling cache might be helpful when frames contain large objects.

matplotlib.animation.HTMLWriter

```
HTMLWriter(fps=30, codec=None, bitrate=None, extra_args=None, metadata=None,
            embed_frames=False, default_mode='loop', embed_limit=None)
```

Writer for JavaScript-based HTML movies.

matplotlib.animation.ImageMagickBase

```
ImageMagickBase()
```

Mixin class for ImageMagick output.

This is a base class for the concrete `ImageMagickWriter` and `ImageMagickFileWriter` classes, which define an `input_names` attribute (or property) specifying the input names passed to ImageMagick.

matplotlib.animation.ImageMagickFileWriter

```
ImageMagickFileWriter(*args, **kwargs)
```

File-based animated gif writer.

Frames are written to temporary files on disk and then stitched together at the end.

matplotlib.animation.ImageMagickWriter

```
ImageMagickWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
```

Pipe-based animated gif writer.

Frames are streamed directly to ImageMagick via a pipe and written in a single pass.

matplotlib.animation.MovieWriter

```
MovieWriter(fps=5, codec=None, bitrate=None, extra_args=None, metadata=None)
```

Base class for writing movies.

This is a base class for `MovieWriter` subclasses that write a movie frame data to a pipe. You cannot instantiate this class directly. See examples for how to use its subclasses.

Attributes

`frame_format` : str

The format used in writing frame data, defaults to 'rgba'.

`fig` : `~matplotlib.figure.Figure``

The figure to capture data from.

This must be provided by the subclasses.

matplotlib.animation.MovieWriterRegistry

```
MovieWriterRegistry()
```

Registry of available writer classes by human readable name.

matplotlib.animation.PillowWriter

```
PillowWriter(fps=5, metadata=None, codec=None, bitrate=None)
```

Abstract base class for writing movies, providing a way to grab frames by calling `~AbstractMovieWriter.grab_frame``.

``setup`` is called to start the process and ``finish`` is called afterwards.
``saving`` is provided as a context manager to facilitate this process as ::

```
with moviewriter.saving(fig, outfile='myfile.mp4', dpi=100):  
    # Iterate over frames  
    moviewriter.grab_frame(**savefig_kwargs)
```

The use of the context manager ensures that ``setup`` and ``finish`` are performed as necessary.

An instance of a concrete subclass of this class can be given as the ```writer``` argument of ``Animation.save()``.

matplotlib.animation.TimedAnimation

```
TimedAnimation(fig, interval=200, repeat_delay=0, repeat=True, event_source=None,  
               *args, **kwargs)
```

``Animation`` subclass for time-based animation.

A new frame is drawn every `*interval*` milliseconds.

.. note::

You must store the created Animation in a variable that lives as long as the animation should run. Otherwise, the Animation object will be garbage-collected and the animation stops.

Parameters

`fig` : `~matplotlib.figure.Figure``

The figure object used to get needed events, such as draw or resize.

`interval` : int, default: 200

Delay between frames in milliseconds.

`repeat_delay` : int, default: 0

The delay in milliseconds between consecutive animation runs, if `*repeat*` is True.

`repeat` : bool, default: True

Whether the animation repeats when the sequence of frames is completed.

`blit` : bool, default: False

Whether blitting is used to optimize drawing.

matplotlib.animation.adjusted_figsize

```
adjusted_figsize(w, h, dpi, n)
```

Compute figure size so that pixels are a multiple of n.

Parameters

w, h : float
Size in inches.

dpi : float
The dpi.

n : int
The target multiple.

Returns

wnew, hnew : float
The new figure size in inches.

matplotlib.artist

```
artist(...)
```

No description available.

matplotlib.artist.Artist

```
Artist()
```

Abstract base class for objects that render into a FigureCanvas.

Typically, all visible elements in a figure are subclasses of Artist.

matplotlib.artist.ArtistInspector

```
ArtistInspector(o)
```

A helper class to inspect an `~matplotlib.artist.Artist` and return information about its settable properties and their current values.

matplotlib.artist.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" ``[[xmin, ymin], [xmax, ymax]]``.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" ``(xmin, ymin, xmax, ymax)``

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" ``(xmin, ymin, width, height)``.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting ``ignore=True`` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify ``ignore`` explicitly. If not, the default value of ``ignore`` can be changed at any time by code with access to your Bbox, for example using the method ``~.Bbox.ignore``.

****Properties of the ``null`` bbox****

.. note::

The current behavior of ``Bbox.null()`` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may

change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[ -inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[ -inf, -inf], [inf, inf]])
```

matplotlib.artist.BboxBase

```
BboxBase(shorthand_name=None)
```

The base class of all bounding boxes.

This class is immutable; `Bbox` is a mutable subclass.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

matplotlib.artist.IdentityTransform

```
IdentityTransform(*args, **kwargs)
```

A special class that does one thing, the identity transform, in a fast way.

matplotlib.artist.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- `*vertices*`: an (N, 2) float array of vertices
- `*codes*`: an N-length `numpy.uint8` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three `'CURVE4'` codes.

The code types are:

- ``STOP`` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- ``MOVETO`` : 1 vertex

Pick up the pen and move to the given vertex.

- ``LINETO`` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'.

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.artist.Transform

`Transform(shorthand_name=None)`

The base class of all ``TransformNode`` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of ``Affine2D``.

Subclasses of this class should override the following members (at minimum):

- :attr:``input_dims``
- :attr:``output_dims``
- :meth:``transform``
- :meth:``inverted`` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr:``is_separable`` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr:``has_inverse`` (defaults to True if :meth:``inverted`` is overridden, False otherwise)

If the transform needs to do something non-standard with `matplotlib.path.Path`` objects, such as adding curves where there were once line segments, it should override:

- :meth:``transform_path``

matplotlib.artist.TransformedBbox

```
TransformedBbox(bbox, transform, **kwargs)
```

A `Bbox`` that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.

matplotlib.artist.TransformedPatchPath

```
TransformedPatchPath(patch)
```

A `TransformedPatchPath`` caches a non-affine transformed copy of the `~.patches.Patch``. This cached copy is automatically updated when the non-affine part of the transform or the patch changes.

matplotlib.artist.TransformedPath

```
TransformedPath(path, transform)
```

A `TransformedPath`` caches a non-affine transformed copy of the `~.path.Path``. This cached copy is automatically updated when the non-affine part of the transform changes.

.. note::

Paths are considered immutable by this class. Any update to the path's vertices/codes will not trigger a transform recomputation.

matplotlib.artist.allow_rasterization

```
allow_rasterization(draw)
```

Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependent renderer attributes or making

other setup function calls, such as starting and flushing a mixed-mode renderer.

matplotlib.artist.get

```
get(obj, property=None)
```

Return the value of an `Artist`'s `property`, or print all of them.

Parameters

`obj` : `~matplotlib.artist.Artist`

The queried artist; e.g., a `Line2D`, a `Text`, or an `axes.Axes`.

`property` : str or None, default: None

If `property` is 'somename', this function returns

`obj.get_somename()`.

If it's None (or unset), it *prints* all gettable properties from `obj`. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

See Also

setp

matplotlib.artist.getp

```
getp(obj, property=None)
```

Return the value of an `Artist`'s `property`, or print all of them.

Parameters

`obj` : `~matplotlib.artist.Artist`

The queried artist; e.g., a `Line2D`, a `Text`, or an `axes.Axes`.

`property` : str or None, default: None

If `property` is 'somename', this function returns

`obj.get_somename()`.

If it's None (or unset), it *prints* all gettable properties from `obj`. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

See Also

setp

matplotlib.artist.kwdoc

```
kwdoc(artist)
```

Inspect an `~matplotlib.artist.Artist`` class (using ``.ArtistInspector``) and return information about its settable properties and their current values.

Parameters

artist : `~matplotlib.artist.Artist`` or an iterable of ``Artist`s`

Returns

str

The settable properties of `*artist*`, as plain text if `:rc:`docstring.hardcopy`` is False and as a rst table (intended for use in Sphinx) if it is True.

matplotlib.artist.setp

```
setp(obj, *args, file=None, **kwargs)
```

Set one or more properties on an ``.Artist``, or list allowed values.

Parameters

obj : `~matplotlib.artist.Artist`` or list of ``.Artist``

The artist(s) whose properties are being set or queried. When setting properties, all artists are affected; when querying the allowed values, only the first instance in the sequence is queried.

For example, two lines can be made thicker and red with a single call:

```
>>> x = arange(0, 1, 0.01)
>>> lines = plot(x, sin(2*pi*x), x, sin(4*pi*x))
>>> setp(lines, linewidth=2, color='r')
```

file : file-like, default: ``sys.stdout``

Where ``setp`` writes its output when asked to list allowed values.

```
>>> with open('output.log') as file:
...     setp(line, file=file)
```

The default, ```None```, means ``sys.stdout``.

`*args, **kwargs`

The properties to set. The following combinations are supported:

- Set the linestyle of a line to be dashed:

```
>>> line, = plot([1, 2, 3])
>>> setp(line, linestyle='--')
```

- Set multiple properties at once:

```
>>> setp(line, linewidth=2, color='r')
```

- List allowed values for a line's linestyle:

```
>>> setp(line, 'linestyle')
linestyle: {'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}
```

- List all properties that can be set, and their allowed values:

```
>>> setp(line)
agg_filter: a filter function, ...
[long output listing omitted]
```

``setp`` also supports MATLAB style string/value pairs. For example, the following are equivalent:

```
>>> setp(lines, 'linewidth', 2, 'color', 'r') # MATLAB style
>>> setp(lines, linewidth=2, color='r') # Python style
```

See Also

`getp`

matplotlib.axes

```
axes(...)
```

No description available.

matplotlib.axes.Axes

```
Axes(fig, *args, facecolor=None, frameon=True, sharex=None, sharey=None, label='',
xscale=None, yscale=None, box_aspect=None, forward_navigation_events='auto', **kwargs)
```

An `Axes` object encapsulates all the elements of an individual (sub-)plot in a figure.

It contains most of the (sub-)plot elements: ``~.axis.Axis``, ``~.axis.Tick``, ``~.lines.Line2D``, ``~.text.Text``, ``~.patches.Polygon``, etc., and sets the coordinate system.

Like all visible elements in a figure, `Axes` is an ``Artist`` subclass.

The ``Axes`` instance supports callbacks through a `callbacks` attribute which is a ``~.cbook.CallbackRegistry`` instance. The events you can connect to

are 'xlim_changed' and 'ylim_changed' and the callback will be called with `func(*ax*)` where `*ax*` is the ``Axes`` instance.

.. note::

As a user, you do not instantiate Axes directly, but use Axes creation methods instead; e.g. from ``pyplot`` or ``Figure``:

``~.pyplot.subplots``, ``~.pyplot.subplot_mosaic`` or ``Figure.add_axes``.

matplotlib.axes.Subplot

```
Subplot(fig, *args, facecolor=None, frameon=True, sharex=None, sharey=None, label='',
xscale=None, yscale=None, box_aspect=None, forward_navigation_events='auto', **kwargs)
```

An Axes object encapsulates all the elements of an individual (sub-)plot in a figure.

It contains most of the (sub-)plot elements: ``~.axis.Axis``, ``~.axis.Tick``, ``~.lines.Line2D``, ``~.text.Text``, ``~.patches.Polygon``, etc., and sets the coordinate system.

Like all visible elements in a figure, Axes is an ``Artist`` subclass.

The ``Axes`` instance supports callbacks through a `callbacks` attribute which is a ``~.cbook.CallbackRegistry`` instance. The events you can connect to are 'xlim_changed' and 'ylim_changed' and the callback will be called with `func(*ax*)` where `*ax*` is the ``Axes`` instance.

.. note::

As a user, you do not instantiate Axes directly, but use Axes creation methods instead; e.g. from ``pyplot`` or ``Figure``:

``~.pyplot.subplots``, ``~.pyplot.subplot_mosaic`` or ``Figure.add_axes``.

matplotlib.axes.SubplotBase

```
SubplotBase()
```

No description available.

matplotlib.axes.subplot_class_factory

```
subplot_class_factory(cls)
```

No description available.

matplotlib.axis

```
axis(...)
```

Classes for the ticks and x- and y-axis.

matplotlib.axis.Axis

```
Axis(axes, *, pickradius=15, clear=True)
```

Base class for `.XAxis`` and `.YAxis``.

Attributes

`isDefault_label` : bool

`axes` : `~matplotlib.axes.Axes``

The `~.axes.Axes`` to which the Axis belongs.

`major` : `~matplotlib.axis.Ticker``

Determines the major tick positions and their label format.

`minor` : `~matplotlib.axis.Ticker``

Determines the minor tick positions and their label format.

`callbacks` : `~matplotlib.cbook.CallbackRegistry``

`label` : `~matplotlib.text.Text``

The axis label.

`labelpad` : float

The distance between the axis label and the tick labels.

Defaults to `:rc:`axes.labelpad``.

`offsetText` : `~matplotlib.text.Text``

A `.Text`` object containing the data offset of the ticks (if any).

`pickradius` : float

The acceptance radius for containment tests. See also `.Axis.contains``.

`majorTicks` : list of `.Tick``

The major ticks.

.. warning::

Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that changes to individual ticks will not survive if you work on the figure further (including also panning/zooming on a displayed figure).

Working on the individual ticks is a method of last resort.

Use `.set_tick_params`` instead if possible.

`minorTicks` : list of `.Tick``

The minor ticks.

matplotlib.axis.Tick

```
Tick(axes, loc, *, size=None, width=None, color=None, tickdir=None, pad=None,
      labelsz=None, labelcolor=None, labelfontfamily=None, zorder=None, gridOn=None,
      tick1On=True, tick2On=True, label1On=True, label2On=False, major=True,
      labelrotation=0, grid_color=None, grid_linestyle=None, grid_linewidth=None,
      grid_alpha=None, **kwargs)
```

Abstract base class for the axis ticks, grid lines and labels.

Ticks mark a position on an Axis. They contain two lines as markers and two labels; one each for the bottom and top positions (in case of an `.XAxis``) or for the left and right positions (in case of a `.YAxis``).

Attributes

tick1line : ``~matplotlib.lines.Line2D``
The left/bottom tick marker.
tick2line : ``~matplotlib.lines.Line2D``
The right/top tick marker.
gridline : ``~matplotlib.lines.Line2D``
The grid line associated with the label position.
label1 : ``~matplotlib.text.Text``
The left/bottom tick label.
label2 : ``~matplotlib.text.Text``
The right/top tick label.

matplotlib.axis.Ticker

```
Ticker()
```

A container for the objects defining tick position and format.

Attributes

locator : ``~matplotlib.ticker.Locator`` subclass
Determines the positions of the ticks.
formatter : ``~matplotlib.ticker.Formatter`` subclass
Determines the format of the tick labels.

matplotlib.axis.XAxis

```
XAxis(*args, **kwargs)
```

Base class for ``XAxis`` and ``YAxis``.

Attributes

isDefault_label : bool

axes : ``~matplotlib.axes.Axes``
The ``~matplotlib.axes.Axes`` to which the Axis belongs.
major : ``~matplotlib.axis.Ticker``
Determines the major tick positions and their label format.
minor : ``~matplotlib.axis.Ticker``
Determines the minor tick positions and their label format.
callbacks : ``~matplotlib.cbook.CallbackRegistry``

label : ``~matplotlib.text.Text``
The axis label.
labelpad : float
The distance between the axis label and the tick labels.
Defaults to `:rc:`axes.labelpad``.
offsetText : ``~matplotlib.text.Text``
A ``Text`` object containing the data offset of the ticks (if any).
pickradius : float
The acceptance radius for containment tests. See also ``Axis.contains``.
majorTicks : list of ``Tick``

The major ticks.

.. warning::

Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that changes to individual ticks will not survive if you work on the figure further (including also panning/zooming on a displayed figure).

Working on the individual ticks is a method of last resort. Use ``.set_tick_params`` instead if possible.

minorTicks : list of ``.Tick``
The minor ticks.

matplotlib.axis.XTick

```
XTick(*args, **kwargs)
```

Contains all the Artists needed to make an x tick - the tick line, the label text and the grid line

matplotlib.axis.YAxis

```
YAxis(*args, **kwargs)
```

Base class for ``.XAxis`` and ``.YAxis``.

Attributes

isDefault_label : bool

axes : ``.~matplotlib.axes.Axes``

The ``.axes.Axes`` to which the Axis belongs.

major : ``.~matplotlib.axis.Ticker``

Determines the major tick positions and their label format.

minor : ``.~matplotlib.axis.Ticker``

Determines the minor tick positions and their label format.

callbacks : ``.~matplotlib.cbook.CallbackRegistry``

label : ``.~matplotlib.text.Text``

The axis label.

labelpad : float

The distance between the axis label and the tick labels.

Defaults to `:rc:`axes.labelpad``.

offsetText : ``.~matplotlib.text.Text``

A ``.Text`` object containing the data offset of the ticks (if any).

pickradius : float

The acceptance radius for containment tests. See also ``.Axis.contains``.

majorTicks : list of ``.Tick``

The major ticks.

.. warning::

Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that changes to individual ticks will not survive if you work on the figure further (including also panning/zooming on a displayed figure).

Working on the individual ticks is a method of last resort. Use ``.set_tick_params`` instead if possible.

`minorTicks` : list of ``.Tick``
The minor ticks.

matplotlib.axis.YTick

```
YTick(*args, **kwargs)
```

Contains all the Artists needed to make a Y tick - the tick line, the label text and the grid line

matplotlib.backend_bases

```
backend_bases(...)
```

Abstract base classes define the primitives that renderers and graphics contexts must implement to serve as a Matplotlib backend.

``.RendererBase``

An abstract base class to handle drawing/rendering operations.

``.FigureCanvasBase``

The abstraction layer that separates the ``.Figure`` from the backend specific details like a user interface drawing area.

``.GraphicsContextBase``

An abstract base class that provides color, line styles, etc.

``.Event``

The base class for all of the Matplotlib event handling. Derived classes such as ``.KeyEvent`` and ``.MouseEvent`` store the meta data like keys and buttons pressed, x and y locations in pixel and ``.~.axes.Axes`` coordinates.

``.ShowBase``

The base class for the ```Show``` class of each interactive backend; the 'show' callable is then set to ```Show.__call__```.

``.ToolContainerBase``

The base class for the Toolbar class of each interactive backend.

matplotlib.backend_bases.Affine2D

```
Affine2D(matrix=None, **kwargs)
```

A mutable 2D affine transformation.

matplotlib.backend_bases.CapStyle

```
CapStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the two endpoints (caps) of an unclosed line are drawn.

How to draw the start and end points of lines that represent a closed curve (i.e. that end in a `~.path.Path.CLOSEPOLY`) is controlled by the line's `JoinStyle`. For all other lines, how the start and end points are drawn is controlled by the `*CapStyle*`.

For a visual impression of each `*CapStyle*`, view these docs online `<CapStyle>` or run `CapStyle.demo`.

By default, `~.backend_bases.GraphicsContextBase` draws a stroked line as squared off at its endpoints.

Supported values:

.. rst-class:: value-list

'butt'

the line is squared off at its endpoint.

'projecting'

the line is squared off as in `*butt*`, but the filled in area extends beyond the endpoint a distance of `linewidth/2`.

'round'

like `*butt*`, but a semicircular cap is added to the end of the line, of radius `linewidth/2`.

.. plot::

:alt: Demo of possible CapStyle's

```
from matplotlib._enums import CapStyle
CapStyle.demo()
```

matplotlib.backend_bases.CloseEvent

```
CloseEvent(name, canvas, guiEvent=None)
```

An event triggered by a figure being closed.

matplotlib.backend_bases.ConstrainedLayoutEngine

```
ConstrainedLayoutEngine(*, h_pad=None, w_pad=None, hspace=None, wspace=None, rect=(0,
0, 1, 1), compress=False, **kwargs)
```

Implements the `constrained_layout` geometry management. See `:ref:constrainedlayout_guide` for details.

matplotlib.backend_bases.DrawEvent

```
DrawEvent(name, canvas, renderer)
```

An event triggered by a draw operation on the canvas.

In most backends, callbacks subscribed to this event will be fired after the rendering is complete but before the screen is updated. Any extra artists drawn to the canvas's renderer will be reflected without an explicit call to ``blit``.

.. warning::

Calling ``canvas.draw`` and ``canvas.blit`` in these callbacks may not be safe with all backends and may cause infinite recursion.

A DrawEvent has a number of special attributes in addition to those defined by the parent ``Event`` class.

Attributes

renderer : ``RendererBase``

The renderer for the draw event.

matplotlib.backend_bases.Event

```
Event(name, canvas, guiEvent=None)
```

A Matplotlib event.

The following attributes are defined and shown with their default values. Subclasses may define additional attributes.

Attributes

name : str

The event name.

canvas : ``FigureCanvasBase``

The backend-specific canvas instance generating the event.

guiEvent

The GUI event that triggered the Matplotlib event.

matplotlib.backend_bases.FigureCanvasBase

```
FigureCanvasBase(figure=None)
```

The canvas the figure renders into.

Attributes

figure : ``~matplotlib.figure.Figure``

A high-level figure instance.

matplotlib.backend_bases.FigureManagerBase

```
FigureManagerBase(canvas, num)
```

A backend-independent abstraction of a figure container and controller.

The figure manager is used by pyplot to interact with the window in a backend-independent way. It's an adapter for the real (GUI) framework that represents the visual figure on screen.

The figure manager is connected to a specific canvas instance, which in turn is connected to a specific figure instance. To access a figure manager for a given figure in user code, you typically use `fig.canvas.manager`.

GUI backends derive from this class to translate common operations such as `*show*` or `*resize*` to the GUI-specific code. Non-GUI backends do not support these operations and can just use the base class.

The following basic operations are accessible:

****Window operations****

- `~.FigureManagerBase.show``
- `~.FigureManagerBase.destroy``
- `~.FigureManagerBase.full_screen_toggle``
- `~.FigureManagerBase.resize``
- `~.FigureManagerBase.get_window_title``
- `~.FigureManagerBase.set_window_title``

****Key and mouse button press handling****

The figure manager sets up default key and mouse button press handling by hooking up the `~.key_press_handler`` to the matplotlib event system. This ensures the same shortcuts and mouse actions across backends.

****Other operations****

Subclasses will have additional attributes and functions to access additional functionality. This is of course backend-specific. For example, most GUI backends have ```window``` and ```toolbar``` attributes that give access to the native GUI widgets of the respective framework.

Attributes

`canvas : `FigureCanvasBase``

The backend-specific canvas instance.

`num : int or str`

The figure number.

`key_press_handler_id : int`

The default key handler cid, when using the toolmanager.

To disable the default key press handling use::

```
figure.canvas.mpl_disconnect(
figure.canvas.manager.key_press_handler_id)
```

`button_press_handler_id : int`

The default mouse button handler cid, when using the toolmanager.

To disable the default button press handling use::

```
figure.canvas.mpl_disconnect(  
figure.canvas.manager.button_press_handler_id)
```

matplotlib.backend_bases.Gcf

```
Gcf()
```

Singleton to maintain the relation between figures and their managers, and keep track of and "active" figure and manager.

The canvas of a figure created through pyplot is associated with a figure manager, which handles the interaction between the figure and the backend. pyplot keeps track of figure managers using an identifier, the "figure number" or "manager number" (which can actually be any hashable value); this number is available as the `:attr:`number`` attribute of the manager.

This class is never instantiated; it consists of an ``OrderedDict`` mapping figure/manager numbers to managers, and a set of class methods that manipulate this ``OrderedDict``.

Attributes

`figs` : `OrderedDict`

``OrderedDict`` mapping numbers to managers; the active manager is at the end.

matplotlib.backend_bases.GraphicsContextBase

```
GraphicsContextBase()
```

An abstract base class that provides color, line styles, etc.

matplotlib.backend_bases.JoinStyle

```
JoinStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,  
boundary=None)
```

Define how the connection between two line segments is drawn.

For a visual impression of each `*JoinStyle*`, ``view these docs online <JoinStyle>``, or run ``JoinStyle.demo``.

Lines in Matplotlib are typically defined by a 1D ``~.path.Path`` and a finite ``linewidth``, where the underlying 1D ``~.path.Path`` represents the center of the stroked line.

By default, ``~.backend_bases.GraphicsContextBase`` defines the boundaries of a stroked line to simply be every point within some radius, ``linewidth/2``, away from any point of the center line. However, this results in corners appearing "rounded", which may not be the desired behavior if you are drawing, for example, a polygon or pointed star.

****Supported values:****

.. rst-class:: value-list

'miter'

the "arrow-tip" style. Each boundary of the filled-in area will extend in a straight line parallel to the tangent vector of the centerline at the point it meets the corner, until they meet in a sharp point.

'round'

stokes every point within a radius of ``linewidth/2`` of the center lines.

'bevel'

the "squared-off" style. It can be thought of as a rounded corner where the "circular" part of the corner has been cut off.

.. note::

Very long miter tips are cut off (to form a *bevel*) after a backend-dependent limit called the "miter limit", which specifies the maximum allowed ratio of miter length to line width. For example, the PDF backend uses the default value of 10 specified by the PDF standard, while the SVG backend does not even specify the miter limit, resulting in a default value of 4 per the SVG specification. Matplotlib does not currently allow the user to adjust this parameter.

A more detailed description of the effect of a miter limit can be found in the `Mozilla Developer Docs

<<https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-miterlimit>>`_

.. plot::

:alt: Demo of possible JoinStyle's

```
from matplotlib._enums import JoinStyle
JoinStyle.demo()
```

matplotlib.backend_bases.KeyEvent

```
KeyEvent(name, canvas, key, x=0, y=0, guiEvent=None)
```

A key event (key press, key release).

A KeyEvent has a number of special attributes in addition to those defined by the parent `Event` and `LocationEvent` classes.

Attributes

key : None or str

The key(s) pressed. Could be *None*, a single case sensitive Unicode character ("g", "G", "#", etc.), a special key ("control", "shift", "f1", "up", etc.) or a combination of the above (e.g., "ctrl+alt+g", "ctrl+alt+G").

Notes

Modifier keys will be prefixed to the pressed key and will be in the order "ctrl", "alt", "super". The exception to this rule is when the pressed key

is itself a modifier key, therefore "ctrl+alt" and "alt+control" can both be valid key values.

Examples

::

```
def on_key(event):
    print('you pressed', event.key, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('key_press_event', on_key)
```

matplotlib.backend_bases.LocationEvent

```
LocationEvent(name, canvas, x, y, guiEvent=None, *, modifiers=None)
```

An event that has a screen location.

A LocationEvent has a number of special attributes in addition to those defined by the parent ``Event`` class.

Attributes

x, y : int or None

Event location in pixels from bottom left of canvas.

inaxes : ``~matplotlib.axes.Axes`` or None

The ``~.axes.Axes`` instance over which the mouse is, if any.

xdata, ydata : float or None

Data coordinates of the mouse within `*inaxes*`, or `*None*` if the mouse is not over an Axes.

modifiers : frozenset

The keyboard modifiers currently being pressed (except for KeyEvent).

matplotlib.backend_bases.MouseButton

```
MouseButton(value, names=None, *, module=None, qualname=None, type=None, start=1,
             boundary=None)
```

Enum where members are also (and must be) ints

matplotlib.backend_bases.MouseEvent

```
MouseEvent(name, canvas, x, y, button=None, key=None, step=0, dblclick=False,
            guiEvent=None, *, buttons=None, modifiers=None)
```

A mouse event ('button_press_event', 'button_release_event', 'scroll_event', 'motion_notify_event').

A MouseEvent has a number of special attributes in addition to those defined by the parent ``Event`` and ``LocationEvent`` classes.

Attributes

button : None or ``MouseButton`` or {'up', 'down'}

The button pressed. 'up' and 'down' are used for scroll events.

Note that LEFT and RIGHT actually refer to the "primary" and "secondary" buttons, i.e. if the user inverts their left and right buttons ("left-handed setting") then the LEFT button will be the one physically on the right.

If this is unset, **name** is "scroll_event", and **step** is nonzero, then this will be set to "up" or "down" depending on the sign of **step**.

buttons : None or frozenset

For 'motion_notify_event', the mouse buttons currently being pressed (a set of zero or more MouseButtons);
for other events, None.

.. note::

For 'motion_notify_event', this attribute is more accurate than the ``button`` (singular) attribute, which is obtained from the last 'button_press_event' or 'button_release_event' that occurred within the canvas (and thus 1. be wrong if the last change in mouse state occurred when the canvas did not have focus, and 2. cannot report when multiple buttons are pressed).

This attribute is not set for 'button_press_event' and 'button_release_event' because GUI toolkits are inconsistent as to whether they report the button state **before** or **after** the press/release occurred.

.. warning::

On macOS, the Tk backends only report a single button even if multiple buttons are pressed.

key : None or str

The key pressed when the mouse event triggered, e.g. 'shift'.
See `KeyEvent`.

.. warning::

This key is currently obtained from the last 'key_press_event' or 'key_release_event' that occurred within the canvas. Thus, if the last change of keyboard state occurred while the canvas did not have focus, this attribute will be wrong. On the other hand, the ``modifiers`` attribute should always be correct, but it can only report on modifier keys.

step : float

The number of scroll steps (positive for 'up', negative for 'down').
This applies only to 'scroll_event' and defaults to 0 otherwise.

dblclick : bool

Whether the event is a double-click. This applies only to 'button_press_event' and is False otherwise. In particular, it's not used in 'button_release_event'.

Examples

::

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('button_press_event', on_press)
```

matplotlib.backend_bases.NavigationToolbar2

`NavigationToolbar2(canvas)`

Base class for the navigation cursor, version 2.

Backends must implement a canvas that handles connections for 'button_press_event' and 'button_release_event'. See :meth:`FigureCanvasBase.mpl_connect` for more information.

They must also define

:meth:`save_figure`
Save the current figure.

:meth:`draw_rubberband` (optional)
Draw the zoom to rect "rubberband" rectangle.

:meth:`set_message` (optional)
Display message.

:meth:`set_history_buttons` (optional)
You can change the history back / forward buttons to indicate disabled / enabled state.

and override ``__init__`` to set up the toolbar -- without forgetting to call the base-class init. Typically, ``__init__`` needs to set up toolbar buttons connected to the 'home', 'back', 'forward', 'pan', 'zoom', and 'save_figure' methods and using standard icons in the "images" subdirectory of the data path.

That's it, we'll do the rest!

matplotlib.backend_bases.NonGuiException

`NonGuiException(...)`

Raised when trying show a figure in a non-GUI backend.

matplotlib.backend_bases.Path

`Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)`

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- `*vertices*`: an (N, 2) float array of vertices

- `*codes*`: an N-length ``numpy.uint8`` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three ``CURVE4`` codes.

The code types are:

- ``STOP`` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- ``MOVETO`` : 1 vertex

Pick up the pen and move to the given vertex.

- ``LINETO`` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.backend_bases.PickEvent

```
PickEvent(name, canvas, mouseevent, artist, guiEvent=None, **kwargs)
```

A pick event.

This event is fired when the user picks a location on the canvas

sufficiently close to an artist that has been made pickable with
``Artist.set_picker``.

A `PickEvent` has a number of special attributes in addition to those defined by the parent ``Event`` class.

Attributes

`mouseevent` : ``MouseEvent``

The mouse event that generated the pick.

`artist` : ``~matplotlib.artist.Artist``

The picked artist. Note that artists are not pickable by default (see ``Artist.set_picker``).

`other`

Additional attributes may be present depending on the type of the picked object; e.g., a ``Line2D`` pick may define different extra attributes than a ``PatchCollection`` pick.

Examples

Bind a function ```on_pick()``` to pick events, that prints the coordinates of the picked data point::

```
ax.plot(np.rand(100), 'o', picker=5) # 5 points tolerance
```

```
def on_pick(event):  
    line = event.artist  
    xdata, ydata = line.get_data()  
    ind = event.ind  
    print(f'on pick line: {xdata[ind]:.3f}, {ydata[ind]:.3f}')
```

```
cid = fig.canvas.mpl_connect('pick_event', on_pick)
```

matplotlib.backend_bases.RendererBase

`RendererBase()`

An abstract base class to handle drawing/rendering operations.

The following methods must be implemented in the backend for full functionality (though just implementing ``draw_path`` alone would give a highly capable backend):

- * ``draw_path``
- * ``draw_image``
- * ``draw_gouraud_triangles``

The following methods *should* be implemented in the backend for optimization reasons:

- * ``draw_text``
- * ``draw_markers``
- * ``draw_path_collection``
- * ``draw_quad_mesh``

matplotlib.backend_bases.ResizeEvent

```
ResizeEvent(name, canvas)
```

An event triggered by a canvas resize.

A `ResizeEvent` has a number of special attributes in addition to those defined by the parent `Event` class.

Attributes

`width` : int

Width of the canvas in pixels.

`height` : int

Height of the canvas in pixels.

matplotlib.backend_bases.ShowBase

```
ShowBase()
```

Simple base class to generate a `show()` function in backends.

Subclass must override `mainloop()` method.

matplotlib.backend_bases.TexManager

```
TexManager()
```

Convert strings to dvi files using TeX, caching the results to a directory.

The cache directory is called `tex.cache` and is located in the directory returned by `.get_cachedir`.

Repeated calls to this constructor always return the same instance.

matplotlib.backend_bases.TimerBase

```
TimerBase(interval=None, callbacks=None)
```

A base class for providing timer events, useful for things animations.

Backends need to implement a few specific methods in order to use their own timing mechanisms so that the timer events are integrated into their event loops.

Subclasses must override the following methods:

- `_timer_start`: Backend-specific code for starting the timer.
- `_timer_stop`: Backend-specific code for stopping the timer.

Subclasses may additionally override the following methods:

- `_timer_set_single_shot`: Code for setting the timer to single shot operating mode, if supported by the timer object. If not, the `Timer` class itself will store the flag and the `_on_timer` method should be overridden to support such behavior.

- ``_timer_set_interval``: Code for setting the interval on the timer, if there is a method for doing so on the timer object.

- ``_on_timer``: The internal function that any timer object should call, which will handle the task of running all callbacks that have been set.

matplotlib.backend_bases.ToolContainerBase

```
ToolContainerBase(toolmanager)
```

Base class for all tool containers, e.g. toolbars.

Attributes

toolmanager : ``ToolManager``

The tools with which this ``ToolContainer`` wants to communicate.

matplotlib.backend_bases.ToolManager

```
ToolManager(figure=None)
```

Manager for actions triggered by user interactions (key press, toolbar clicks, ...) on a Figure.

Attributes

figure : ``Figure``

keypresslock : ``~matplotlib.widgets.LockDraw``

``LockDraw`` object to know if the ``canvas`` key_press_event is locked.

messagelock : ``~matplotlib.widgets.LockDraw``

``LockDraw`` object to know if the message is available to write.

matplotlib.backend_bases.button_press_handler

```
button_press_handler(event, canvas=None, toolbar=None)
```

The default Matplotlib button actions for extra mouse buttons.

Parameters are as for ``key_press_handler``, except that *event* is a ``MouseEvent``.

matplotlib.backend_bases.cursors

```
cursors(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
```

Backend-independent cursor types.

matplotlib.backend_bases.get_registered_canvas_class

```
get_registered_canvas_class(format)
```

Return the registered default canvas for given file format.
Handles deferred import of required backend.

matplotlib.backend_bases.is_interactive

```
is_interactive()
```

Return whether to redraw after every plotting command.

.. note::

This function is only intended for use in backends. End users should use ``pyplot.isinteractive`` instead.

matplotlib.backend_bases.key_press_handler

```
key_press_handler(event, canvas=None, toolbar=None)
```

Implement the default Matplotlib key bindings for the canvas and toolbar described at :ref:`key-event-handling`.

Parameters

event : ``KeyEvent``

A key press/release event.

canvas : ``FigureCanvasBase``, default: ```event.canvas```

The backend-specific canvas instance. This parameter is kept for back-compatibility, but, if set, should always be equal to ```event.canvas```.

toolbar : ``NavigationToolbar2``, default: ```event.canvas.toolbar```

The navigation cursor toolbar. This parameter is kept for back-compatibility, but, if set, should always be equal to ```event.canvas.toolbar```.

matplotlib.backend_bases.register_backend

```
register_backend(format, backend, description=None)
```

Register a backend for saving to a given file format.

Parameters

format : str

File extension

backend : module string or canvas class

Backend for handling file output

description : str, default: ""

Description of the file type.

matplotlib.backend_managers

```
backend_managers(...)
```

No description available.

matplotlib.backend_managers.ToolEvent

```
ToolEvent(name, sender, tool, data=None)
```

Event for tool manipulation (add/remove).

matplotlib.backend_managers.ToolManager

```
ToolManager(figure=None)
```

Manager for actions triggered by user interactions (key press, toolbar clicks, ...) on a Figure.

Attributes

figure : ``Figure``

keypresslock : ``~matplotlib.widgets.LockDraw``

``LockDraw`` object to know if the ``canvas`` `key_press_event` is locked.

messagelock : ``~matplotlib.widgets.LockDraw``

``LockDraw`` object to know if the message is available to write.

matplotlib.backend_managers.ToolManagerMessageEvent

```
ToolManagerMessageEvent(name, sender, message)
```

Event carrying messages from toolmanager.

Messages usually get displayed to the user by the toolbar.

matplotlib.backend_managers.ToolTriggerEvent

```
ToolTriggerEvent(name, sender, tool, canvasevent=None, data=None)
```

Event to inform that a tool has been triggered.

matplotlib.backend_tools

```
backend_tools(...)
```

Abstract base classes define the primitives for Tools.

These tools are used by ``matplotlib.backend_managers.ToolManager``

:class: ``ToolBase``

Simple stateless tool

:class: ``ToolToggleBase``

Tool that has two states, only one Toggle tool can be active at any given time for the same

``matplotlib.backend_managers.ToolManager``

matplotlib.backend_tools.AxisScaleBase

```
AxisScaleBase(*args, **kwargs)
```

Base Tool to toggle between linear and logarithmic.

matplotlib.backend_tools.ConfigureSubplotsBase

```
ConfigureSubplotsBase(toolmanager, name)
```

Base tool for the configuration of subplots.

matplotlib.backend_tools.Cursors

```
Cursors(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
```

Backend-independent cursor types.

matplotlib.backend_tools.Gcf

```
Gcf()
```

Singleton to maintain the relation between figures and their managers, and keep track of and "active" figure and manager.

The canvas of a figure created through pyplot is associated with a figure manager, which handles the interaction between the figure and the backend. pyplot keeps track of figure managers using an identifier, the "figure number" or "manager number" (which can actually be any hashable value); this number is available as the `:attr: 'number'` attribute of the manager.

This class is never instantiated; it consists of an ``OrderedDict`` mapping figure/manager numbers to managers, and a set of class methods that manipulate this ``OrderedDict``.

Attributes

`figs` : `OrderedDict`

``OrderedDict`` mapping numbers to managers; the active manager is at the end.

matplotlib.backend_tools.RubberbandBase

```
RubberbandBase(toolmanager, name)
```

Draw and remove a rubberband.

matplotlib.backend_tools.SaveFigureBase

```
SaveFigureBase(toolmanager, name)
```

Base tool for figure saving.

matplotlib.backend_tools.ToolBack

```
ToolBack(toolmanager, name)
```

Move back up the view limits stack.

matplotlib.backend_tools.ToolBase

```
ToolBase(toolmanager, name)
```

Base tool class.

A base tool, only implements `trigger` method or no method at all.
The tool is instantiated by `matplotlib.backend_managers.ToolManager`.

matplotlib.backend_tools.ToolCopyToClipboardBase

```
ToolCopyToClipboardBase(toolmanager, name)
```

Tool to copy the figure to the clipboard.

matplotlib.backend_tools.ToolCursorPosition

```
ToolCursorPosition(*args, **kwargs)
```

Send message with the current pointer position.

This tool runs in the background reporting the position of the cursor.

matplotlib.backend_tools.ToolForward

```
ToolForward(toolmanager, name)
```

Move forward in the view lim stack.

matplotlib.backend_tools.ToolFullScreen

```
ToolFullScreen(toolmanager, name)
```

Tool to toggle full screen.

matplotlib.backend_tools.ToolGrid

```
ToolGrid(toolmanager, name)
```

Tool to toggle the major grids of the figure.

matplotlib.backend_tools.ToolHelpBase

```
ToolHelpBase(toolmanager, name)
```

Base tool class.

A base tool, only implements `trigger` method or no method at all.
The tool is instantiated by `matplotlib.backend_managers.ToolManager`.

matplotlib.backend_tools.ToolHome

```
ToolHome(toolmanager, name)
```

Restore the original view limits.

matplotlib.backend_tools.ToolMinorGrid

```
ToolMinorGrid(toolmanager, name)
```

Tool to toggle the major and minor grids of the figure.

matplotlib.backend_tools.ToolPan

```
ToolPan(*args)
```

Pan Axes with left mouse, zoom with right.

matplotlib.backend_tools.ToolQuit

```
ToolQuit(toolmanager, name)
```

Tool to call the figure manager destroy method.

matplotlib.backend_tools.ToolQuitAll

```
ToolQuitAll(toolmanager, name)
```

Tool to call the figure manager destroy method.

matplotlib.backend_tools.ToolSetCursor

```
ToolSetCursor(*args, **kwargs)
```

Change to the current cursor while inaxes.

This tool, keeps track of all `ToolToggleBase` derived tools, and updates the cursor when a tool gets triggered.

matplotlib.backend_tools.ToolToggleBase

```
ToolToggleBase(*args, **kwargs)
```

Toggleable tool.

Every time it is triggered, it switches between enable and disable.

Parameters

`**args`

Variable length argument to be used by the Tool.

`**kwargs`

``toggled`` if present and True, sets the initial state of the Tool

Arbitrary keyword arguments to be consumed by the Tool

matplotlib.backend_tools.ToolViewsPositions

```
ToolViewsPositions(*args, **kwargs)
```

Auxiliary Tool to handle changes in views and positions.

Runs in the background and should get used by all the tools that

need to access the figure's history of views and positions, e.g.

```
* `ToolZoom`  
* `ToolPan`  
* `ToolHome`  
* `ToolBack`  
* `ToolForward`
```

matplotlib.backend_tools.ToolXScale

```
ToolXScale(*args, **kwargs)
```

Tool to toggle between linear and logarithmic scales on the X axis.

matplotlib.backend_tools.ToolYScale

```
ToolYScale(*args, **kwargs)
```

Tool to toggle between linear and logarithmic scales on the Y axis.

matplotlib.backend_tools.ToolZoom

```
ToolZoom(*args)
```

A Tool for zooming using a rectangle selector.

matplotlib.backend_tools.ViewsPositionsBase

```
ViewsPositionsBase(toolmanager, name)
```

Base class for `ToolHome`, `ToolBack` and `ToolForward`.

matplotlib.backend_tools.ZoomPanBase

```
ZoomPanBase(*args)
```

Base class for `ToolZoom` and `ToolPan`.

matplotlib.backend_tools.add_tools_to_container

```
add_tools_to_container(container, tools=[['navigation', ['home', 'back', 'forward']],  
                               ['zoompan', ['pan', 'zoom', 'subplots']], ['io', ['save', 'help']]])
```

Add multiple tools to the container.

Parameters

container : Container

`matplotlib.backend_bases.ToolContainerBase` object that will get the tools added.

tools : list, optional

List in the form ``[[group1, [tool1, tool2 ...]], [group2, [...]]``

where the tools ``[tool1, tool2, ...]`` will display in group1.

See `matplotlib.backend_bases.ToolContainerBase.add_tool` for details.

matplotlib.backend_tools.add_tools_to_manager

```
add_tools_to_manager(toolmanager, tools={'home': , 'back': , 'forward': , 'zoom': ,
'pan': , 'subplots': , 'save': , 'grid': , 'grid_minor': , 'fullscreen': , 'quit': ,
'quit_all': , 'xscale': , 'yscale': , 'position': , 'viewpos': , 'cursor': ,
'rubberband': , 'help': , 'copy': })
```

Add multiple tools to a `.ToolManager``.

Parameters

`toolmanager` : `.backend_managers.ToolManager``

Manager to which the tools are added.

`tools` : {str: class_like}, optional

The tools to add in a {name: tool} dict, see

`.backend_managers.ToolManager.add_tool`` for more info.

matplotlib.backend_tools.cursors

```
cursors(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Backend-independent cursor types.

matplotlib.backends

```
backends(...)
```

No description available.

matplotlib.backends.BackendFilter

```
BackendFilter(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Filter used with `:meth:`~matplotlib.backends.registry.BackendRegistry.list_builtin``

.. versionadded:: 3.9

matplotlib.backends.backend_agg

```
backend_agg(...)
```

An ``Anti-Grain Geometry`_ (AGG)` backend.

Features that are implemented:

- * capstyles and join styles
- * dashes
- * linewidth
- * lines, rectangles, ellipses
- * clipping to a rectangle
- * output to RGBA and Pillow-supported image formats
- * alpha blending
- * DPI scaling properly - everything scales properly (dashes, linewidths, etc)

- * draw polygon
- * freetype2 w/ ft2font

Still TODO:

- * integrate screen dpi w/ ppi and text

.. _Anti-Grain Geometry: <http://agg.sourceforge.net/antigrain.com>

matplotlib.backends.backend_cairo

```
backend_cairo(...)
```

A Cairo backend for Matplotlib

=====

:Author: Steve Chaplin and others

This backend depends on cairocffi or pycairo.

matplotlib.backends.backend_mixed

```
backend_mixed(...)
```

No description available.

matplotlib.backends.backend_nbagg

```
backend_nbagg(...)
```

Interactive figures in the IPython notebook.

matplotlib.backends.backend_pdf

```
backend_pdf(...)
```

A PDF Matplotlib backend.

Author: Jouni K Seppänen <jks@iki.fi> and others.

matplotlib.backends.backend_pgf

```
backend_pgf(...)
```

No description available.

matplotlib.backends.backend_ps

```
backend_ps(...)
```

A PostScript backend, which can produce both PostScript .ps and .eps.

matplotlib.backends.backend_qt

```
backend_qt(...)
```

No description available.

matplotlib.backends.backend_qt5

```
backend_qt5(...)
```

No description available.

matplotlib.backends.backend_qt5agg

```
backend_qt5agg(...)
```

Render to qt from agg

matplotlib.backends.backend_qt5cairo

```
backend_qt5cairo(...)
```

No description available.

matplotlib.backends.backend_qtagg

```
backend_qtagg(...)
```

Render to qt from agg.

matplotlib.backends.backend_qtcairo

```
backend_qtcairo(...)
```

No description available.

matplotlib.backends.backend_svg

```
backend_svg(...)
```

No description available.

matplotlib.backends.backend_template

```
backend_template(...)
```

A fully functional, do-nothing backend intended as a template for backend writers. It is fully functional in that you can select it as a backend e.g. with ::

```
import matplotlib
matplotlib.use("template")
```

and your program will (should!) run without error, though no output is produced. This provides a starting point for backend writers; you can selectively implement drawing methods (`~.RendererTemplate.draw_path``, `~.RendererTemplate.draw_image``, etc.) and slowly see your figure come to life instead having to have a full-blown implementation before getting any results.

Copy this file to a directory outside the Matplotlib source tree, somewhere where Python can import it (by adding the directory to your ``sys.path`` or by packaging it as a normal Python package); if the backend is importable as ``import my.backend`` you can then select it using ::

```
import matplotlib
matplotlib.use("module://my.backend")
```

If your backend implements support for saving figures (i.e. has a ``print_xyz`` method), you can register it as the default handler for a given file type::

```
from matplotlib.backend_bases import register_backend
register_backend('xyz', 'my_backend', 'XYZ File Format')
...
plt.savefig("figure.xyz")
```

matplotlib.backends.backend_tkagg

```
backend_tkagg(...)
```

No description available.

matplotlib.backends.backend_tkcairo

```
backend_tkcairo(...)
```

No description available.

matplotlib.backends.backend_webagg

```
backend_webagg(...)
```

Displays Agg images in the browser, with interactivity.

matplotlib.backends.backend_webagg_core

```
backend_webagg_core(...)
```

Displays Agg images in the browser, with interactivity.

matplotlib.backends.qt_compat

```
qt_compat(...)
```

Qt binding and backend selector.

The selection logic is as follows:

- if any of PyQt6, PySide6, PyQt5, or PySide2 have already been imported (checked in that order), use it;
- otherwise, if the QT_API environment variable (used by Enthought) is set, use it to determine which binding to use;
- otherwise, use whatever the rcParams indicate.

matplotlib.backends.qt_editor

```
qt_editor(...)
```

No description available.

matplotlib.backends.registry

```
registry(...)
```

No description available.

matplotlib.bezier

```
bezier(...)
```

A module providing some utility functions regarding Bézier path manipulation.

matplotlib.bezier.BezierSegment

```
BezierSegment(control_points)
```

A d-dimensional Bézier segment.

Parameters

control_points : (N, d) array

Location of the *N* control points.

matplotlib.bezier.NonIntersectingPathException

```
NonIntersectingPathException(...)
```

Inappropriate argument value (of correct type).

matplotlib.bezier.check_if_parallel

```
check_if_parallel(dx1, dy1, dx2, dy2, tolerance=1e-05)
```

Check if two lines are parallel.

Parameters

dx1, dy1, dx2, dy2 : float

The gradients dy/dx of the two lines.

tolerance : float

The angular tolerance in radians up to which the lines are considered parallel.

Returns

is_parallel

- 1 if two lines are parallel in same direction.

- -1 if two lines are parallel in opposite direction.

- False otherwise.

matplotlib.bezier.find_bezier_t_intersecting_with_closedpath

```
find_bezier_t_intersecting_with_closedpath(bezier_point_at_t, inside_closedpath,
t0=0.0, t1=1.0, tolerance=0.01)
```

Find the intersection of the Bézier curve with a closed path.

The intersection point **t** is approximated by two parameters **t0**, **t1** such that **t0** <= **t** <= **t1**.

Search starts from **t0** and **t1** and uses a simple bisection algorithm therefore one of the end points must be inside the path while the other doesn't. The search stops when the distance of the points parametrized by **t0** and **t1** gets smaller than the given **tolerance**.

Parameters

bezier_point_at_t : callable

A function returning x, y coordinates of the Bézier at parameter **t**.

It must have the signature::

bezier_point_at_t(*t*: float) -> tuple[float, float]

inside_closedpath : callable

A function returning True if a given point (x, y) is inside the closed path. It must have the signature::

inside_closedpath(point: tuple[float, float]) -> bool

t0, *t1* : float

Start parameters for the search.

tolerance : float

Maximal allowed distance between the final points.

Returns

t0, *t1* : float

The Bézier path parameters.

matplotlib.bezier.find_control_points

```
find_control_points(c1x, c1y, mmx, mmy, c2x, c2y)
```

Find control points of the Bézier curve passing through (**c1x**, **c1y**), (**mmx**, **mmy**), and (**c2x**, **c2y**), at parametric values 0, 0.5, and 1.

matplotlib.bezier.get_cos_sin

```
get_cos_sin(x0, y0, x1, y1)
```

No description available.

matplotlib.bezier.get_intersection

```
get_intersection(cx1, cy1, cos_t1, sin_t1, cx2, cy2, cos_t2, sin_t2)
```

Return the intersection between the line through (*cx1*, *cy1*) at angle *t1* and the line through (*cx2*, *cy2*) at angle *t2*.

matplotlib.bezier.get_normal_points

```
get_normal_points(cx, cy, cos_t, sin_t, length)
```

For a line passing through (*cx*, *cy*) and having an angle *t*, return locations of the two points located along its perpendicular line at the distance of *length*.

matplotlib.bezier.get_parallels

```
get_parallels(bezier2, width)
```

Given the quadratic Bézier control points *bezier2*, returns control points of quadratic Bézier lines roughly parallel to given one separated by *width*.

matplotlib.bezier.inside_circle

```
inside_circle(cx, cy, r)
```

Return a function that checks whether a point is in a circle with center (*cx*, *cy*) and radius *r*.

The returned function has the signature::

f(xy: tuple[float, float]) -> bool

matplotlib.bezier.make_wedged_bezier2

```
make_wedged_bezier2(bezier2, width, w1=1.0, wm=0.5, w2=0.0)
```

Being similar to `get_parallels`, returns control points of two quadratic Bézier lines having a width roughly parallel to given one separated by *width*.

matplotlib.bezier.split_bezier_intersecting_with_closedpath

```
split_bezier_intersecting_with_closedpath(bezier, inside_closedpath, tolerance=0.01)
```

Split a Bézier curve into two at the intersection with a closed path.

Parameters

bezier : (N, 2) array-like

Control points of the Bézier segment. See `.BezierSegment`.

inside_closedpath : callable

A function returning True if a given point (x, y) is inside the closed path. See also `.find_bezier_t_intersecting_with_closedpath`.

tolerance : float

The tolerance for the intersection. See also
`.find_bezier_t_intersecting_with_closedpath`.

Returns

left, right

Lists of control points for the two Bézier segments.

matplotlib.bezier.split_de_casteljau

```
split_de_casteljau(beta, t)
```

Split a Bézier segment defined by its control points **beta** into two separate segments divided at **t** and return their control points.

matplotlib.bezier.split_path_inout

```
split_path_inout(path, inside, tolerance=0.01, reorder_inout=False)
```

Divide a path into two segments at the point where ``inside(x, y)`` becomes False.

matplotlib.category

```
category(...)
```

Plotting of string "category" data: ``plot(['d', 'f', 'a'], [1, 2, 3])`` will plot three points with x-axis values of 'd', 'f', 'a'.

See :doc:`/gallery/lines_bars_and_markers/categorical_variables` for an example.

The module uses Matplotlib's ``matplotlib.units`` mechanism to convert from strings to integers and provides a tick locator, a tick formatter, and the ``.UnitData`` class that creates and stores the string-to-integer mapping.

matplotlib.category.StrCategoryConverter

```
StrCategoryConverter()
```

The minimal interface for a converter to take custom data types (or sequences) and convert them to values Matplotlib can use.

matplotlib.category.StrCategoryFormatter

```
StrCategoryFormatter(units_mapping)
```

String representation of the data at every tick.

matplotlib.category.StrCategoryLocator

```
StrCategoryLocator(units_mapping)
```

Tick at every integer mapping of the string data.

matplotlib.category.UnitData

```
UnitData(data=None)
```

No description available.

matplotlib.cbook

```
cbook(...)
```

A collection of utility functions and classes. Originally, many (but not all) were from the Python Cookbook -- hence the name cbook.

matplotlib.cbook.CallbackRegistry

```
CallbackRegistry(exception_handler=, *, signals=None)
```

Handle registering, processing, blocking, and disconnecting for a set of signals and callbacks:

```
>>> def oneat(x):
...     print('eat', x)
>>> def ondrink(x):
...     print('drink', x)

>>> from matplotlib.cbook import CallbackRegistry
>>> callbacks = CallbackRegistry()

>>> id_eat = callbacks.connect('eat', oneat)
>>> id_drink = callbacks.connect('drink', ondrink)

>>> callbacks.process('drink', 123)
drink 123
>>> callbacks.process('eat', 456)
eat 456
>>> callbacks.process('be merry', 456) # nothing will be called

>>> callbacks.disconnect(id_eat)
>>> callbacks.process('eat', 456) # nothing will be called

>>> with callbacks.blocked(signal='drink'):
...     callbacks.process('drink', 123) # nothing will be called
>>> callbacks.process('drink', 123)
drink 123
```

In practice, one should always disconnect all callbacks when they are no longer needed to avoid dangling references (and thus memory leaks). However, real code in Matplotlib rarely does so, and due to its design, it is rather difficult to place this kind of code. To get around this, and prevent this class of memory leaks, we instead store weak references to bound methods only, so when the destination object needs to die, the CallbackRegistry won't keep it alive.

Parameters

`exception_handler` : callable, optional

If not None, `*exception_handler*` must be a function that takes an ``Exception`` as single parameter. It gets called with any ``Exception``

raised by the callbacks during ``CallbackRegistry.process``, and may either re-raise the exception or handle it in another manner.

The default handler prints the exception (with ``traceback.print_exc``) if an interactive event loop is running; it re-raises the exception if no interactive event loop is running.

signals : list, optional

If not None, *signals* is a list of signals that this registry handles: attempting to ``process`` or to ``connect`` to a signal not in the list throws a ``ValueError``. The default, None, does not restrict the handled signals.

matplotlib.cbook.Grouper

```
Grouper(init=())
```

A disjoint-set data structure.

Objects can be joined using `:meth:`join``, tested for connectedness using `:meth:`joined``, and all disjoint sets can be retrieved by using the object as an iterator.

The objects being joined must be hashable and weak-referenceable.

Examples

```
-----
>>> from matplotlib.cbook import Grouper
>>> class Foo:
...     def __init__(self, s):
...         self.s = s
...     def __repr__(self):
...         return self.s
...
>>> a, b, c, d, e, f = [Foo(x) for x in 'abcdef']
>>> grp = Grouper()
>>> grp.join(a, b)
>>> grp.join(b, c)
>>> grp.join(d, e)
>>> list(grp)
[[a, b, c], [d, e]]
>>> grp.joined(a, b)
True
>>> grp.joined(a, c)
True
>>> grp.joined(a, d)
False
```

matplotlib.cbook.GrouperView

```
GrouperView(grouper)
```

Immutable view over a ``Grouper``.

```
boxplot_stats(X, whis=1.5, bootstrap=None, labels=None, autorange=False)
```

Return a list of dictionaries of statistics used to draw a series of box and whisker plots using `~.Axes.bxp``.

Parameters

`X` : array-like

Data that will be represented in the boxplots. Should have 2 or fewer dimensions.

`whis` : float or (float, float), default: 1.5

The position of the whiskers.

If a float, the lower whisker is at the lowest datum above

`Q1 - whis*(Q3-Q1)``, and the upper whisker at the highest datum below

`Q3 + whis*(Q3-Q1)``, where `Q1` and `Q3` are the first and third

quartiles. The default value of `whis = 1.5`` corresponds to Tukey's original definition of boxplots.

If a pair of floats, they indicate the percentiles at which to draw the

whiskers (e.g., (5, 95)). In particular, setting this to (0, 100)

results in whiskers covering the whole range of the data.

In the edge case where `Q1 == Q3``, `*whis*` is automatically set to

(0, 100) (cover the whole range of the data) if `*autorange*` is `True`.

Beyond the whiskers, data are considered outliers and are plotted as individual points.

`bootstrap` : int, optional

Number of times the confidence intervals around the median should be bootstrapped (percentile method).

`labels` : list of str, optional

Labels for each dataset. Length must be compatible with dimensions of `*X*`.

`autorange` : bool, optional (False)

When `True`` and the data are distributed such that the 25th and 75th percentiles are equal, `whis`` is set to (0, 100) such that the whisker ends are at the minimum and maximum of the data.

Returns

list of dict

A list of dictionaries containing the results for each column of data. Keys of each dictionary are the following:

=====

Key Value Description

=====

label tick label for the boxplot

mean arithmetic mean value

med 50th percentile
q1 first quartile (25th percentile)
q3 third quartile (75th percentile)
iqr interquartile range
cilo lower notch around the median
cihi upper notch around the median
whislo end of the lower whisker
whishi end of the upper whisker
fliers outliers

=====

Notes

Non-bootstrapping approach to confidence interval uses Gaussian-based asymptotic approximation:

.. math::

$\mathrm{med} \pm 1.57 \times \frac{\mathrm{iqr}}{\sqrt{N}}$

General approach from:

McGill, R., Tukey, J.W., and Larsen, W.A. (1978) "Variations of Boxplots", The American Statistician, 32:12-16.

matplotlib.cbook.contiguous_regions

`contiguous_regions(mask)`

Return a list of (ind0, ind1) such that `mask[ind0:ind1].all()` is True and we cover all such regions.

matplotlib.cbook.delete_masked_points

`delete_masked_points(*args)`

Find all masked and/or non-finite points in a set of arguments, and return the arguments with only the unmasked points remaining.

Arguments can be in any of 5 categories:

- 1) 1-D masked arrays
- 2) 1-D ndarrays
- 3) ndarrays with more than one dimension
- 4) other non-string iterables
- 5) anything else

The first argument must be in one of the first four categories; any argument with a length differing from that of the first argument (and hence anything in category 5) then will be passed through unchanged.

Masks are obtained from all arguments of the correct length in categories 1, 2, and 4; a point is bad if masked in a masked array or if it is a nan or inf. No attempt is made to

extract a mask from categories 2, 3, and 4 if ``numpy.isfinite`` does not yield a Boolean array.

All input arguments that are not passed unchanged are returned as ndarrays after removing the points or rows corresponding to masks in any of the arguments.

A vastly simpler version of this function was originally written as a helper for `Axes.scatter()`.

matplotlib.cbook.file_requires_unicode

```
file_requires_unicode(x)
```

Return whether the given writable file-like object requires Unicode to be written to it.

matplotlib.cbook.flatten

```
flatten(seq, scalarp=)
```

Return a generator of flattened nested containers.

For example:

```
>>> from matplotlib.cbook import flatten
>>> l = (('John', ['Hunter']), (1, 23), [[[42, (5, 23)], ]])
>>> print(list(flatten(l)))
['John', 'Hunter', 1, 23, 42, 5, 23]
```

By: Composite of Holger Krekel and Luther Blissett

From: <https://code.activestate.com/recipes/121294-simple-generator-for-flattening-nested-containers/>
and Recipe 1.12 in cookbook

matplotlib.cbook.get_sample_data

```
get_sample_data(fname, asfileobj=True)
```

Return a sample data file. `*fname*` is a path relative to the `:file:`mpl-data/sample_data`` directory. If `*asfileobj*` is ``True`` return a file object, otherwise just a file path.

Sample data files are stored in the `'mpl-data/sample_data'` directory within the Matplotlib package.

If the filename ends in `.gz`, the file is implicitly unzipped. If the filename ends with `.npy` or `.npz`, and `*asfileobj*` is ``True``, the file is loaded with ``numpy.load``.

matplotlib.cbook.index_of

```
index_of(y)
```

A helper function to create reasonable x values for the given `*y*`.

This is used for plotting (x, y) if x values are not explicitly given.

First try ```y.index``` (assuming `*y*` is a ``pandas.Series``), if that fails, use ```range(len(y))```.

This will be extended in the future to deal with more types of labeled data.

Parameters

y : float or array-like

Returns

x, y : ndarray

The x and y values to plot.

matplotlib.cbook.is_math_text

```
is_math_text(s)
```

Return whether the string `*s*` contains math expressions.

This is done by checking whether `*s*` contains an even number of non-escaped dollar signs.

matplotlib.cbook.is_scalar_or_string

```
is_scalar_or_string(val)
```

Return whether the given object is a scalar or string like.

matplotlib.cbook.is_writable_file_like

```
is_writable_file_like(obj)
```

Return whether `*obj*` looks like a file object with a `*write*` method.

matplotlib.cbook.normalize_kwargs

```
normalize_kwargs(kw, alias_mapping=None)
```

Helper function to normalize kwarg inputs.

Parameters

kw : dict or None

A dict of keyword arguments. None is explicitly supported and treated as an empty dict, to support functions with an optional parameter of the form ```props=None```.

alias_mapping : dict or Artist subclass or Artist instance, optional

A mapping between a canonical name to a list of aliases, in order of

precedence from lowest to highest.

If the canonical value is not in the list it is assumed to have the highest priority.

If an Artist subclass or instance is passed, use its properties alias mapping.

Raises

TypeError

To match what Python raises if invalid arguments/keyword arguments are passed to a callable.

matplotlib.cbook.open_file_cm

```
open_file_cm(path_or_file, mode='r', encoding=None)
```

Pass through file objects and context-manage path-likes.

matplotlib.cbook.print_cycles

```
print_cycles(objects, outstream=<_io.TextIOWrapper name='' mode='w' encoding='utf-8'>,
show_progress=False)
```

Print loops of cyclic references in the given *objects*.

It is often useful to pass in ``gc.garbage`` to find the cycles that are preventing some objects from being garbage collected.

Parameters

objects

A list of objects to find cycles in.

outstream

The stream for output.

show_progress : bool

If True, print the number of objects reached as they are found.

matplotlib.cbook.pts_to_midstep

```
pts_to_midstep(x, *args)
```

Convert continuous line to mid-steps.

Given a set of ``N`` points convert to ``2N`` points which when connected linearly give a step function which changes values at the middle of the intervals.

Parameters

x : array

The x location of the steps. May be empty.

y1, ..., yp : array

y arrays to be turned into steps; all must be the same length as ``x``.

Returns

array

The x and y values converted to steps in the same order as the input; can be unpacked as ``x_out, y1_out, ..., yp_out``. If the input is length ``N``, each of these arrays will be length ``2N``.

Examples

```
>>> x_s, y1_s, y2_s = pts_to_midstep(x, y1, y2)
```

matplotlib.cbook.pts_to_poststep

```
pts_to_poststep(x, *args)
```

Convert continuous line to post-steps.

Given a set of ``N`` points convert to ``2N + 1`` points, which when connected linearly give a step function which changes values at the end of the intervals.

Parameters

x : array

The x location of the steps. May be empty.

y1, ..., yp : array

y arrays to be turned into steps; all must be the same length as ``x``.

Returns

array

The x and y values converted to steps in the same order as the input; can be unpacked as ``x_out, y1_out, ..., yp_out``. If the input is length ``N``, each of these arrays will be length ``2N + 1``. For ``N=0``, the length will be 0.

Examples

```
>>> x_s, y1_s, y2_s = pts_to_poststep(x, y1, y2)
```

matplotlib.cbook.pts_to_prestep

```
pts_to_prestep(x, *args)
```

Convert continuous line to pre-steps.

Given a set of ``N`` points, convert to ``2N - 1`` points, which when connected linearly give a step function which changes values at the beginning of the intervals.

Parameters

`x` : array

The `x` location of the steps. May be empty.

`y1, ..., yp` : array

`y` arrays to be turned into steps; all must be the same length as `x`.

Returns

array

The `x` and `y` values converted to steps in the same order as the input; can be unpacked as `x_out, y1_out, ..., yp_out`. If the input is length `N`, each of these arrays will be length `2N + 1`. For `N=0`, the length will be 0.

Examples

```
>>> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)
```

matplotlib.cbook.safe_first_element

```
safe_first_element(obj)
```

Return the first element in `*obj`.

This is a type-independent way of obtaining the first element, supporting both index access and the iterator protocol.

matplotlib.cbook.safe_masked_invalid

```
safe_masked_invalid(x, copy=False)
```

No description available.

matplotlib.cbook.sanitize_sequence

```
sanitize_sequence(data)
```

Convert dictview objects to list. Other inputs are returned unchanged.

matplotlib.cbook.silent_list

```
silent_list(type, seq=None)
```

A list with a short `repr()`.

This is meant to be used for a homogeneous list of artists, so that they don't cause long, meaningless output.

Instead of ::

```
[<matplotlib.lines.Line2D object at 0x7f5749fed3c8>,  
<matplotlib.lines.Line2D object at 0x7f5749fed4e0>,
```

<matplotlib.lines.Line2D object at 0x7f5758016550>]

one will get ::

<a list of 3 Line2D objects>

If ``self.type`` is None, the type name is obtained from the first item in the list (if any).

matplotlib.cbook.simple_linear_interpolation

```
simple_linear_interpolation(a, steps)
```

Resample an array with ``steps - 1`` points between original point pairs.

Along each column of *a*, ``(steps - 1)`` points are introduced between each original values; the values are linearly interpolated.

Parameters

a : array, shape (n, ...)

steps : int

Returns

array

shape ``((n - 1) * steps + 1, ...)``

matplotlib.cbook.strip_math

```
strip_math(s)
```

Remove latex formatting from mathtext.

Only handles fully math and fully non-math strings.

matplotlib.cbook.to_filehandle

```
to_filehandle(fname, flag='r', return_opened=False, encoding=None)
```

Convert a path to an open file handle or pass-through a file-like object.

Consider using ``open_file_cm`` instead, as it allows one to properly close newly created file objects more easily.

Parameters

fname : str or path-like or file-like

If ``str`` or ``os.PathLike``, the file is opened using the flags specified by *flag* and *encoding*. If a file-like object, it is passed through.

flag : str, default: 'r'

Passed as the *mode* argument to ``open`` when *fname* is ``str`` or ``os.PathLike``; ignored if *fname* is file-like.

return_opened : bool, default: False

If True, return both the file object and a boolean indicating whether this was a new file (that the caller needs to close). If False, return only the new file.

encoding : str or None, default: None

Passed as the *mode* argument to ``open`` when *fname* is ``str`` or ``os.PathLike``; ignored if *fname* is file-like.

Returns

fh : file-like

opened : bool

opened is only returned if *return_opened* is True.

matplotlib.cbook.violin_stats

```
violin_stats(X, method, points=100, quantiles=None)
```

Return a list of dictionaries of data which can be used to draw a series of violin plots.

See the ``Returns`` section below to view the required keys of the dictionary.

Users can skip this function and pass a user-defined set of dictionaries with the same keys to ``~.axes.Axes.violinplot`` instead of using Matplotlib to do the calculations. See the *Returns* section below for the keys that must be present in the dictionaries.

Parameters

X : array-like

Sample data that will be used to produce the gaussian kernel density estimates. Must have 2 or fewer dimensions.

method : callable

The method used to calculate the kernel density estimate for each column of data. When called via ``method(v, coords)``, it should return a vector of the values of the KDE evaluated at the values specified in coords.

points : int, default: 100

Defines the number of points to evaluate each of the gaussian kernel density estimates at.

quantiles : array-like, default: None

Defines (if not None) a list of floats in interval [0, 1] for each column of data, which represents the quantiles that will be rendered for that column of data. Must have 2 or fewer dimensions. 1D array will be treated as a singleton list containing them.

Returns

list of dict

A list of dictionaries containing the results for each column of data.

The dictionaries contain at least the following:

- coords: A list of scalars containing the coordinates this particular kernel density estimate was evaluated at.
- vals: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in *coords*.
- mean: The mean value for this column of data.
- median: The median value for this column of data.
- min: The minimum value for this column of data.
- max: The maximum value for this column of data.
- quantiles: The quantile values for this column of data.

matplotlib.cm

```
cm(...)
```

Builtin colormaps, colormap handling utilities, and the `ScalarMappable` mixin.

.. seealso::

:doc:`gallery/color/colormap_reference` for a list of builtin colormaps.

:ref:`colormap-manipulation` for examples of how to make colormaps.

:ref:`colormaps` an in-depth discussion of choosing colormaps.

:ref:`colormapnorms` for more details about data normalization.

matplotlib.cm.ColormapRegistry

```
ColormapRegistry(cmaps)
```

Container for colormaps that are known to Matplotlib by name.

The universal registry instance is `matplotlib.colormaps`. There should be no need for users to instantiate `ColormapRegistry` themselves.

Read access uses a dict-like interface mapping names to `Colormap`'s::

```
import matplotlib as mpl
cmap = mpl.colormaps['viridis']
```

Returned `Colormap`'s are copies, so that their modification does not change the global definition of the colormap.

Additional colormaps can be added via `ColormapRegistry.register`::

```
mpl.colormaps.register(my_colormap)
```

To get a list of all registered colormaps, you can do::

```
from matplotlib import colormaps
```

list(colormaps)

matplotlib.cm.ScalarMappable

```
ScalarMappable(norm=None, cmap=None, *, colorizer=None, **kwargs)
```

A mixin class to map one or multiple sets of scalar data to RGBA.

The ScalarMappable applies data normalization before returning RGBA colors from the given `~matplotlib.colors.Colormap`.

matplotlib.cm.get_cmap

```
get_cmap(name=None, lut=None)
```

[*Deprecated*] Get a colormap instance, defaulting to rc values if **name** is None.

Parameters

name : `~matplotlib.colors.Colormap` or str or None, default: None
If a `~matplotlib.colors.Colormap` instance, it will be returned. Otherwise, the name of a colormap known to Matplotlib, which will be resampled by **lut**. The default, None, means `:rc:`image.cmap``.

lut : int or None, default: None

If **name** is not already a Colormap instance and **lut** is not None, the colormap will be resampled to have **lut** entries in the lookup table.

Returns

Colormap

Notes

.. deprecated:: 3.7

Use `matplotlib.colormaps[name]` or `matplotlib.colormaps.get_cmap()` or `pyplot.get_cmap()` instead.

matplotlib.collections

```
collections(...)
```

Classes for the efficient drawing of large collections of objects that share most properties, e.g., a large number of line segments or polygons.

The classes are not meant to be as flexible as their single element counterparts (e.g., you may not be able to select all line styles) but they are meant to be fast for common use cases (e.g., a large set of solid line segments).

matplotlib.collections.AsteriskPolygonCollection

```
AsteriskPolygonCollection(numsides, *, rotation=0, sizes=(1,), **kwargs)
```

Draw a collection of regular asterisks with **numsides** points.

matplotlib.collections.CapStyle

```
CapStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the two endpoints (caps) of an unclosed line are drawn.

How to draw the start and end points of lines that represent a closed curve (i.e. that end in a `~.path.Path.CLOSEPOLY`) is controlled by the line's `JoinStyle`. For all other lines, how the start and end points are drawn is controlled by the `*CapStyle*`.

For a visual impression of each `*CapStyle*`, [view these docs online <CapStyle>](#) or run `CapStyle.demo`.

By default, `~.backend_bases.GraphicsContextBase` draws a stroked line as squared off at its endpoints.

Supported values:

.. rst-class:: value-list

'butt'

the line is squared off at its endpoint.

'projecting'

the line is squared off as in `*butt*`, but the filled in area extends beyond the endpoint a distance of `linewidth/2`.

'round'

like `*butt*`, but a semicircular cap is added to the end of the line, of radius `linewidth/2`.

.. plot::

:alt: Demo of possible CapStyle's

```
from matplotlib._enums import CapStyle
CapStyle.demo()
```

matplotlib.collections.CircleCollection

```
CircleCollection(sizes, **kwargs)
```

A collection of circles, drawn using splines.

matplotlib.collections.Collection

```
Collection(*, edgecolors=None, facecolors=None, linewidths=None, linestyle='solid',
capstyle=None, joinstyle=None, antialiaseds=None, offsets=None, offset_transform=None,
norm=None, cmap=None, colorizer=None, pickradius=5.0, hatch=None, urls=None, zorder=1,
**kwargs)
```

Base class for Collections. Must be subclassed to be usable.

A Collection represents a sequence of `~.Patch`'es that can be drawn more efficiently together than individually. For example, when a single path is being drawn repeatedly at different offsets, the renderer can

typically execute a `draw_marker()` call much more efficiently than a series of repeated calls to `draw_path()` with the offsets put in one-by-one.

Most properties of a collection can be configured per-element. Therefore, Collections have "plural" versions of many of the properties of a `Patch` (e.g. `Collection.get_paths` instead of `Patch.get_path`). Exceptions are the `*zorder*`, `*hatch*`, `*pickradius*`, `*capstyle*` and `*joinstyle*` properties, which can only be set globally for the whole collection.

Besides these exceptions, all properties can be specified as single values (applying to all elements) or sequences of values. The property of the `i`th element of the collection is::

```
prop[i % len(prop)]
```

Each Collection can optionally be used as its own `ScalarMappable` by passing the `*norm*` and `*cmap*` parameters to its constructor. If the Collection's `ScalarMappable` matrix `__A__` has been set (via a call to `Collection.set_array`), then at draw time this internal scalar mappable will be used to set the `facecolors` and `edgecolors`, ignoring those that were manually passed in.

matplotlib.collections.EllipseCollection

```
EllipseCollection(widths, heights, angles, *, units='points', **kwargs)
```

A collection of ellipses, drawn using splines.

matplotlib.collections.EventCollection

```
EventCollection(positions, orientation='horizontal', *, lineoffset=0, linelength=1, linewidth=None, color=None, linestyle='solid', antialiased=None, **kwargs)
```

A collection of locations along a single axis at which an "event" occurred.

The events are given by a 1-dimensional array. They do not have an amplitude and are displayed as parallel lines.

matplotlib.collections.FillBetweenPolyCollection

```
FillBetweenPolyCollection(t_direction, t, f1, f2, *, where=None, interpolate=False, step=None, **kwargs)
```

`PolyCollection` that fills the area between two x- or y-curves.

matplotlib.collections.JoinStyle

```
JoinStyle(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
```

Define how the connection between two line segments is drawn.

For a visual impression of each `JoinStyle`, view these docs online [<JoinStyle>](#), or run `JoinStyle.demo`.

Lines in Matplotlib are typically defined by a 1D `~.path.Path` and a finite `linewidth`, where the underlying 1D `~.path.Path` represents the center of the stroked line.

By default, `~.backend_bases.GraphicsContextBase` defines the boundaries of a stroked line to simply be every point within some radius, `linewidth/2`, away from any point of the center line. However, this results in corners appearing "rounded", which may not be the desired behavior if you are drawing, for example, a polygon or pointed star.

Supported values:

.. rst-class:: value-list

'miter'

the "arrow-tip" style. Each boundary of the filled-in area will extend in a straight line parallel to the tangent vector of the centerline at the point it meets the corner, until they meet in a sharp point.

'round'

stokes every point within a radius of `linewidth/2` of the center lines.

'bevel'

the "squared-off" style. It can be thought of as a rounded corner where the "circular" part of the corner has been cut off.

.. note::

Very long miter tips are cut off (to form a *bevel*) after a backend-dependent limit called the "miter limit", which specifies the maximum allowed ratio of miter length to line width. For example, the PDF backend uses the default value of 10 specified by the PDF standard, while the SVG backend does not even specify the miter limit, resulting in a default value of 4 per the SVG specification. Matplotlib does not currently allow the user to adjust this parameter.

A more detailed description of the effect of a miter limit can be found in the `~Mozilla Developer Docs`

<<https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-miterlimit>>`_

.. plot::

:alt: Demo of possible JoinStyle's

```
from matplotlib._enums import JoinStyle
JoinStyle.demo()
```

matplotlib.collections.LineCollection

```
LineCollection(segments, *, zorder=2, **kwargs)
```

Represents a sequence of `~.Line2D`'s that should be drawn together.

This class extends `~.Collection` to represent a sequence of

`.Line2D``'s instead of just a sequence of `.Patch``'s.
 Just as in `.Collection``, each property of a `*LineCollection*` may be either a single value or a list of values. This list is then used cyclically for each element of the `LineCollection`, so the property of the `i``th element of the collection is::

```
prop[i % len(prop)]
```

The properties of each member of a `*LineCollection*` default to their values in `:rc:`lines.*`` instead of `:rc:`patch.*``, and the property `*colors*` is added in place of `*edgecolors*`.

matplotlib.collections.PatchCollection

```
PatchCollection(patches, *, match_original=False, **kwargs)
```

A generic collection of patches.

`PatchCollection` draws faster than a large number of equivalent individual `Patches`. It also makes it easier to assign a colormap to a heterogeneous collection of patches.

matplotlib.collections.PathCollection

```
PathCollection(paths, sizes=None, **kwargs)
```

A collection of `~.path.Path``'s, as created by e.g. `~.Axes.scatter``.

matplotlib.collections.PolyCollection

```
PolyCollection(verts, sizes=None, *, closed=True, **kwargs)
```

Base class for collections that have an array of sizes.

matplotlib.collections.PolyQuadMesh

```
PolyQuadMesh(coordinates, **kwargs)
```

Class for drawing a quadrilateral mesh as individual `Polygons`.

A quadrilateral mesh is a grid of `M` by `N` adjacent quadrilaterals that are defined via a `(M+1, N+1)` grid of vertices. The quadrilateral `(m, n)` is defined by the vertices ::

```
(m+1, n) ----- (m+1, n+1)
//
//
//
(m, n) ----- (m, n+1)
```

The mesh need not be regular and the polygons need not be convex.

Parameters

coordinates : (M+1, N+1, 2) array-like
The vertices. ``coordinates[m, n]`` specifies the (x, y) coordinates of vertex (m, n).

Notes

Unlike ``QuadMesh``, this class will draw each cell as an individual Polygon. This is significantly slower, but allows for more flexibility when wanting to add additional properties to the cells, such as hatching.

Another difference from ``QuadMesh`` is that if any of the vertices or data of a cell are masked, that Polygon will **not** be drawn and it won't be in the list of paths returned.

matplotlib.collections.QuadMesh

```
QuadMesh(coordinates, *, antialiased=True, shading='flat', **kwargs)
```

Class for the efficient drawing of a quadrilateral mesh.

A quadrilateral mesh is a grid of M by N adjacent quadrilaterals that are defined via a (M+1, N+1) grid of vertices. The quadrilateral (m, n) is defined by the vertices ::

```
(m+1, n) ----- (m+1, n+1)
//
//
//
(m, n) ----- (m, n+1)
```

The mesh need not be regular and the polygons need not be convex.

Parameters

coordinates : (M+1, N+1, 2) array-like
The vertices. ``coordinates[m, n]`` specifies the (x, y) coordinates of vertex (m, n).

antialiased : bool, default: True

shading : {'flat', 'gouraud'}, default: 'flat'

Notes

Unlike other ``Collection``s, the default *pickradius* of ``QuadMesh`` is 0, i.e. ``~.Artist.contains`` checks whether the test point is within any of the mesh quadrilaterals.

matplotlib.collections.RegularPolyCollection

```
RegularPolyCollection(numsides, *, rotation=0, sizes=(1,), **kwargs)
```

A collection of n-sided regular polygons.

matplotlib.collections.StarPolygonCollection

```
StarPolygonCollection(numsides, *, rotation=0, sizes=(1,), **kwargs)
```

Draw a collection of regular stars with *numsides* points.

matplotlib.collections.TriMesh

```
TriMesh(triangulation, **kwargs)
```

Class for the efficient drawing of a triangular mesh using Gouraud shading.

A triangular mesh is a `~matplotlib.tri.Triangulation`` object.

matplotlib.colorbar

```
colorbar(...)
```

Colorbars are a visualization of the mapping from scalar values to colors. In Matplotlib they are drawn into a dedicated `~.axes.Axes``.

.. note::

Colorbars are typically created through `.Figure.colorbar`` or its pyplot wrapper `.pyplot.colorbar``, which internally use `.Colorbar`` together with `.make_axes_gridspec`` (for `.GridSpec``-positioned Axes) or `.make_axes`` (for non-`.GridSpec``-positioned Axes).

End-users most likely won't need to directly use this module's API.

matplotlib.colorbar.Colorbar

```
Colorbar(ax, mappable=None, *, alpha=None, location=None, extend=None, extendfrac=None, extendrect=False, ticks=None, format=None, values=None, boundaries=None, spacing='uniform', drawedges=False, label='', cmap=None, norm=None, orientation=None, ticklocation='auto')
```

Draw a colorbar in an existing Axes.

Typically, colorbars are created using `.Figure.colorbar`` or `.pyplot.colorbar`` and associated with `.ScalarMappable`s` (such as an `.AxesImage`` generated via `~.axes.Axes.imshow``).

In order to draw a colorbar not associated with other elements in the figure, e.g. when showing a colormap by itself, one can create an empty `.ScalarMappable``, or directly pass *cmap* and *norm* instead of *mappable* to `.Colorbar``.

Useful public methods are `:meth:`set_label`` and `:meth:`add_lines``.

Attributes

ax : `~matplotlib.axes.Axes``

The `~.axes.Axes`` instance in which the colorbar is drawn.

lines : list

A list of `.LineCollection`` (empty if no lines were drawn).

dividers : `.LineCollection``

A `.LineCollection`` (empty if *drawedges* is `False``).

matplotlib.colorbar.ColorbarBase

```
ColorbarBase(ax, mappable=None, *, alpha=None, location=None, extend=None,
             extendfrac=None, extendrect=False, ticks=None, format=None, values=None,
             boundaries=None, spacing='uniform', drawedges=False, label='', cmap=None, norm=None,
             orientation=None, ticklocation='auto')
```

Draw a colorbar in an existing Axes.

Typically, colorbars are created using `Figure.colorbar`` or `pyplot.colorbar`` and associated with `ScalarMappable`s` (such as an `AxesImage`` generated via `~.axes.Axes.imshow``).

In order to draw a colorbar not associated with other elements in the figure, e.g. when showing a colormap by itself, one can create an empty `ScalarMappable``, or directly pass `*cmap*` and `*norm*` instead of `*mappable*` to `Colorbar``.

Useful public methods are `:meth:`set_label`` and `:meth:`add_lines``.

Attributes

`ax` : `~matplotlib.axes.Axes``
The `~.axes.Axes`` instance in which the colorbar is drawn.
`lines` : list
A list of `.LineCollection`` (empty if no lines were drawn).
`dividers` : `.LineCollection``
A `LineCollection`` (empty if `*drawedges*` is `False``).

matplotlib.colorbar.make_axes

```
make_axes(parents, location=None, orientation=None, fraction=0.15, shrink=1.0,
          aspect=20, **kwargs)
```

Create an `~.axes.Axes`` suitable for a colorbar.

The Axes is placed in the figure of the `*parents*` Axes, by resizing and repositioning `*parents*`.

Parameters

`parents` : `~matplotlib.axes.Axes`` or iterable or `numpy.ndarray`` of `~.axes.Axes``
The Axes to use as parents for placing the colorbar.

`location` : None or {'left', 'right', 'top', 'bottom'}
The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the `*orientation*` of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the `*orientation*` if it is set (vertical colorbars on the right, horizontal ones at the bottom), or default to 'right' if `*orientation*` is unset.

`orientation` : None or {'vertical', 'horizontal'}
The orientation of the colorbar. It is preferable to set the `*location*` of the colorbar, as that also determines the `*orientation*`; passing

incompatible values for **location** and **orientation** raises an exception.

fraction : float, default: 0.15

Fraction of original Axes to use for colorbar.

shrink : float, default: 1.0

Fraction by which to multiply the size of the colorbar.

aspect : float, default: 20

Ratio of long to short dimensions.

pad : float, default: 0.05 if vertical, 0.15 if horizontal

Fraction of original Axes between colorbar and new image Axes.

anchor : (float, float), optional

The anchor point of the colorbar Axes.

Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

panchor : (float, float), or **False**, optional

The anchor point of the colorbar parent Axes. If **False**, the parent axes' anchor will be unchanged.

Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

Returns

cax : `~matplotlib.axes.Axes``

The child Axes.

kwargs : dict

The reduced keyword dictionary to be passed when creating the colorbar instance.

matplotlib.colorbar.make_axes_gridspec

```
make_axes_gridspec(parent, *, location=None, orientation=None, fraction=0.15,
                    shrink=1.0, aspect=20, **kwargs)
```

Create an `~.axes.Axes`` suitable for a colorbar.

The Axes is placed in the figure of the **parent** Axes, by resizing and repositioning **parent**.

This function is similar to `~.make_axes`` and mostly compatible with it. Primary differences are

- `~.make_axes_gridspec`` requires the **parent** to have a `subplotspec``.
- `~.make_axes`` positions the Axes in figure coordinates;
- `~.make_axes_gridspec`` positions it using a `subplotspec``.
- `~.make_axes`` updates the position of the parent. `~.make_axes_gridspec`` replaces the parent `gridspec`` with a new one.

Parameters

parent : `~matplotlib.axes.Axes``

The Axes to use as parent for placing the colorbar.

location : None or {'left', 'right', 'top', 'bottom'}

The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the **orientation** of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the **orientation** if it is set (vertical colorbars on the right, horizontal ones at the bottom), or default to 'right' if **orientation** is unset.

orientation : None or {'vertical', 'horizontal'}

The orientation of the colorbar. It is preferable to set the **location** of the colorbar, as that also determines the **orientation**; passing incompatible values for **location** and **orientation** raises an exception.

fraction : float, default: 0.15

Fraction of original Axes to use for colorbar.

shrink : float, default: 1.0

Fraction by which to multiply the size of the colorbar.

aspect : float, default: 20

Ratio of long to short dimensions.

pad : float, default: 0.05 if vertical, 0.15 if horizontal

Fraction of original Axes between colorbar and new image Axes.

anchor : (float, float), optional

The anchor point of the colorbar Axes.

Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

panchor : (float, float), or **False**, optional

The anchor point of the colorbar parent Axes. If **False**, the parent axes' anchor will be unchanged.

Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

Returns

cax : `~matplotlib.axes.Axes`

The child Axes.

kwargs : dict

The reduced keyword dictionary to be passed when creating the colorbar instance.

matplotlib.colorbar

```
colorizer(...)
```

The Colorizer class which handles the data to color pipeline via a normalization and a colormap.

.. admonition:: Provisional status of colorizer

The ```colorizer``` module and classes in this file are considered provisional and may change at any time without a deprecation period.

.. seealso::

:doc:`gallery/color/colormap_reference` for a list of builtin colormaps.

:ref:`colormap-manipulation` for examples of how to make colormaps.

:ref:`colormaps` for an in-depth discussion of choosing colormaps.

:ref:`colormapnorms` for more details about data normalization.

matplotlib.colorbar.Colorizer

```
Colorizer(cmap=None, norm=None)
```

Data to color pipeline.

This pipeline is accessible via `.Colorizer.to_rgba` and executed via the .Colorizer.norm` and .Colorizer.cmap` attributes.`

Parameters

`cmap`: `colorbar.Colorbar` or str or None, default: None
The colormap used to color data.

`norm`: `colors.Normalize` or str or None, default: None
The normalization used to normalize the data

matplotlib.colorbar.ColorizingArtist

```
ColorizingArtist(colorizer, **kwargs)
```

Base class for artists that make map data to color using a `.colorizer.Colorizer`.`

The `.colorizer.Colorizer` applies data normalization before returning RGBA colors from a ~matplotlib.colors.Colormap`.`

matplotlib.colors

```
colors(...)
```

A module for converting numbers or color arguments to `*RGB*` or `*RGBA*`.

`*RGB*` and `*RGBA*` are sequences of, respectively, 3 or 4 floats in the range 0-1.

This module includes functions and classes for color specification conversions, and for mapping numbers to colors in a 1-D array of colors called a colormap.

Mapping data onto colors using a colormap typically involves two steps: a data array is first mapped onto the range 0-1 using a subclass of ``Normalize``, then this number is mapped to a color using a subclass of ``Colormap``. Two subclasses of ``Colormap`` provided here: ``LinearSegmentedColormap``, which uses piecewise-linear interpolation to define colormaps, and ``ListedColormap``, which makes a colormap from a list of colors.

.. seealso::

:ref:`colormap-manipulation` for examples of how to make colormaps and

:ref:`colormaps` for a list of built-in colormaps.

:ref:`colormapnorms` for more details about data normalization

More colormaps are available at [palettable_](#).

The module also provides functions for checking whether an object can be interpreted as a color (``is_color_like``), for converting such an object to an RGBA tuple (``to_rgba``) or to an HTML-like hex string in the `"#rrggbb"` format (``to_hex``), and a sequence of colors to an `(n, 4)` RGBA array (``to_rgba_array``). Caching is used for efficiency.

Colors that Matplotlib recognizes are listed at :ref:`colors_def`.

.. _palettable: <https://jiffyclub.github.io/palettable/>

.. _xkcd color survey: <https://xkcd.com/color/rgb/>

matplotlib.colors.AsinhNorm

```
AsinhNorm(linear_width=1, vmin=None, vmax=None, clip=False)
```

The inverse hyperbolic sine scale is approximately linear near the origin, but becomes logarithmic for larger positive or negative values. Unlike the ``SymLogNorm``, the transition between these linear and logarithmic regions is smooth, which may reduce the risk of visual artifacts.

.. note::

This API is provisional and may be revised in the future based on early user feedback.

Parameters

`linear_width` : float, default: 1

The effective width of the linear region, beyond which the transformation becomes asymptotically logarithmic

matplotlib.colors.BivarColormap

```
BivarColormap(N=256, M=256, shape='square', origin=(0, 0), name='bivariate colormap')
```

Base class for all bivariate to RGBA mappings.

Designed as a drop-in replacement for `Colormap` when using a 2D lookup table. To be used with ``~matplotlib.cm.ScalarMappable``.

matplotlib.colors.BivarColormapFromImage

```
BivarColormapFromImage(lut, shape='square', origin=(0, 0), name='from image')
```

BivarColormap object generated by supersampling a regular grid.

Parameters

lut : nparray of shape (N, M, 3) or (N, M, 4)

The look-up-table

shape: {'square', 'circle', 'ignore', 'circleignore'}

- If 'square' each variate is clipped to [0,1] independently
- If 'circle' the variates are clipped radially to the center of the colormap, and a circular mask is applied when the colormap is displayed
- If 'ignore' the variates are not clipped, but instead assigned the 'outside' color
- If 'circleignore' a circular mask is applied, but the data is not clipped

origin: (float, float)

The relative origin of the colormap. Typically (0, 0), for colormaps that are linear on both axis, and (.5, .5) for circular colormaps.

Used when getting 1D colormaps from 2D colormaps.

name : str, optional

The name of the colormap.

matplotlib.colors.BoundaryNorm

```
BoundaryNorm(boundaries, ncolors, clip=False, *, extend='neither')
```

Generate a colormap index based on discrete intervals.

Unlike ``Normalize`` or ``LogNorm``, ``BoundaryNorm`` maps values to integers instead of to the interval 0-1.

matplotlib.colors.CenteredNorm

```
CenteredNorm(vcenter=0, halfrange=None, clip=False)
```

A class which, when called, maps values within the interval ```[vmin, vmax]``` linearly to the interval ```[0.0, 1.0]```. The mapping of values outside ```[vmin, vmax]``` depends on `*clip*`.

Examples

::

```
x = [-2, -1, 0, 1, 2]
```

```
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=False)
```

```
norm(x) # [-0.5, 0., 0.5, 1., 1.5]
```

```
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=True)
```

```
norm(x) # [0., 0., 0.5, 1., 1.]
```

See Also

:ref:`colormapnorms`

matplotlib.colors.ColorConverter

```
ColorConverter()
```

A class only kept for backwards compatibility.

Its functionality is entirely provided by module-level functions.

matplotlib.colors.ColorSequenceRegistry

```
ColorSequenceRegistry()
```

Container for sequences of colors that are known to Matplotlib by name.

The universal registry instance is `matplotlib.color_sequences`. There should be no need for users to instantiate `.ColorSequenceRegistry` themselves.

Read access uses a dict-like interface mapping names to lists of colors::

```
import matplotlib as mpl
colors = mpl.color_sequences['tab10']
```

For a list of built in color sequences, see :doc:`/gallery/color/color_sequences`. The returned lists are copies, so that their modification does not change the global definition of the color sequence.

Additional color sequences can be added via `.ColorSequenceRegistry.register`::`

```
mpl.color_sequences.register('rgb', ['r', 'g', 'b'])
```

matplotlib.colors.Colormap

```
Colormap(name, N=256)
```

Baseclass for all scalar to RGBA mappings.

Typically, Colormap instances are used to convert data values (floats) from the interval `[0, 1]` to the RGBA color that the respective Colormap represents. For scaling of data into the `[0, 1]` interval see `matplotlib.colors.Normalize`. Subclasses of `matplotlib.cm.ScalarMappable` make heavy use of this `data -> normalize -> map-to-color` processing chain.

matplotlib.colors.FuncNorm

```
FuncNorm(functions, vmin=None, vmax=None, clip=False)
```

Arbitrary normalization using functions for the forward and inverse.

Parameters

functions : (callable, callable)

two-tuple of the forward and inverse functions for the normalization.

The forward function must be monotonic.

Both functions must have the signature ::

```
def forward(values: array-like) -> array-like
```

vmin, vmax : float or None

If **vmin** and/or **vmax** is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`.

clip : bool, default: False

Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed by the function, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these **under** and **over** values with specific colors.

If clipping is on, values below **vmin** are mapped to 0 and values above **vmax** are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

matplotlib.colors.LightSource

```
LightSource(azdeg=315, altdeg=45, hsv_min_val=0, hsv_max_val=1, hsv_min_sat=1,
hsv_max_sat=0)
```

Create a light source coming from the specified azimuth and elevation. Angles are in degrees, with the azimuth measured clockwise from north and elevation up from the zero plane of the surface.

``shade`` is used to produce "shaded" RGB values for a data array.

``shade_rgb`` can be used to combine an RGB image with an elevation map.

``hillshade`` produces an illumination map of a surface.

matplotlib.colors.LinearSegmentedColormap

```
LinearSegmentedColormap(name, segmentdata, N=256, gamma=1.0)
```

Colormap objects based on lookup tables using linear segments.

The lookup table is generated using linear interpolation for each primary color, with the 0-1 domain divided into any number of segments.

matplotlib.colors.ListedColormap

```
ListedColormap(colors, name='from_list', N=None)
```

Colormap object generated from a list of colors.

This may be most useful when indexing directly into a colormap, but it can also be used to generate special colormaps for ordinary mapping.

Parameters

colors : list, array

Sequence of Matplotlib color specifications (color names or RGB(A) values).

name : str, optional

String to identify the colormap.

N : int, optional

Number of entries in the map. The default is **None**, in which case there is one colormap entry for each element in the list of colors.

If ::

$N < \text{len}(\text{colors})$

the list will be truncated at **N**. If ::

$N > \text{len}(\text{colors})$

the list will be extended by repetition.

matplotlib.colors.LogNorm

```
LogNorm(vmin=None, vmax=None, clip=False)
```

Normalize a given value to the 0-1 range on a log scale.

matplotlib.colors.MultivarColormap

```
MultivarColormap(colormaps, combination_mode, name='multivariate colormap')
```

Class for holding multiple `~matplotlib.colors.Colormap`` for use in a `~matplotlib.cm.ScalarMappable`` object

matplotlib.colors.NoNorm

```
NoNorm(vmin=None, vmax=None, clip=False)
```

Dummy replacement for ``Normalize``, for the case where we want to use indices directly in a `~matplotlib.cm.ScalarMappable``.

matplotlib.colors.Normalize

```
Normalize(vmin=None, vmax=None, clip=False)
```

A class which, when called, maps values within the interval ```[vmin, vmax]``` linearly to the interval ```[0.0, 1.0]```. The mapping of values outside ```[vmin, vmax]``` depends on **clip**.

Examples

::

```
x = [-2, -1, 0, 1, 2]
```

```
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=False)
```

```
norm(x) # [-0.5, 0., 0.5, 1., 1.5]
```

```
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=True)
```

```
norm(x) # [0., 0., 0.5, 1., 1.]
```

See Also

:ref:`colormapnorms`

matplotlib.colors.PowerNorm

```
PowerNorm(gamma, vmin=None, vmax=None, clip=False)
```

Linearly map a given value to the 0-1 range and then apply a power-law normalization over that range.

Parameters

gamma : float

Power law exponent.

vmin, vmax : float or None

If *vmin* and/or *vmax* is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., ``__call__(A)`` calls ``autoscale_None(A)``.

clip : bool, default: False

Determines the behavior for mapping values outside the range ``[vmin, vmax]``.

If clipping is off, values above *vmax* are transformed by the power function, resulting in values above 1, and values below *vmin* are linearly transformed resulting in values below 0. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

Notes

The normalization formula is

.. math::

$$\left(\frac{x - v_{\min}}{v_{\max} - v_{\min}} \right)^{\gamma}$$

For input values below *vmin*, gamma is set to one.

matplotlib.colors.SegmentedBivarColormap

```
SegmentedBivarColormap(patch, N=256, shape='square', origin=(0, 0), name='segmented
bivariate colormap')
```

BivarColormap object generated by supersampling a regular grid.

Parameters

patch : np.array

Patch is required to have a shape (k, l, 3), and will get supersampled to a lut of shape (N, N, 4).

N : int

The number of RGB quantization levels along each axis.

shape : {'square', 'circle', 'ignore', 'circleignore'}

- If 'square' each variate is clipped to [0,1] independently
- If 'circle' the variates are clipped radially to the center of the colormap, and a circular mask is applied when the colormap is displayed
- If 'ignore' the variates are not clipped, but instead assigned the 'outside' color
- If 'circleignore' a circular mask is applied, but the data is not clipped

origin : (float, float)

The relative origin of the colormap. Typically (0, 0), for colormaps that are linear on both axis, and (.5, .5) for circular colormaps.

Used when getting 1D colormaps from 2D colormaps.

name : str, optional

The name of the colormap.

matplotlib.colors.SymLogNorm

```
SymLogNorm(linthresh, linscale=1.0, vmin=None, vmax=None, clip=False, *, base=10)
```

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter `*linthresh*` allows the user to specify the size of this range (`-*linthresh*`, `*linthresh*`).

Parameters

linthresh : float

The range within which the plot is linear (to avoid having the plot go to infinity around zero).

linscale : float, default: 1

This allows the linear range (`-*linthresh*` to `*linthresh*`) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when `*linscale* == 1.0` (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

base : float, default: 10

matplotlib.colors.TwoSlopeNorm

```
TwoSlopeNorm(vcenter, vmin=None, vmax=None)
```

A class which, when called, maps values within the interval `[vmin, vmax]` linearly to the interval `[0.0, 1.0]`. The mapping of values outside `[vmin, vmax]` depends on `*clip*`.

Examples

```
-----  
::
```

```
x = [-2, -1, 0, 1, 2]
```

```
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=False)  
norm(x) # [-0.5, 0., 0.5, 1., 1.5]  
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=True)  
norm(x) # [0., 0., 0.5, 1., 1.]
```

See Also

```
-----
```

:ref:`colormapnorms`

matplotlib.colors.from_levels_and_colors

```
from_levels_and_colors(levels, colors, extend='neither')
```

A helper routine to generate a cmap and a norm instance which behave similar to `contourf`'s `levels` and `colors` arguments.

Parameters

```
-----
```

`levels` : sequence of numbers

The quantization levels used to construct the `BoundaryNorm`.

Value `v` is quantized to level `i` if `lev[i] <= v < lev[i+1]`.

`colors` : sequence of colors

The fill color to use for each level. If `*extend*` is "neither" there must be `n_level - 1` colors. For an `*extend*` of "min" or "max" add one extra color, and for an `*extend*` of "both" add two colors.

`extend` : {'neither', 'min', 'max', 'both'}, optional

The behaviour when a value falls out of range of the given levels.

See `~.Axes.contourf` for details.

Returns

```
-----
```

`cmap` : `~matplotlib.colors.Colormap`
`norm` : `~matplotlib.colors.Normalize`

matplotlib.colors.get_named_colors_mapping

```
get_named_colors_mapping()
```

Return the global mapping of names to named colors.

matplotlib.colors.hex2color

```
hex2color(c)
```

Convert *c* to an RGB color, silently dropping the alpha channel.

matplotlib.colors.hsv_to_rgb

```
hsv_to_rgb(hsv)
```

Convert HSV values to RGB.

Parameters

hsv : (... , 3) array-like

All values assumed to be in range [0, 1]

Returns

(..., 3) `~numpy.ndarray`

Colors converted to RGB values in range [0, 1]

matplotlib.colors.is_color_like

```
is_color_like(c)
```

Return whether *c* can be interpreted as an RGB(A) color.

matplotlib.colors.make_norm_from_scale

```
make_norm_from_scale(scale_cls, base_norm_cls=None, *, init=None)
```

Decorator for building a `Normalize`` subclass from a `~.scale.ScaleBase`` subclass.

After ::

```
@make_norm_from_scale(scale_cls)
```

```
class norm_cls(Normalize):
```

```
...
```

norm_cls is filled with methods so that normalization computations are forwarded to *scale_cls* (i.e., *scale_cls* is the scale that would be used for the colorbar of a mappable normalized with *norm_cls*).

If *init* is not passed, then the constructor signature of *norm_cls* will be `norm_cls(vmin=None, vmax=None, clip=False)``; these three parameters will be forwarded to the base class (`Normalize.__init__``), and a *scale_cls* object will be initialized with no arguments (other than a dummy axis).

If the *scale_cls* constructor takes additional parameters, then *init* should be passed to `make_norm_from_scale``. It is a callable which is

only used for its signature. First, this signature will become the signature of *norm_cls*. Second, the *norm_cls* constructor will bind the parameters passed to it using this signature, extract the bound *vmin*, *vmax*, and *clip* values, pass those to ``Normalize.__init__``, and forward the remaining bound values (including any defaults defined by the signature) to the *scale_cls* constructor.

matplotlib.colors.rgb2hex

```
rgb2hex(c, keep_alpha=False)
```

Convert *c* to a hex color.

Parameters

c : :ref:`color <colors_def>` or `numpy.ma.masked`

keep_alpha : bool, default: False

If False, use the ``#rrggbb`` format, otherwise use ``#rrggbbaa``.

Returns

str

``#rrggbb`` or ``#rrggbbaa`` hex color string

matplotlib.colors.rgb_to_hsv

```
rgb_to_hsv(arr)
```

Convert an array of float RGB values (in the range [0, 1]) to HSV values.

Parameters

arr : (... , 3) array-like

All values must be in the range [0, 1]

Returns

(..., 3) ~numpy.ndarray`

Colors converted to HSV values in range [0, 1]

matplotlib.colors.same_color

```
same_color(c1, c2)
```

Return whether the colors *c1* and *c2* are the same.

c1, *c2* can be single colors or lists/arrays of colors.

matplotlib.colors.to_hex

```
to_hex(c, keep_alpha=False)
```

Convert *c* to a hex color.

Parameters

c : :ref:`color <colors_def>` or `numpy.ma.masked`

keep_alpha : bool, default: False

If False, use the ``#rrggbb`` format, otherwise use ``#rrggbbaa``.

Returns

str

``#rrggbb`` or ``#rrggbbaa`` hex color string

matplotlib.colors.to_rgb

```
to_rgb(c)
```

Convert *c* to an RGB color, silently dropping the alpha channel.

matplotlib.colors.to_rgba

```
to_rgba(c, alpha=None)
```

Convert *c* to an RGBA color.

Parameters

c : Matplotlib color or ``np.ma.masked``

alpha : float, optional

If *alpha* is given, force the alpha value of the returned RGBA tuple to *alpha*.

If None, the alpha value from *c* is used. If *c* does not have an alpha channel, then alpha defaults to 1.

alpha is ignored for the color value ``"none"`` (case-insensitive), which always maps to ``(0, 0, 0, 0)``.

Returns

tuple

Tuple of floats ``(r, g, b, a)`` , where each channel (red, green, blue, alpha) can assume values between 0 and 1.

matplotlib.colors.to_rgba_array

```
to_rgba_array(c, alpha=None)
```

Convert *c* to a (n, 4) array of RGBA colors.

Parameters

c : Matplotlib color or array of colors

If **c** is a masked array, an `~numpy.ndarray` is returned with a (0, 0, 0, 0) row for each masked value or row in **c**.

alpha : float or sequence of floats, optional

If **alpha** is given, force the alpha value of the returned RGBA tuple to **alpha**.

If None, the alpha value from **c** is used. If **c** does not have an alpha channel, then alpha defaults to 1.

alpha is ignored for the color value `""none""` (case-insensitive), which always maps to `“(0, 0, 0, 0)”`.

If **alpha** is a sequence and **c** is a single color, **c** will be repeated to match the length of **alpha**.

Returns

array

(n, 4) array of RGBA colors, where each channel (red, green, blue, alpha) can assume values between 0 and 1.

matplotlib.container

```
container(...)
```

No description available.

matplotlib.container.Artist

```
Artist()
```

Abstract base class for objects that render into a FigureCanvas.

Typically, all visible elements in a figure are subclasses of Artist.

matplotlib.container.BarContainer

```
BarContainer(patches, errorbar=None, *, datavalues=None, orientation=None, **kwargs)
```

Container for the artists of bar plots (e.g. created by `.Axes.bar``).

The container can be treated as a tuple of the **patches** themselves. Additionally, you can access these and further parameters by the attributes.

Attributes

patches : list of :class:`~matplotlib.patches.Rectangle`
The artists of the bars.

errorbar : None or :class:`~matplotlib.container.ErrorbarContainer`
A container for the error bar artists if error bars are present.
None otherwise.

datavalues : None or array-like
The underlying data values corresponding to the bars.

orientation : {'vertical', 'horizontal'}, default: None
If 'vertical', the bars are assumed to be vertical.
If 'horizontal', the bars are assumed to be horizontal.

matplotlib.container.Container

```
Container(*args, **kwargs)
```

Base class for containers.

Containers are classes that collect semantically related Artists such as the bars of a bar plot.

matplotlib.container.ErrorbarContainer

```
ErrorbarContainer(lines, has_xerr=False, has_yerr=False, **kwargs)
```

Container for the artists of error bars (e.g. created by `.Axes.errorbar`).

The container can be treated as the `*lines*` tuple itself.
Additionally, you can access these and further parameters by the attributes.

Attributes

lines : tuple

Tuple of ``(data_line, caplines, barlinecols)``.

- data_line : A `~matplotlib.lines.Line2D` instance of x, y plot markers and/or line.

- caplines : A tuple of `~matplotlib.lines.Line2D` instances of the error bar caps.

- barlinecols : A tuple of `~matplotlib.collections.LineCollection` with the horizontal and vertical error ranges.

has_xerr, has_yerr : bool

``True`` if the errorbar has x/y errors.

matplotlib.container.StemContainer

```
StemContainer(markerline_stemlines_baseline, **kwargs)
```

Container for the artists created in a `:meth:`.Axes.stem`` plot.

The container can be treated like a namedtuple ``(markerline, stemlines, baseline)``.

Attributes

markerline : `~matplotlib.lines.Line2D`

The artist of the markers at the stem heads.

stemlines : `~matplotlib.collections.LineCollection``
The artists of the vertical lines for all stems.

baseline : `~matplotlib.lines.Line2D``
The artist of the horizontal baseline.

matplotlib.contour

```
contour(...)
```

Classes to support contour plotting and labelling for the Axes class.

matplotlib.contour.ContourLabeler

```
ContourLabeler()
```

Mixin to provide labelling capability to `~.ContourSet``.

matplotlib.contour.ContourSet

```
ContourSet(ax, *args, levels=None, filled=False, linewidths=None, linestyle=None,
hatches=(None,), alpha=None, origin=None, extent=None, cmap=None, colors=None,
norm=None, vmin=None, vmax=None, colorizer=None, extend='neither', antialiased=None,
nchunk=0, locator=None, transform=None, negative_linestyle=None, clip_path=None,
**kwargs)
```

Store a set of contour lines or filled regions.

User-callable method: `~.Axes.clabel``

Parameters

ax : `~matplotlib.axes.Axes``

levels : [level0, level1, ..., levelN]

A list of floating point numbers indicating the contour levels.

allsegs : [level0segs, level1segs, ...]

List of all the polygon segments for all the `*levels*`.

For contour lines `len(allsegs) == len(levels)`, and for filled contour regions `len(allsegs) = len(levels)-1`. The lists should look like ::

level0segs = [polygon0, polygon1, ...]

polygon0 = [[x0, y0], [x1, y1], ...]

allkinds : `None`` or [level0kinds, level1kinds, ...]

Optional list of all the polygon vertex kinds (code types), as described and used in Path. This is used to allow multiply-connected paths such as holes within filled polygons.

If not `None``, `len(allkinds) == len(allsegs)`. The lists should look like ::

```
level0kinds = [polygon0kinds, ...]
polygon0kinds = [vertexcode0, vertexcode1, ...]
```

If **allkinds** is not ``None``, usually all polygons for a particular contour level are grouped together so that ``level0segs = [polygon0]`` and ``level0kinds = [polygon0kinds]``.

****kwargs**

Keyword arguments are as described in the docstring of `~.Axes.contour``.

Attributes

levels : array

The values of the contour levels.

layers : array

Same as levels for line contours; half-way between levels for filled contours. See ```ContourSet._process_colors```.

matplotlib.contour.Line2D

```
Line2D(xdata, ydata, *, linewidth=None, linestyle=None, color=None, gapcolor=None,
marker=None, markersize=None, markeredgewidth=None, markeredgewidth=None,
markerfacecolor=None, markerfacecoloralt='none', fillstyle=None, antialiased=None,
dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None,
pickradius=5, drawstyle=None, markevery=None, **kwargs)
```

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create "stepped" lines in various styles.

matplotlib.contour.MouseButton

```
MouseButton(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Enum where members are also (and must be) ints

matplotlib.contour.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- **vertices**: an (N, 2) float array of vertices
- **codes**: an N-length ``numpy.uint8`` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must

provide three vertices and three `CURVE4` codes.

The code types are:

- `STOP` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- `MOVETO` : 1 vertex

Pick up the pen and move to the given vertex.

- `LINETO` : 1 vertex

Draw a line from the current position to the given vertex.

- `CURVE3` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- `CURVE4` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- `CLOSEPOLY` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If *codes* is None, it is interpreted as a `MOVETO` followed by a series of `LINETO`.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use `iter_segments` or `cleaned` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of *codes* being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.contour.QuadContourSet

```
QuadContourSet(ax, *args, levels=None, filled=False, linewidths=None, linestyle=None,
hatches=(None,), alpha=None, origin=None, extent=None, cmap=None, colors=None,
norm=None, vmin=None, vmax=None, colorizer=None, extend='neither', antialiased=None,
nchunk=0, locator=None, transform=None, negative_linestyle=None, clip_path=None,
**kwargs)
```

Create and store a set of contour lines or filled regions.

This class is typically not instantiated directly by the user but by `~.Axes.contour` and `~.Axes.contourf`.

Attributes

levels : array

The values of the contour levels.

layers : array

Same as levels for line contours; half-way between

levels for filled contours. See ``ContourSet._process_colors``.

matplotlib.contour.Text

```
Text(x=0, y=0, text='', *, color=None, verticalalignment='baseline',
horizontalalignment='left', multialignment=None, fontproperties=None, rotation=None,
linespacing=None, rotation_mode=None, usetex=None, wrap=False,
transform_rotates_text=False, parse_math=None, antialiased=None, **kwargs)
```

Handle storing and drawing of text in window or data coordinates.

matplotlib.cycler

```
cycler(*args, **kwargs)
```

Create a ``~matplotlib.cycler.Cycler`` object much like :func:`matplotlib.cycler`, but includes input validation.

Call signatures::

cycler(cycler)

cycler(label=values, label2=values2, ...)

cycler(label, values)

Form 1 copies a given ``~matplotlib.cycler.Cycler`` object.

Form 2 creates a ``~matplotlib.cycler.Cycler`` which cycles over one or more properties simultaneously. If multiple properties are given, their value lists must have the same length.

Form 3 creates a ``~matplotlib.cycler.Cycler`` for a single property. This form exists for compatibility with the original cycler. Its use is discouraged in favor of the kwarg form, i.e. ``cycler(label=values)``.

Parameters

cycler : Cycler

Copy constructor for Cycler.

label : str

The property key. Must be a valid ``.Artist`` property.

For example, 'color' or 'linestyle'. Aliases are allowed, such as 'c' for 'color' and 'lw' for 'linewidth'.

values : iterable

Finite-length iterable of the property values. These values

are validated and will raise a `ValueError` if invalid.

Returns

Cycler

A new `:class:`~cycler.Cycler`` for the given properties.

Examples

Creating a cycler for a single property:

```
>>> c = cycler(color=['red', 'green', 'blue'])
```

Creating a cycler for simultaneously cycling over multiple properties
(e.g. red circle, green plus, blue cross):

```
>>> c = cycler(color=['red', 'green', 'blue'],  
... marker=['o', '+', 'x'])
```

matplotlib.dates

`dates(...)`

Matplotlib provides sophisticated date plotting capabilities, standing on the shoulders of python `:mod:`datetime`` and the add-on module `dateutil_`.

By default, Matplotlib uses the units machinery described in ``~matplotlib.units`` to convert ``datetime.datetime``, and ``numpy.datetime64`` objects when plotted on an x- or y-axis. The user does not need to do anything for dates to be formatted, but dates often have strict formatting needs, so this module provides many tick locators and formatters. A basic example using ``numpy.datetime64`` is::

```
import numpy as np
```

```
times = np.arange(np.datetime64('2001-01-02'),  
np.datetime64('2002-02-03'), np.timedelta64(75, 'm'))  
y = np.random.randn(len(times))
```

```
fig, ax = plt.subplots()  
ax.plot(times, y)
```

.. seealso::

- :doc:`/gallery/text_labels_and_annotations/date`
- :doc:`/gallery/ticks/date_concise_formatter`
- :doc:`/gallery/ticks/date_demo_convert`

.. _date-format:

Matplotlib date format

Matplotlib represents dates using floating point numbers specifying the number

of days since a default epoch of 1970-01-01 UTC; for example, 1970-01-01, 06:00 is the floating point number 0.25. The formatters and locators require the use of ``datetime.datetime`` objects, so only dates between year 0001 and 9999 can be represented. Microsecond precision is achievable for (approximately) 70 years on either side of the epoch, and 20 microseconds for the rest of the allowable range of dates (year 0001 to 9999). The epoch can be changed at import time via ``dates.set_epoch`` or `:rc:`date.epoch`` to other dates if necessary; see `:doc:`gallery/ticks/date_precision_and_epochs`` for a discussion.

.. note::

Before Matplotlib 3.3, the epoch was 0000-12-31 which lost modern microsecond precision and also made the default axis limit of 0 an invalid `datetime`. In 3.3 the epoch was changed as above. To convert old ordinal floats to the new epoch, users can do::

```
new_ordinal = old_ordinal + mdates.date2num(np.datetime64('0000-12-31'))
```

There are a number of helper functions to convert between `:mod:`datetime`` objects and Matplotlib dates:

.. currentmodule:: matplotlib.dates

.. autosummary::

:nosignatures:

datestr2num
date2num
num2date
num2timedelta
drange
set_epoch
get_epoch

.. note::

Like Python's ``datetime.datetime``, Matplotlib uses the Gregorian calendar for all conversions between dates and floating point numbers. This practice is not universal, and calendar differences can cause confusing differences between what Python and Matplotlib give as the number of days since 0001-01-01 and what other software and databases yield. For example, the US Naval Observatory uses a calendar that switches from Julian to Gregorian in October, 1582. Hence, using their calculator, the number of days between 0001-01-01 and 2006-04-01 is 732403, whereas using the Gregorian calendar via the `datetime` module we find::

```
In [1]: date(2006, 4, 1).toordinal() - date(1, 1, 1).toordinal()
Out[1]: 732401
```

All the Matplotlib date converters, locators and formatters are timezone aware. If no explicit timezone is provided, `:rc:`timezone`` is assumed, provided as a string. If you want to use a different timezone, pass the `*tz*` keyword

argument of ``num2date`` to any date tick locators or formatters you create. This can be either a ``datetime.tzinfo`` instance or a string with the timezone name that can be parsed by ``~dateutil.tz.gettz``.

A wide range of specific and general purpose date tick locators and formatters are provided in this module. See `:mod:`matplotlib.ticker`` for general information on tick locators and formatters. These are described below.

The `dateutil_` module provides additional code to handle date ticking, making it easy to place ticks on any kinds of dates. See examples below.

.. `_dateutil`: <https://dateutil.readthedocs.io>

.. `_date-locators`:

Date tick locators

Most of the date tick locators can locate single or multiple ticks. For example::

```
# import constants for the days of the week
from matplotlib.dates import MO, TU, WE, TH, FR, SA, SU
```

```
# tick on Mondays every week
loc = WeekdayLocator(byweekday=MO, tz=tz)
```

```
# tick on Mondays and Saturdays
loc = WeekdayLocator(byweekday=(MO, SA))
```

In addition, most of the constructors take an interval argument::

```
# tick on Mondays every second week
loc = WeekdayLocator(byweekday=MO, interval=2)
```

The `rrule` locator allows completely general date ticking::

```
# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
```

The available date tick locators are:

- * ``MicrosecondLocator``: Locate microseconds.
- * ``SecondLocator``: Locate seconds.
- * ``MinuteLocator``: Locate minutes.
- * ``HourLocator``: Locate hours.
- * ``DayLocator``: Locate specified days of the month.
- * ``WeekdayLocator``: Locate days of the week, e.g., MO, TU.

* `MonthLocator`: Locate months, e.g., 7 for July.

* `YearLocator`: Locate years that are multiples of base.

* `RRuleLocator`: Locate using a `rrulewrapper`.

`rrulewrapper` is a simple wrapper around `dateutil's dateutil.rrule` which allow almost arbitrary date tick specifications.

See `:doc:rrule example </gallery/ticks/date_demo_rrule>`.

* `AutoDateLocator`: On autoscale, this class picks the best `DateLocator` (e.g., `RRuleLocator`) to set the view limits and the tick locations. If called with `interval_multiples=True` it will make ticks line up with sensible multiples of the tick intervals. For example, if the interval is 4 hours, it will pick hours 0, 4, 8, etc. as ticks. This behaviour is not guaranteed by default.

.. `_date-formatters`:

Date formatters

The available date formatters are:

* `AutoDateFormatter`: attempts to figure out the best format to use. This is most useful when used with the `AutoDateLocator`.

* `ConciseDateFormatter`: also attempts to figure out the best format to use, and to make the format as compact as possible while still having complete date information. This is most useful when used with the `AutoDateLocator`.

* `DateFormatter`: use `~datetime.datetime.strftime` format strings.

matplotlib.dates.AutoDateFormatter

```
AutoDateFormatter(locator, tz=None, defaultfmt='%Y-%m-%d', *, usetex=None)
```

A `.Formatter` which attempts to figure out the best format to use. This is most useful when used with the `AutoDateLocator`.

`.AutoDateFormatter` has a `.scale` dictionary that maps tick scales (the interval in days between one major tick) to format strings; this dictionary defaults to ::

```
self.scaled = {
    DAYS_PER_YEAR: rcParams['date.autoformatter.year'],
    DAYS_PER_MONTH: rcParams['date.autoformatter.month'],
    1: rcParams['date.autoformatter.day'],
    1 / HOURS_PER_DAY: rcParams['date.autoformatter.hour'],
    1 / MINUTES_PER_DAY: rcParams['date.autoformatter.minute'],
    1 / SEC_PER_DAY: rcParams['date.autoformatter.second'],
    1 / MUSECONDS_PER_DAY: rcParams['date.autoformatter.microsecond'],
}
```

The formatter uses the format string corresponding to the lowest key in

the dictionary that is greater or equal to the current scale. Dictionary entries can be customized::

```
locator = AutoDateLocator()
formatter = AutoDateFormatter(locator)
formatter.scaled[1/(24*60)] = '%M:%S' # only show min and sec
```

Custom callables can also be used instead of format strings. The following example shows how to use a custom format function to strip trailing zeros from decimal seconds and adds the date to the first ticklabel::

```
def my_format_function(x, pos=None):
    x = matplotlib.dates.num2date(x)
    if pos == 0:
        fmt = '%D %H:%M:%S.%f'
    else:
        fmt = '%H:%M:%S.%f'
    label = x.strftime(fmt)
    label = label.rstrip("0")
    label = label.rstrip(".")
    return label

formatter.scaled[1/(24*60)] = my_format_function
```

matplotlib.dates.AutoDateLocator

```
AutoDateLocator(tz=None, minticks=5, maxticks=None, interval_multiples=True)
```

On autoscale, this class picks the best `DateLocator` to set the view limits and the tick locations.

Attributes

interval : dict

Mapping of tick frequencies to multiples allowed for that ticking. The default is ::

```
self.interval = {
    YEARLY : [1, 2, 4, 5, 10, 20, 40, 50, 100, 200, 400, 500,
    1000, 2000, 4000, 5000, 10000],
    MONTHLY : [1, 2, 3, 4, 6],
    DAILY : [1, 2, 3, 7, 14, 21],
    HOURLY : [1, 2, 3, 4, 6, 12],
    MINUTELY : [1, 5, 10, 15, 30],
    SECONDLY : [1, 5, 10, 15, 30],
    MICROSECONDLY : [1, 2, 5, 10, 20, 50, 100, 200, 500,
    1000, 2000, 5000, 10000, 20000, 50000,
    100000, 200000, 500000, 1000000],
}
```

where the keys are defined in `dateutil.rrule`.

The interval is used to specify multiples that are appropriate for

the frequency of ticking. For instance, every 7 days is sensible for daily ticks, but for minutes/seconds, 15 or 30 make sense.

When customizing, you should only modify the values for the existing keys. You should not add or delete entries.

Example for forcing ticks every 3 hours::

```
locator = AutoDateLocator()
locator.intervald[HOURLY] = [3] # only show every 3 hours
```

matplotlib.dates.ConciseDateConverter

```
ConciseDateConverter(formats=None, zero_formats=None, offset_formats=None,
show_offset=True, *, interval_multiples=True)
```

Converter for ``datetime.date`` and ``datetime.datetime`` data, or for date/time data represented as it would be converted by ``date2num``.

The 'unit' tag for such data is None or a ``~datetime.tzinfo`` instance.

matplotlib.dates.ConciseDateFormatter

```
ConciseDateFormatter(locator, tz=None, formats=None, offset_formats=None,
zero_formats=None, show_offset=True, *, usetex=None)
```

A ``Formatter`` which attempts to figure out the best format to use for the date, and to make it as compact as possible, but still be complete. This is most useful when used with the ``AutoDateLocator``:

```
>>> locator = AutoDateLocator()
>>> formatter = ConciseDateFormatter(locator)
```

Parameters

locator : ``ticker.Locator``

Locator that this axis is using.

tz : str or ``~datetime.tzinfo``, default: `:rc:`timezone``

Ticks timezone, passed to ``dates.num2date``.

formats : list of 6 strings, optional

Format strings for 6 levels of tick labelling: mostly years, months, days, hours, minutes, and seconds. Strings use the same format codes as ``~datetime.datetime.strftime``. Default is ``[%Y', '%b', '%d', '%H:%M', '%H:%M', '%S.%f']``

zero_formats : list of 6 strings, optional

Format strings for tick labels that are "zeros" for a given tick level. For instance, if most ticks are months, ticks around 1 Jan 2005 will be labeled "Dec", "2005", "Feb". The default is ``['', '%Y', '%b', '%b-%d', '%H:%M', '%H:%M']``

offset_formats : list of 6 strings, optional

Format strings for the 6 levels that is applied to the "offset"

string found on the right side of an x-axis, or top of a y-axis.
Combined with the tick labels this should completely specify the date. The default is::

```
['', '%Y', '%Y-%b', '%Y-%b-%d', '%Y-%b-%d', '%Y-%b-%d %H:%M']
```

`show_offset` : bool, default: True
Whether to show the offset or not.

`usetex` : bool, default: :rc:`text.usetex`
To enable/disable the use of TeX's math mode for rendering the results of the formatter.

Examples

See :doc:`/gallery/ticks/date_concise_formatter`

.. plot::

```
import datetime
import matplotlib.dates as mdates

base = datetime.datetime(2005, 2, 1)
dates = np.array([base + datetime.timedelta(hours=(2 * i))
for i in range(732)])
N = len(dates)
np.random.seed(19680801)
y = np.cumsum(np.random.randn(N))

fig, ax = plt.subplots(constrained_layout=True)
locator = mdates.AutoDateLocator()
formatter = mdates.ConciseDateFormatter(locator)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)

ax.plot(dates, y)
ax.set_title('Concise Date Formatter')
```

matplotlib.dates.DateConverter

```
DateConverter(*, interval_multiples=True)
```

Converter for `datetime.date` and `datetime.datetime` data, or for date/time data represented as it would be converted by `date2num`.

The 'unit' tag for such data is None or a `~datetime.tzinfo` instance.

matplotlib.dates.DateFormatter

```
DateFormatter(fmt, tz=None, *, usetex=None)
```

Format a tick (in days since the epoch) with a `~datetime.datetime.strftime` format string.

matplotlib.dates.DateLocator

```
DateLocator(tz=None)
```

Determines the tick locations when plotting dates.

This class is subclassed by other Locators and is not meant to be used on its own.

matplotlib.dates.DayLocator

```
DayLocator(bymonthday=None, interval=1, tz=None)
```

Make ticks on occurrences of each day of the month. For example, 1, 15, 30.

matplotlib.dates.HourLocator

```
HourLocator(byhour=None, interval=1, tz=None)
```

Make ticks on occurrences of each hour.

matplotlib.dates.MicrosecondLocator

```
MicrosecondLocator(interval=1, tz=None)
```

Make ticks on regular intervals of one or more microsecond(s).

.. note::

By default, Matplotlib uses a floating point representation of time in days since the epoch, so plotting data with microsecond time resolution does not work well for dates that are far (about 70 years) from the epoch (check with ``~.dates.get_epoch``).

If you want sub-microsecond resolution time plots, it is strongly recommended to use floating point seconds, not datetime-like time representation.

If you really must use `datetime.datetime()` or similar and still need microsecond precision, change the time origin via ``~.dates.set_epoch`` to something closer to the dates being plotted. See `:doc:`gallery/ticks/date_precision_and_epochs``.

matplotlib.dates.MinuteLocator

```
MinuteLocator(byminute=None, interval=1, tz=None)
```

Make ticks on occurrences of each minute.

matplotlib.dates.MonthLocator

```
MonthLocator(bymonth=None, bymonthday=1, interval=1, tz=None)
```

Make ticks on occurrences of each month, e.g., 1, 3, 12.

matplotlib.dates.RRuleLocator

```
RRuleLocator(o, tz=None)
```

Determines the tick locations when plotting dates.

This class is subclassed by other Locators and is not meant to be used on its own.

matplotlib.dates.SecondLocator

```
SecondLocator(bysecond=None, interval=1, tz=None)
```

Make ticks on occurrences of each second.

matplotlib.dates.WeekdayLocator

```
WeekdayLocator(byweekday=1, interval=1, tz=None)
```

Make ticks on occurrences of each weekday.

matplotlib.dates.YearLocator

```
YearLocator(base=1, month=1, day=1, tz=None)
```

Make ticks on a given day of each year that is a multiple of base.

Examples::

```
# Tick every year on Jan 1st
locator = YearLocator()
```

```
# Tick every 5 years on July 4th
locator = YearLocator(5, month=7, day=4)
```

matplotlib.dates.date2num

```
date2num(d)
```

Convert datetime objects to Matplotlib dates.

Parameters

d : ``datetime.datetime`` or ``numpy.datetime64`` or sequences of these

Returns

float or sequence of floats

Number of days since the epoch. See ``.get_epoch`` for the epoch, which can be changed by `:`rc:`date.epoch`` or ``.set_epoch``. If the epoch is "1970-01-01T00:00:00" (default) then noon Jan 1 1970 ("1970-01-01T12:00:00") returns 0.5.

Notes

The Gregorian calendar is assumed; this is not universal practice.
For details see the module docstring.

matplotlib.dates.datestr2num

```
datestr2num(d, default=None)
```

Convert a date string to a datenum using ``dateutil.parser.parse``.

Parameters

d : str or sequence of str
The dates to convert.

default : `datetime.datetime`, optional
The default date to use when fields are missing in `*d*`.

matplotlib.dates.drange

```
drange(dstart, dend, delta)
```

Return a sequence of equally spaced Matplotlib dates.

The dates start at `*dstart*` and reach up to, but not including `*dend*`.
They are spaced by `*delta*`.

Parameters

dstart, dend : ``~datetime.datetime``
The date limits.
delta : ``datetime.timedelta``
Spacing of the dates.

Returns

``numpy.array``
A list floats representing Matplotlib dates.

matplotlib.dates.get_epoch

```
get_epoch()
```

Get the epoch used by ``dates``.

Returns

epoch : str
String for the epoch (parsable by ``numpy.datetime64``).

matplotlib.dates.num2date

```
num2date(x, tz=None)
```

Convert Matplotlib dates to `~datetime.datetime`` objects.

Parameters

`x` : float or sequence of floats

Number of days (fraction part represents hours, minutes, seconds) since the epoch. See `~datetime.datetime.get_epoch`` for the

epoch, which can be changed by `:rc:`date.epoch`` or `~datetime.datetime.set_epoch``.

`tz` : str or `~datetime.tzinfo``, default: `:rc:`timezone``

Timezone of `*x*`. If a string, `*tz*` is passed to `dateutil.tz``.

Returns

`~datetime.datetime`` or sequence of `~datetime.datetime``

Dates are returned in timezone `*tz*`.

If `*x*` is a sequence, a sequence of `~datetime.datetime`` objects will be returned.

Notes

The Gregorian calendar is assumed; this is not universal practice.

For details, see the module docstring.

matplotlib.dates.num2timedelta

```
num2timedelta(x)
```

Convert number of days to a `~datetime.timedelta`` object.

If `*x*` is a sequence, a sequence of `~datetime.timedelta`` objects will be returned.

Parameters

`x` : float, sequence of floats

Number of days. The fraction part represents hours, minutes, seconds.

Returns

`~datetime.timedelta`` or list[`~datetime.timedelta``]

matplotlib.dates.rrulewrapper

```
rrulewrapper(freq, tzinfo=None, **kwargs)
```

A simple wrapper around a `dateutil.rrule`` allowing flexible date tick specifications.

matplotlib.dates.set_epoch

```
set_epoch(epoch)
```

Set the epoch (origin for dates) for datetime calculations.

The default epoch is `:rc:`date.epoch``.

If microsecond accuracy is desired, the date being plotted needs to be within approximately 70 years of the epoch. Matplotlib internally represents dates as days since the epoch, so floating point dynamic range needs to be within a factor of 2^{52} .

``~.dates.set_epoch`` must be called before any dates are converted (i.e. near the import section) or a `RuntimeError` will be raised.

See also `:doc:`/gallery/ticks/date_precision_and_epochs``.

Parameters

epoch : str
valid UTC date parsable by ``numpy.datetime64`` (do not include timezone).

matplotlib.dviread

`dviread(...)`

A module for reading dvi files output by TeX. Several limitations make this not (currently) useful as a general-purpose dvi preprocessor, but it is currently used by the pdf backend for processing usetex text.

Interface::

```
with Dvi(filename, 72) as dvi:
# iterate over pages:
for page in dvi:
w, h, d = page.width, page.height, page.descent
for x, y, font, glyph, width in page.text:
fontname = font.textname
pointsize = font.size
...
for x, y, height, width in page.bboxes:
...
```

matplotlib.dviread.Box

`Box(x, y, height, width)`

`Box(x, y, height, width)`

matplotlib.dviread.Dvi

`Dvi(filename, dpi)`

A reader for a dvi ("device-independent") file, as produced by TeX.

The current implementation can only iterate through pages in order, and does not even attempt to verify the postamble.

This class can be used as a context manager to close the underlying file upon exit. Pages can be read via iteration. Here is an overly simple way to extract text without trying to detect whitespace::

```
>>> with matplotlib.dviread.Dvi('input.dvi', 72) as dvi:
...     for page in dvi:
...         print("".join(chr(t.glyph) for t in page.text))
```

matplotlib.dviread.DviFont

```
DviFont(scale, tfm, texname, vf)
```

Encapsulation of a font that a DVI file can refer to.

This class holds a font's texname and size, supports comparison, and knows the widths of glyphs in the same units as the AFM file. There are also internal attributes (for use by dviread.py) that are *not* used for comparison.

The size is in Adobe points (converted from TeX points).

Parameters

scale : float

Factor by which the font is scaled from its natural size.

tfm : Tfm

TeX font metrics for this font

texname : bytes

Name of the font as used internally by TeX and friends, as an ASCII bytestring. This is usually very different from any external font names; ``PsfontsMap`` can be used to find the external name of the font.

vf : Vf

A TeX "virtual font" file, or None if this font is not virtual.

Attributes

texname : bytes

size : float

Size of the font in Adobe points, converted from the slightly smaller TeX points.

widths : list

Widths of glyphs in glyph-space units, typically 1/1000ths of the point size.

matplotlib.dviread.Page

```
Page(text, boxes, height, width, descent)
```

Page(text, boxes, height, width, descent)

matplotlib.dviread.PsFont

```
PsFont(texname, psname, effects, encoding, filename)
```

PsFont(texname, psname, effects, encoding, filename)

matplotlib.dviread.PsfontsMap

PsfontsMap(filename)

A psfonts.map formatted file, mapping TeX fonts to PS fonts.

Parameters

filename : str or path-like

Notes

For historical reasons, TeX knows many Type-1 fonts by different names than the outside world. (For one thing, the names have to fit in eight characters.) Also, TeX's native fonts are not Type-1 but Metafont, which is nontrivial to convert to PostScript except as a bitmap. While high-quality conversions to Type-1 format exist and are shipped with modern TeX distributions, we need to know which Type-1 fonts are the counterparts of which native fonts. For these reasons a mapping is needed from internal font names to font file names.

A texmf tree typically includes mapping files called e.g. :file:`psfonts.map`, :file:`pdftex.map`, or :file:`dvipdfm.map`. The file :file:`psfonts.map` is used by :program:`dvips`, :file:`pdftex.map` by :program:`pdfTeX`, and :file:`dvipdfm.map` by :program:`dvipdfm`. :file:`psfonts.map` might avoid embedding the 35 PostScript fonts (i.e., have no filename for them, as in the Times-Bold example above), while the pdf-related files perhaps only avoid the "Base 14" pdf fonts. But the user may have configured these files differently.

Examples

```
>>> map = PsfontsMap(find_tex_file('pdftex.map'))
>>> entry = map[b'ptmbo8r']
>>> entry.texname
b'ptmbo8r'
>>> entry.psname
b'Times-Bold'
>>> entry.encoding
'/usr/local/texlive/2008/texmf-dist/fonts/enc/dvips/base/8r.enc'
>>> entry.effects
{'slant': 0.16700000000000001}
>>> entry.filename
```

matplotlib.dviread.Text

Text(x, y, font, glyph, width)

A glyph in the dvi file.

The *x* and *y* attributes directly position the glyph. The *font*,

`*glyph*`, and `*width*` attributes are kept public for back-compatibility, but users wanting to draw the glyph themselves are encouraged to instead load the font specified by ``font_path`` at ``font_size``, warp it with the effects specified by ``font_effects``, and load the glyph specified by ``glyph_name_or_index``.

matplotlib.dviread.Tfm

```
Tfm(filename)
```

A TeX Font Metric file.

This implementation covers only the bare minimum needed by the Dvi class.

Parameters

filename : str or path-like

Attributes

checksum : int

Used for verifying against the dvi file.

design_size : int

Design size of the font (unknown units)

width, height, depth : dict

Dimensions of each character, need to be scaled by the factor specified in the dvi file. These are dicts because indexing may not start from 0.

matplotlib.dviread.Vf

```
Vf(filename)
```

A virtual font (`*.vf` file) containing subroutines for dvi files.

Parameters

filename : str or path-like

Notes

The virtual font format is a derivative of dvi:

<http://mirrors.ctan.org/info/knuth/virtual-fonts>

This class reuses some of the machinery of ``Dvi`` but replaces the ``_read`` loop and dispatch mechanism.

Examples

::

```
vf = Vf(filename)
```

```
glyph = vf[code]
```

```
glyph.text, glyph.bboxes, glyph.width
```

matplotlib.figure

```
figure(...)
```

`matplotlib.figure` implements the following classes:

`Figure`

Top level `~matplotlib.artist.Artist`, which holds all plot elements.

Many methods are implemented in `FigureBase`.

`SubFigure`

A logical figure inside a figure, usually added to a figure (or parent `SubFigure`)

with `Figure.add_subfigure` or `Figure.subfigures` methods.

Figures are typically created using pyplot methods `~.pyplot.figure`,

`~.pyplot.subplots`, and `~.pyplot.subplot_mosaic`.

.. plot::

:include-source:

```
fig, ax = plt.subplots(figsize=(2, 2), facecolor='lightskyblue',
layout='constrained')
fig.suptitle('Figure')
ax.set_title('Axes', loc='left', fontstyle='oblique', fontsize='medium')
```

Some situations call for directly instantiating a `~.figure.Figure` class, usually inside an application of some sort (see :ref:`user_interfaces` for a list of examples) . More information about Figures can be found at :ref:`figure-intro`.

matplotlib.figure.Affine2D

```
Affine2D(matrix=None, **kwargs)
```

A mutable 2D affine transformation.

matplotlib.figure.Artist

```
Artist()
```

Abstract base class for objects that render into a `FigureCanvas`.

Typically, all visible elements in a figure are subclasses of `Artist`.

matplotlib.figure.Axes

```
Axes(fig, *args, facecolor=None, frameon=True, sharex=None, sharey=None, label='',
xscale=None, yscale=None, box_aspect=None, forward_navigation_events='auto', **kwargs)
```

An `Axes` object encapsulates all the elements of an individual (sub-)plot in a figure.

It contains most of the (sub-)plot elements: `~.axis.Axis`, `~.axis.Tick`, `~.lines.Line2D`, `~.text.Text`, `~.patches.Polygon`, etc., and sets the coordinate system.

Like all visible elements in a figure, Axes is an `.Artist` subclass.

The `Axes` instance supports callbacks through a `callbacks` attribute which is a `~.cbook.CallbackRegistry` instance. The events you can connect to are `'xlim_changed'` and `'ylim_changed'` and the callback will be called with `func(*ax*)` where `*ax*` is the `Axes` instance.

.. note::

As a user, you do not instantiate Axes directly, but use Axes creation methods instead; e.g. from `.pyplot` or `.Figure`:

`~.pyplot.subplots`, `~.pyplot.subplot_mosaic` or `.Figure.add_axes`.

matplotlib.figure.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" `[[xmin, ymin], [xmax, ymax]]`.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" `(xmin, ymin, xmax, ymax)`

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" `(xmin, ymin, width, height)`.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

```
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting ``ignore=True`` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify ``ignore`` explicitly. If not, the default value of ``ignore`` can be changed at any time by code with access to your Bbox, for example using the method `~.Bbox.ignore``.

****Properties of the ``null`` bbox****

.. note::

The current behavior of `Bbox.null()` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[-inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[-inf, -inf], [inf, inf]])
```

matplotlib.figure.BboxTransformTo

```
BboxTransformTo(boxout, **kwargs)
```

`BboxTransformTo`` is a transformation that linearly transforms points from the unit bounding box to a given `Bbox``.

matplotlib.figure.ConstrainedLayoutEngine

```
ConstrainedLayoutEngine(*, h_pad=None, w_pad=None, hspace=None, wspace=None, rect=(0, 0, 1, 1), compress=False, **kwargs)
```

Implements the `constrained_layout`` geometry management. See `:ref:constrainedlayout_guide`` for details.

matplotlib.figure.DrawEvent

```
DrawEvent(name, canvas, renderer)
```

An event triggered by a draw operation on the canvas.

In most backends, callbacks subscribed to this event will be fired after the rendering is complete but before the screen is updated. Any extra artists drawn to the canvas's renderer will be reflected without an explicit call to ``blit``.

.. warning::

Calling ``canvas.draw`` and ``canvas.blit`` in these callbacks may not be safe with all backends and may cause infinite recursion.

A DrawEvent has a number of special attributes in addition to those defined by the parent `Event` class.

Attributes

renderer : `RendererBase`

The renderer for the draw event.

matplotlib.figure.Figure

```
Figure(figsize=None, dpi=None, *, facecolor=None, edgecolor=None, linewidth=0.0, frameon=None, subplotpars=None, tight_layout=None, constrained_layout=None, layout=None, **kwargs)
```

The top level container for all the plot elements.

See `matplotlib.figure` for an index of class methods.

Attributes

patch

The `.Rectangle` instance representing the figure background patch.

suppressComposite

For multiple images, the figure will make composite images depending on the renderer option_image_nocomposite function. If *suppressComposite* is a boolean, this will override the renderer.

matplotlib.figure.FigureBase

```
FigureBase(**kwargs)
```

Base class for `.Figure` and `.SubFigure` containing the methods that add artists to the figure or subfigure, create Axes, etc.

matplotlib.figure.FigureCanvasBase

```
FigureCanvasBase(figure=None)
```

The canvas the figure renders into.

Attributes

figure : `~matplotlib.figure.Figure``
A high-level figure instance.

matplotlib.figure.GridSpec

```
GridSpec(nrows, ncols, figure=None, left=None, bottom=None, right=None, top=None,
         wspace=None, hspace=None, width_ratios=None, height_ratios=None)
```

A grid layout to place subplots within a figure.

The location of the grid cells is determined in a similar way to `~.SubplotParams`` using `*left*`, `*right*`, `*top*`, `*bottom*`, `*wspace*` and `*hspace*`.

Indexing a `GridSpec` instance returns a `~.SubplotSpec``.

matplotlib.figure.LayoutEngine

```
LayoutEngine(**kwargs)
```

Base class for Matplotlib layout engines.

A layout engine can be passed to a figure at instantiation or at any time with `~.figure.Figure.set_layout_engine``. Once attached to a figure, the layout engine ```execute``` function is called at draw time by `~.figure.Figure.draw``, providing a special draw-time hook.

.. note::

However, note that layout engines affect the creation of colorbars, so `~.figure.Figure.set_layout_engine`` should be called before any colorbars are created.

Currently, there are two properties of `~LayoutEngine`` classes that are consulted while manipulating the figure:

- ```engine.colorbar_gridspec``` tells `~.Figure.colorbar`` whether to make the axes using the `gridspec` method (see `~.colorbar.make_axes_gridspec``) or not (see `~.colorbar.make_axes``);
- ```engine.adjust_compatible``` stops `~.Figure.subplots_adjust`` from being run if it is not compatible with the layout engine.

To implement a custom `~LayoutEngine``:

1. override ```_adjust_compatible``` and ```_colorbar_gridspec```
2. override `~LayoutEngine.set`` to update `*self._params*`
3. override `~LayoutEngine.execute`` with your implementation

matplotlib.figure.MouseButton

```
MouseButton(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Enum where members are also (and must be) ints

matplotlib.figure.NonGuiException

```
NonGuiException(...)
```

Raised when trying show a figure in a non-GUI backend.

matplotlib.figure.PlaceHolderLayoutEngine

```
PlaceHolderLayoutEngine(adjust_compatible, colorbar_gridspec, **kwargs)
```

This layout engine does not adjust the figure layout at all.

The purpose of this `.LayoutEngine`` is to act as a placeholder when the user removes a layout engine to ensure an incompatible `.LayoutEngine`` cannot be set later.

Parameters

`adjust_compatible, colorbar_gridspec` : bool

Allow the PlaceHolderLayoutEngine to mirror the behavior of whatever layout engine it is replacing.

matplotlib.figure.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point `*xy*` and its `*width*` and `*height*`.

The rectangle extends from ```xy[0]``` to ```xy[0] + width``` in x-direction and from ```xy[1]``` to ```xy[1] + height``` in y-direction. ::

```
: +-----+
: |
: height |
: |
: (xy)--- width ----+
```

One may picture `*xy*` as the bottom left corner, but which corner `*xy*` is actually depends on the direction of the axis and the sign of `*width*` and `*height*`; e.g. `*xy*` would be the bottom right corner if the x-axis was inverted or if `*width*` was negative.

matplotlib.figure.SubFigure

```
SubFigure(parent, subplotspec, *, facecolor=None, edgecolor=None, linewidth=0.0,
frameon=None, **kwargs)
```

Logical figure that can be placed inside a figure.

See :ref:`figure-api-subfigure` for an index of methods on this class.

Typically instantiated using `.Figure.add_subfigure`` or

`.SubFigure.add_subfigure``, or `.SubFigure.subfigures``. A subfigure has

the same methods as a figure except for those particularly tied to the size or dpi of the figure, and is confined to a prescribed region of the figure. For example the following puts two subfigures side-by-side::

```
fig = plt.figure()
sfigs = fig.subfigures(1, 2)
axsL = sfigs[0].subplots(1, 2)
axsR = sfigs[1].subplots(2, 1)
```

See :doc:`/gallery/subplots_axes_and_figures/subfigures`

matplotlib.figure.SubplotParams

```
SubplotParams(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
```

Parameters defining the positioning of a subplots grid in a figure.

matplotlib.figure.Text

```
Text(x=0, y=0, text='', *, color=None, verticalalignment='baseline',
horizontalalignment='left', multialignment=None, fontproperties=None, rotation=None,
linespacing=None, rotation_mode=None, usetex=None, wrap=False,
transform_rotates_text=False, parse_math=None, antialiased=None, **kwargs)
```

Handle storing and drawing of text in window or data coordinates.

matplotlib.figure.TightLayoutEngine

```
TightLayoutEngine(*, pad=1.08, h_pad=None, w_pad=None, rect=(0, 0, 1, 1), **kwargs)
```

Implements the ``tight_layout`` geometry management. See :ref:`tight_layout_guide` for details.

matplotlib.figure.TransformedBbox

```
TransformedBbox(bbox, transform, **kwargs)
```

A `Bbox` that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.

matplotlib.figure.allow_rasterization

```
allow_rasterization(draw)
```

Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependent renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

matplotlib.figure.figaspect

```
figaspect(arg)
```

Calculate the width and height for a figure with a specified aspect ratio.

While the height is taken from `rc('figure.figsize')`, the width is adjusted to match the desired aspect ratio. Additionally, it is ensured that the width is in the range `[4., 16.]` and the height is in the range `[2., 16.]`. If necessary, the default height is adjusted to ensure this.

Parameters

`arg` : float or 2D array

If a float, this defines the aspect ratio (i.e. the ratio height / width).

In case of an array the aspect ratio is number of rows / number of columns, so that the array could be fitted in the figure undistorted.

Returns

`size` : (2,) array

The width and height of the figure in inches.

Notes

If you want to create an Axes within the figure, that still preserves the aspect ratio, be sure to create it with equal width and height. See examples below.

Thanks to Fernando Perez for this function.

Examples

Make a figure twice as tall as it is wide::

```
w, h = figaspect(2.)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

Make a figure with the proper aspect for an array::

```
A = rand(5, 3)
w, h = figaspect(A)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

matplotlib.font_manager

```
font_manager(...)
```

A module for finding, managing, and using fonts across platforms.

This module provides a single `FontManager` instance, `fontManager`, that can be shared across backends and platforms. The `findfont` function returns the best TrueType (TTF) font file in the local or system font path that matches the specified `FontProperties`

instance. The `FontManager` also handles Adobe Font Metrics (AFM) font files for use by the PostScript backend. The `FontManager.addfont` function adds a custom font from a file without installing it into your operating system.

The design is based on the W3C Cascading Style Sheet, Level 1 (CSS1) font specification <<http://www.w3.org/TR/1998/REC-CSS2-19980512/>>_. Future versions may implement the Level 2 or 2.1 specifications.

matplotlib.font_manager.FontEntry

```
FontEntry(fname: 'str' = '', name: 'str' = '', style: 'str' = 'normal', variant: 'str' = 'normal', weight: 'str | int' = 'normal', stretch: 'str' = 'normal', size: 'str' = 'medium') -> None
```

A class for storing Font properties.

It is used when populating the font lookup dictionary.

matplotlib.font_manager.FontManager

```
FontManager(size=None, weight='normal')
```

On import, the `FontManager` singleton instance creates a list of ttf and afm fonts and caches their `FontProperties`. The `FontManager.findfont` method does a nearest neighbor search to find the font that most closely matches the specification. If no good enough match is found, the default font is returned.

Fonts added with the `FontManager.addfont` method will not persist in the cache; therefore, `addfont` will need to be called every time Matplotlib is imported. This method should only be used if and when a font cannot be installed on your operating system by other means.

Notes

The `FontManager.addfont` method must be called on the global `FontManager` instance.

Example usage::

```
import matplotlib.pyplot as plt
from matplotlib import font_manager

font_dirs = ["/resources/fonts"] # The path to the custom font file.
font_files = font_manager.findSystemFonts(fontpaths=font_dirs)

for font_file in font_files:
    font_manager.fontManager.addfont(font_file)
```

matplotlib.font_manager.FontProperties

```
FontProperties(family=None, style=None, variant=None, weight=None, stretch=None, size=None, fname=None, math_fontfamily=None)
```

A class for storing and manipulating font properties.

The font properties are the six properties described in the
`W3C Cascading Style Sheet, Level 1`
<<http://www.w3.org/TR/1998/REC-CSS2-19980512/>>` _ font`
specification and `*math_fontfamily*` for math fonts:

- family: A list of font names in decreasing order of priority.
The items may include a generic font family name, either 'sans-serif',
'serif', 'cursive', 'fantasy', or 'monospace'. In that case, the actual
font to be used will be looked up from the associated rcParam during the
search process in `.findfont``. Default: `:rc:`font.family``

- style: Either 'normal', 'italic' or 'oblique'.
Default: `:rc:`font.style``

- variant: Either 'normal' or 'small-caps'.
Default: `:rc:`font.variant``

- stretch: A numeric value in the range 0-1000 or one of
'ultra-condensed', 'extra-condensed', 'condensed',
'semi-condensed', 'normal', 'semi-expanded', 'expanded',
'extra-expanded' or 'ultra-expanded'. Default: `:rc:`font.stretch``

- weight: A numeric value in the range 0-1000 or one of
'ultralight', 'light', 'normal', 'regular', 'book', 'medium',
'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy',
'extra bold', 'black'. Default: `:rc:`font.weight``

- size: Either a relative value of 'xx-small', 'x-small',
'small', 'medium', 'large', 'x-large', 'xx-large' or an
absolute font size, e.g., 10. Default: `:rc:`font.size``

- math_fontfamily: The family of fonts used to render math text.
Supported values are: 'dejavusans', 'dejavuserif', 'cm',
'stix', 'stixsans' and 'custom'. Default: `:rc:`mathtext.fontset``

Alternatively, a font may be specified using the absolute path to a font
file, by using the `*fname*` kwarg. However, in this case, it is typically
simpler to just pass the path (as a ``pathlib.Path``, not a ``str``) to the
`*font*` kwarg of the `.Text`` object.

The preferred usage of font sizes is to use the relative values,
e.g., 'large', instead of absolute font sizes, e.g., 12. This
approach allows all text sizes to be made larger or smaller based
on the font manager's default font size.

This class accepts a single positional string as `fontconfig_ pattern_`,
or alternatively individual properties as keyword arguments::

```
FontProperties(pattern)
FontProperties(*, family=None, style=None, variant=None, ...)
```

This support does not depend on fontconfig; we are merely borrowing its
pattern syntax for use here.

.. _fontconfig: <https://www.freedesktop.org/wiki/Software/fontconfig/>
.. _pattern: <https://www.freedesktop.org/software/fontconfig/fontconfig-user.html>

Note that Matplotlib's internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in Matplotlib than in other applications that use fontconfig.

matplotlib.font_manager.afmFontProperty

```
afmFontProperty(fontpath, font)
```

Extract information from an AFM font file.

Parameters

fontpath : str

The filename corresponding to *font*.

font : AFM

The AFM font file from which information will be extracted.

Returns

`FontEntry`

The extracted font properties.

matplotlib.font_manager.findSystemFonts

```
findSystemFonts(fontpaths=None, fonttext='ttf')
```

Search for fonts in the specified font paths. If no paths are given, will use a standard set of system paths, as well as the list of fonts tracked by fontconfig if fontconfig is installed and available. A list of TrueType fonts are returned by default with AFM fonts as an option.

matplotlib.font_manager.generate_fontconfig_pattern

```
generate_fontconfig_pattern(d)
```

Convert a `.FontProperties` to a fontconfig pattern string.

matplotlib.font_manager.get_font

```
get_font(font_filepaths, hinting_factor=None)
```

Get an `.ft2font.FT2Font` object given a list of file paths.

Parameters

font_filepaths : Iterable[str, Path, bytes], str, Path, bytes

Relative or absolute paths to the font files to be used.

If a single string, bytes, or ``pathlib.Path``, then it will be treated as a list with that entry only.

If more than one filepath is passed, then the returned `FT2Font` object will fall back through the fonts, in the order given, to find a needed glyph.

Returns

``ft2font.FT2Font``

matplotlib.font_manager.get_fonttext_synonyms

```
get_fonttext_synonyms(fonttext)
```

Return a list of file extensions that are synonyms for the given file extension `*fileext*`.

matplotlib.font_manager.json_dump

```
json_dump(data, filename)
```

Dump ``FontManager`` `*data*` as JSON to the file named `*filename*`.

See Also

`json_load`

Notes

File paths that are children of the Matplotlib data path (typically, fonts shipped with Matplotlib) are stored relative to that data path (to remain valid across virtualenvs).

This function temporarily locks the output file to prevent multiple processes from overwriting one another's output.

matplotlib.font_manager.json_load

```
json_load(filename)
```

Load a ``FontManager`` from the JSON file named `*filename*`.

See Also

`json_dump`

matplotlib.font_manager.list_fonts

```
list_fonts(directory, extensions)
```

Return a list of all fonts matching any of the extensions, found recursively under the directory.

matplotlib.font_manager.ttfFontProperty

```
ttfFontProperty(font)
```

Extract information from a TrueType font file.

Parameters

font : ``FT2Font``

The TrueType font file from which information will be extracted.

Returns

``FontEntry``

The extracted font properties.

matplotlib.font_manager.win32FontDirectory

```
win32FontDirectory()
```

Return the user-specified font directory for Win32. This is looked up from the registry key ::

`\\HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\Shell Folders\\Fonts`

If the key is not found, ```%WINDIR%\\Fonts``` will be returned.

matplotlib.ft2font

```
ft2font(...)
```

No description available.

matplotlib.ft2font.FT2Font

```
FT2Font(...)
```

An object representing a single font face.

Outside of the font itself and querying its properties, this object provides methods for processing text strings into glyph shapes.

Commonly, one will use ``FT2Font.set_text`` to load some glyph metrics and outlines. Then ``FT2Font.draw_glyphs_to_bitmap`` and ``FT2Font.get_image`` may be used to get a rendered form of the loaded string.

For single characters, ``FT2Font.load_char`` or ``FT2Font.load_glyph`` may be used, either directly for their return values, or to use ``FT2Font.draw_glyph_to_bitmap`` or ``FT2Font.get_path``.

Useful metrics may be examined via the ``Glyph`` return values or ``FT2Font.get_kerning``. Most dimensions are given in 26.6 or 16.6 fixed-point integers representing subpixels. Divide these values by 64 to produce floating-point pixels.

matplotlib.ft2font.FT2Image

```
FT2Image(...)
```

An image buffer for drawing glyphs.

matplotlib.ft2font.FaceFlags

```
FaceFlags(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Flags returned by `FT2Font.face_flags`.

For more information, see `the FreeType documentation`

`<https://freetype.org/freetype2/docs/reference/ft2-face_creation.html#ft_face_flag_xxx>` `_`.

`.. versionadded:: 3.10`

matplotlib.ft2font.Glyph

```
Glyph(...)
```

Information about a single glyph.

You cannot create instances of this object yourself, but must use

`FT2Font.load_char` or `FT2Font.load_glyph` to generate one. This object may be used in a call to `FT2Font.draw_glyph_to_bitmap`.

For more information on the various metrics, see `the FreeType documentation`

`<https://freetype.org/freetype2/docs/glyphs/glyphs-3.html>` `_`.

matplotlib.ft2font.Kerning

```
Kerning(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Kerning modes for `FT2Font.get_kerning`.

For more information, see `the FreeType documentation`

`<https://freetype.org/freetype2/docs/reference/ft2-glyph_retrieval.html#ft_kerning_mode>` `_`.

`.. versionadded:: 3.10`

matplotlib.ft2font.LoadFlags

```
LoadFlags(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Flags for `FT2Font.load_char`, `FT2Font.load_glyph`, and `FT2Font.set_text`.

For more information, see `the FreeType documentation`

`<https://freetype.org/freetype2/docs/reference/ft2-glyph_retrieval.html#ft_load_xxx>` `_`.

`.. versionadded:: 3.10`

matplotlib.ft2font.StyleFlags

```
StyleFlags(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
```

Flags returned by `FT2Font.style_flags``.

For more information, see `the FreeType documentation`

`<https://freetype.org/freetype2/docs/reference/ft2-face_creation.html#ft_style_flag_xxx>`_`.`

.. versionadded:: 3.10

matplotlib.get_backend

```
get_backend(*, auto_select=True)
```

Return the name of the current backend.

Parameters

`auto_select` : bool, default: True

Whether to trigger backend resolution if no backend has been selected so far. If True, this ensures that a valid backend is returned. If False, this returns None if no backend has been selected so far.

.. versionadded:: 3.10

.. admonition:: Provisional

The `*auto_select*` flag is provisional. It may be changed or removed without prior warning.

See Also

`matplotlib.use`

matplotlib.get_cachedir

```
get_cachedir()
```

Return the string path of the cache directory.

The procedure used to find the directory is the same as for

``get_configdir``, except using ```$XDG_CACHE_HOME``/``$HOME/.cache``` instead.

matplotlib.get_configdir

```
get_configdir()
```

Return the string path of the configuration directory.

The directory is chosen as follows:

1. If the `MPLCONFIGDIR` environment variable is supplied, choose that.
2. On Linux, follow the XDG specification and look first in

``\$XDG_CONFIG_HOME``, if defined, or ``\$HOME/.config``. On other platforms, choose ``\$HOME/.matplotlib``.

3. If the chosen directory exists and is writable, use that as the configuration directory.
4. Else, create a temporary directory, and use it as the configuration directory.

matplotlib.get_data_path

```
get_data_path()
```

Return the path to Matplotlib data.

matplotlib.gridspec

```
gridspec(...)
```

:mod:`~matplotlib.gridspec` contains classes that help to layout multiple `~.axes.Axes` in a grid-like pattern within a figure.

The `GridSpec` specifies the overall grid structure. Individual cells within the grid are referenced by `SubplotSpec`'s.

Often, users need not access this module directly, and can use higher-level methods like `~.pyplot.subplots`, `~.pyplot.subplot_mosaic` and `~.Figure.subplots`. See the tutorial :ref:`arranging_axes` for a guide.

matplotlib.gridspec.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" ``[[xmin, ymin], [xmax, ymax]]``.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" ``(xmin, ymin, xmax, ymax)``

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" ``(xmin, ymin, width, height)``.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting ``ignore=True`` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify ``ignore`` explicitly. If not, the default value of ``ignore`` can be changed at any time by code with access to your Bbox, for example using the method `~.Bbox.ignore`.

****Properties of the ``null`` bbox****

.. note::

The current behavior of `Bbox.null()` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[inf, inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[inf, inf], [inf, inf]])
```

matplotlib.gridspec.GridSpec

```
GridSpec(nrows, ncols, figure=None, left=None, bottom=None, right=None, top=None,
wspace=None, hspace=None, width_ratios=None, height_ratios=None)
```

A grid layout to place subplots within a figure.

The location of the grid cells is determined in a similar way to `.SubplotParams`` using `*left*`, `*right*`, `*top*`, `*bottom*`, `*wspace*` and `*hspace*`.

Indexing a `GridSpec` instance returns a `.SubplotSpec``.

matplotlib.gridspec.GridSpecBase

```
GridSpecBase(nrows, ncols, height_ratios=None, width_ratios=None)
```

A base class of `GridSpec` that specifies the geometry of the grid that a subplot will be placed.

matplotlib.gridspec.GridSpecFromSubplotSpec

```
GridSpecFromSubplotSpec(nrows, ncols, subplot_spec, wspace=None, hspace=None,
height_ratios=None, width_ratios=None)
```

`GridSpec` whose subplot layout parameters are inherited from the location specified by a given `SubplotSpec`.

matplotlib.gridspec.SubplotParams

```
SubplotParams(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
```

Parameters defining the positioning of a subplots grid in a figure.

matplotlib.gridspec.SubplotSpec

```
SubplotSpec(gridspec, num1, num2=None)
```

The location of a subplot in a ``GridSpec``.

.. note::

Likely, you will never instantiate a ``SubplotSpec`` yourself. Instead, you will typically obtain one from a ``GridSpec`` using item-access.

Parameters

`gridspec` : `~matplotlib.gridspec.GridSpec``

The `GridSpec`, which the subplot is referencing.

`num1, num2` : int

The subplot will occupy the `*num1*-th` cell of the given `*gridspec*`. If `*num2*` is provided, the subplot will span between `*num1*-th` cell and `*num2*-th` cell ****inclusive****.

The index starts from 0.

matplotlib.hatch

```
hatch(...)
```

Contains classes for generating hatch patterns.

matplotlib.hatch.Circles

```
Circles(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.HatchPatternBase

```
HatchPatternBase()
```

The base class for a hatch pattern.

matplotlib.hatch.HorizontalHatch

```
HorizontalHatch(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.LargeCircles

```
LargeCircles(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.NorthEastHatch

```
NorthEastHatch(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- **vertices**: an (N, 2) float array of vertices
- **codes**: an N-length `numpy.uint8` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three `CURVE4` codes.

The code types are:

- `STOP` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- ``MOVETO`` : 1 vertex

Pick up the pen and move to the given vertex.

- ``LINETO`` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.hatch.Shapes

```
Shapes(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.SmallCircles

```
SmallCircles(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.SmallFilledCircles

```
SmallFilledCircles(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.SouthEastHatch

```
SouthEastHatch(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.Stars

```
Stars(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.VerticalHatch

```
VerticalHatch(hatch, density)
```

The base class for a hatch pattern.

matplotlib.hatch.get_path

```
get_path(hatchpattern, density=6)
```

Given a hatch specifier, *hatchpattern*, generates Path to render the hatch in a unit square. *density* is the number of lines per unit square.

matplotlib.image

```
image(...)
```

The image module supports basic image loading, rescaling and display operations.

matplotlib.image.Affine2D

```
Affine2D(matrix=None, **kwargs)
```

A mutable 2D affine transformation.

matplotlib.image.AxesImage

```
AxesImage(ax, *, cmap=None, norm=None, colorizer=None, interpolation=None,  
origin=None, extent=None, filternorm=True, filterrad=4.0, resample=False,  
interpolation_stage=None, **kwargs)
```

An image with pixels on a regular grid, attached to an Axes.

Parameters

ax : ~matplotlib.axes.Axes`

The Axes the image will belong to.

cmap : str or ~matplotlib.colors.Colormap`, default: :rc:`image.cmap`

The Colormap instance or registered colormap name used to map scalar data to colors.

norm : str or `~matplotlib.colors.Normalize``
 Maps luminance to 0-1.

interpolation : str, default: `:rc:`image.interpolation``
 Supported values are 'none', 'auto', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'.

interpolation_stage : {'data', 'rgba'}, default: 'data'
 If 'data', interpolation is carried out on the data provided by the user. If 'rgba', the interpolation is carried out after the colormapping has been applied (visual interpolation).

origin : {'upper', 'lower'}, default: `:rc:`image.origin``
 Place the [0, 0] index of the array in the upper left or lower left corner of the Axes. The convention 'upper' is typically used for matrices and images.

extent : tuple, optional
 The data axes (left, right, bottom, top) for making image plots registered with data plots. Default is to label the pixel centers with the zero-based row and column indices.

filternorm : bool, default: True
 A parameter for the antigrain image resize filter (see the antigrain documentation).
 If filternorm is set, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

filterrad : float > 0, default: 4
 The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

resample : bool, default: False
 When True, use a full resampling method. When False, only resample when the output image is larger than the input image.

****kwargs** : `~matplotlib.artist.Artist`` properties

matplotlib.image.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

```
-----
**Create from known bounds**
```

The default constructor takes the boundary "points" ```[[xmin, ymin], [xmax, ymax]]```.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" ```(xmin, ymin, xmax, ymax)```

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" ``(xmin, ymin, width, height)``.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting ``ignore=True`` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify ``ignore`` explicitly. If not, the default value of ``ignore`` can be changed at any time by code with access to your Bbox, for example using the method `~.Bbox.ignore`.

****Properties of the ``null`` bbox****

.. note::

The current behavior of `Bbox.null()` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[ -inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[ -inf, -inf], [inf, inf]])
```

matplotlib.image.BboxBase

```
BboxBase(shorthand_name=None)
```

The base class of all bounding boxes.

This class is immutable; `Bbox` is a mutable subclass.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

matplotlib.image.BboxImage

```
BboxImage(bbox, *, cmap=None, norm=None, colorizer=None, interpolation=None,
origin=None, filternorm=True, filterrad=4.0, resample=False, **kwargs)
```

The Image class whose size is determined by the given bbox.

matplotlib.image.BboxTransform

```
BboxTransform(boxin, boxout, **kwargs)
```

`BboxTransform` linearly transforms points from one `Bbox` to another.

matplotlib.image.BboxTransformTo

```
BboxTransformTo(boxout, **kwargs)
```

`BboxTransformTo` is a transformation that linearly transforms points from the unit bounding box to a given `Bbox`.

matplotlib.image.FigureCanvasBase

```
FigureCanvasBase(figure=None)
```

The canvas the figure renders into.

Attributes

figure : `~matplotlib.figure.Figure`
A high-level figure instance.

matplotlib.image.FigureImage

```
FigureImage(fig, *, cmap=None, norm=None, colorizer=None, offsetx=0, offsety=0,
            origin=None, **kwargs)
```

An image attached to a figure.

matplotlib.image.IdentityTransform

```
IdentityTransform(*args, **kwargs)
```

A special class that does one thing, the identity transform, in a fast way.

matplotlib.image.NonUniformImage

```
NonUniformImage(ax, *, interpolation='nearest', **kwargs)
```

An image with pixels on a rectilinear grid.

In contrast to `.AxesImage``, where pixels are on a regular grid, `NonUniformImage` allows rows and columns with individual heights / widths.

See also :doc:`/gallery/images_contours_and_fields/image_nonuniform`.

matplotlib.image.PcolorImage

```
PcolorImage(ax, x=None, y=None, A=None, *, cmap=None, norm=None, colorizer=None,
            **kwargs)
```

Make a pcolor-style plot with an irregular rectangular grid.

This uses a variation of the original irregular image code, and it is used by `pcolorfast` for the corresponding grid type.

matplotlib.image.TransformedBbox

```
TransformedBbox(bbox, transform, **kwargs)
```

A ``Bbox`` that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this `bbox` will update accordingly.

matplotlib.image.composite_images

```
composite_images(images, renderer, magnification=1.0)
```

Composite a number of RGBA images into one. The images are composited in the order in which they appear in the `*images*` list.

Parameters

`images` : list of Images

Each must have a ``make_image`` method. For each image, ``can_composite`` should return ``True``, though this is not enforced by this function. Each image must have a purely

affine transformation with no shear.

renderer : ``RendererBase``

magnification : float, default: 1

The additional magnification to apply for the renderer in use.

Returns

image : (M, N, 4) ``numpy.uint8`` array

The composited RGBA image.

offset_x, offset_y : float

The (left, bottom) offset where the composited image should be placed in the output figure.

matplotlib.image.imread

```
imread(fname, format=None)
```

Read an image from a file into an array.

.. note::

This function exists for historical reasons. It is recommended to use ``PIL.Image.open`` instead for loading images.

Parameters

fname : str or file-like

The image file to read: a filename, a URL or a file-like object opened in read-binary mode.

Passing a URL is deprecated. Please open the URL

for reading and pass the result to Pillow, e.g. with

``np.array(PIL.Image.open(urllib.request.urlopen(url)))``.

format : str, optional

The image file format assumed for reading the data. The image is loaded as a PNG file if `*format*` is set to "png", if `*fname*` is a path or opened file with a ".png" extension, or if it is a URL. In all other cases, `*format*` is ignored and the format is auto-detected by ``PIL.Image.open``.

Returns

``numpy.array``

The image data. The returned array has shape

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

PNG images are returned as float arrays (0-1). All other formats are returned as int arrays, with a bit depth determined by the file's contents.

matplotlib.image.imsave

```
imsave(fname, arr, vmin=None, vmax=None, cmap=None, format=None, origin=None, dpi=100,
*, metadata=None, pil_kwargs=None)
```

Colormap and save an array as an image file.

RGB(A) images are passed through. Single channel images will be colormapped according to `*cmap*` and `*norm*`.

.. note::

If you want to save a single channel image as gray scale please use an image I/O library (such as pillow, tiff file, or imageio) directly.

Parameters

fname : str or path-like or file-like

A path or a file-like object to store the image in.

If `*format*` is not set, then the output format is inferred from the extension of `*fname*`, if any, and from `:rc:`savefig.format`` otherwise.

If `*format*` is set, it determines the output format.

arr : array-like

The image data. Accepts NumPy arrays or sequences (e.g., lists or tuples). The shape can be one of MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).

vmin, vmax : float, optional

`*vmin*` and `*vmax*` set the color scaling for the image by fixing the values that map to the colormap color limits. If either `*vmin*` or `*vmax*` is None, that limit is determined from the `*arr*` min/max value.

cmap : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

A Colormap instance or registered colormap name. The colormap maps scalar data to colors. It is ignored for RGB(A) data.

format : str, optional

The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under `*fname*`.

origin : {'upper', 'lower'}, default: `:rc:`image.origin``

Indicates whether the `((0, 0))` index of the array is in the upper left or lower left corner of the Axes.

dpi : float

The DPI to store in the metadata of the file. This does not affect the resolution of the output image. Depending on file format, this may be rounded to the nearest integer.

metadata : dict, optional

Metadata in the image file. The supported keys depend on the output format, see the documentation of the respective backends for more information.

Currently only supported for "png", "pdf", "ps", "eps", and "svg".

pil_kwargs : dict, optional

Keyword arguments passed to ``PIL.Image.Image.save``. If the 'pnginfo' key is present, it completely overrides `*metadata*`, including the default 'Software' key.

matplotlib.image.pil_to_array

```
pil_to_array(pilImage)
```

Load a `PIL image`_ and return it as a numpy int array.

.. _PIL image: <https://pillow.readthedocs.io/en/latest/reference/Image.html>

Returns

numpy.array

The array shape depends on the image type:

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

matplotlib.image.thumbnail

```
thumbnail(infile, thumbfile, scale=0.1, interpolation='bilinear', preview=False)
```

Make a thumbnail of image in **infile** with output filename **thumbfile**.

See :doc:`/gallery/misc/image_thumbnail_sgskip`.

Parameters

infile : str or file-like

The image file. Matplotlib relies on Pillow_ for image reading, and thus supports a wide range of file formats, including PNG, JPG, TIFF and others.

.. _Pillow: <https://python-pillow.github.io>

thumbfile : str or file-like

The thumbnail filename.

scale : float, default: 0.1

The scale factor for the thumbnail.

interpolation : str, default: 'bilinear'

The interpolation scheme used in the resampling. See the **interpolation** parameter of `~.Axes.imshow`` for possible values.

preview : bool, default: False

If True, the default backend (presumably a user interface backend) will be used which will cause a figure to be raised if `~matplotlib.pyplot.show`` is called. If it is False, the figure is created using `.FigureCanvasBase`` and the drawing backend is selected as `.Figure.savefig`` would normally do.

Returns

`.Figure``

The figure instance containing the thumbnail.

matplotlib.inset

```
inset(...)
```

The inset module defines the `InsetIndicator` class, which draws the rectangle and connectors required for ``Axes.indicate_inset`` and ``Axes.indicate_inset_zoom``.

matplotlib.inset.ConnectionPatch

```
ConnectionPatch(xyA, xyB, coordsA, coordsB=None, *, axesA=None, axesB=None,
arrowstyle='-', connectionstyle='arc3', patchA=None, patchB=None, shrinkA=0.0,
shrinkB=0.0, mutation_scale=10.0, mutation_aspect=None, clip_on=False, **kwargs)
```

A patch that connects two points (possibly in different Axes).

matplotlib.inset.InsetIndicator

```
InsetIndicator(bounds=None, inset_ax=None, zorder=None, **kwargs)
```

An artist to highlight an area of interest.

An inset indicator is a rectangle on the plot at the position indicated by `*bounds*` that optionally has lines that connect the rectangle to an inset Axes (``Axes.inset_axes``).

.. versionadded:: 3.10

matplotlib.inset.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- `*vertices*`: an (N, 2) float array of vertices
- `*codes*`: an N-length ``numpy.uint8`` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three ``CURVE4`` codes.

The code types are:

- ``STOP`` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- ``MOVETO`` : 1 vertex

Pick up the pen and move to the given vertex.

- ``LINETO`` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint
Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint
Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)
Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.inset.PathPatch

```
PathPatch(path, **kwargs)
```

A general polycurve path patch.

matplotlib.inset.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point `*xy*` and its `*width*` and `*height*`.

The rectangle extends from ``xy[0]`` to ``xy[0] + width`` in x-direction and from ``xy[1]`` to ``xy[1] + height`` in y-direction. ::

```
: +-----+
: |
: | |
: height |
: | |
: (xy)--- width ----+
```

One may picture `*xy*` as the bottom left corner, but which corner `*xy*` is actually depends on the direction of the axis and the sign of `*width*` and `*height*`; e.g. `*xy*` would be the bottom right corner if the x-axis was inverted or if `*width*` was negative.

matplotlib.interactive

```
interactive(b)
```

Set whether to redraw after every plotting command (e.g. ``pyplot.xlabel``).

matplotlib.is_interactive

```
is_interactive()
```

Return whether to redraw after every plotting command.

.. note::

This function is only intended for use in backends. End users should use ``pyplot.isinteractive`` instead.

matplotlib.layout_engine

```
layout_engine(...)
```

Classes to layout elements in a ``Figure``.

Figures have a ``layout_engine`` property that holds a subclass of ``~.LayoutEngine`` defined here (or `*None*` for no layout). At draw time ``figure.get_layout_engine().execute()`` is called, the goal of which is usually to rearrange Axes on the figure to produce a pleasing layout. This is like a ``draw`` callback but with two differences. First, when printing we disable the layout engine for the final draw. Second, it is useful to know the layout engine while the figure is being created. In particular, colorbars are made differently with different layout engines (for historical reasons).

Matplotlib has two built-in layout engines:

- ``TightLayoutEngine`` was the first layout engine added to Matplotlib. See also :ref:`tight_layout_guide`.
- ``ConstrainedLayoutEngine`` is more modern and generally gives better results. See also :ref:`constrainedlayout_guide`.

Third parties can create their own layout engine by subclassing ``LayoutEngine``.

matplotlib.layout_engine.ConstrainedLayoutEngine

```
ConstrainedLayoutEngine(*, h_pad=None, w_pad=None, hspace=None, wspace=None, rect=(0, 0, 1, 1), compress=False, **kwargs)
```

Implements the ``constrained_layout`` geometry management. See :ref:`constrainedlayout_guide` for details.

matplotlib.layout_engine.LayoutEngine

```
LayoutEngine(**kwargs)
```

Base class for Matplotlib layout engines.

A layout engine can be passed to a figure at instantiation or at any time with `~.figure.Figure.set_layout_engine``. Once attached to a figure, the layout engine `execute`` function is called at draw time by `~.figure.Figure.draw``, providing a special draw-time hook.

.. note::

However, note that layout engines affect the creation of colorbars, so `~.figure.Figure.set_layout_engine`` should be called before any colorbars are created.

Currently, there are two properties of `LayoutEngine`` classes that are consulted while manipulating the figure:

- `engine.colorbar_gridspec`` tells `.Figure.colorbar`` whether to make the axes using the `gridspec` method (see `.colorbar.make_axes_gridspec``) or not (see `.colorbar.make_axes``);
- `engine.adjust_compatible`` stops `.Figure.subplots_adjust`` from being run if it is not compatible with the layout engine.

To implement a custom `LayoutEngine``:

1. override `adjust_compatible`` and `colorbar_gridspec``
2. override `LayoutEngine.set`` to update `*self._params*`
3. override `LayoutEngine.execute`` with your implementation

matplotlib.layout_engine.PlaceHolderLayoutEngine

```
PlaceHolderLayoutEngine(adjust_compatible, colorbar_gridspec, **kwargs)
```

This layout engine does not adjust the figure layout at all.

The purpose of this `.LayoutEngine`` is to act as a placeholder when the user removes a layout engine to ensure an incompatible `.LayoutEngine`` cannot be set later.

Parameters

`adjust_compatible, colorbar_gridspec` : bool

Allow the `PlaceHolderLayoutEngine` to mirror the behavior of whatever layout engine it is replacing.

matplotlib.layout_engine.TightLayoutEngine

```
TightLayoutEngine(*, pad=1.08, h_pad=None, w_pad=None, rect=(0, 0, 1, 1), **kwargs)
```

Implements the `tight_layout`` geometry management. See `:ref:`tight_layout_guide`` for details.

matplotlib.layout_engine.do_constrained_layout

```
do_constrained_layout(fig, h_pad, w_pad, hspace=None, wspace=None, rect=(0, 0, 1, 1), compress=False)
```

Do the `constrained_layout`. Called at draw time in `figure.constrained_layout()``

Parameters

`fig` : `~matplotlib.figure.Figure`
`.Figure`` instance to do the layout in.

`h_pad, w_pad` : float
Padding around the Axes elements in figure-normalized units.

`hspace, wspace` : float
Fraction of the figure to dedicate to space between the Axes. These are evenly spread between the gaps between the Axes. A value of 0.2 for a three-column layout would have a space of 0.1 of the figure width between each column. If `h/wspace < h/w_pad`, then the pads are used instead.

`rect` : tuple of 4 floats
Rectangle in figure coordinates to perform constrained layout in [left, bottom, width, height], each from 0-1.

`compress` : bool
Whether to shift Axes so that white space in between them is removed. This is useful for simple grids of fixed-aspect Axes (e.g. a grid of images).

Returns

`layoutgrid` : private debugging structure

matplotlib.layout_engine.get_subplotspec_list

```
get_subplotspec_list(axes_list, grid_spec=None)
```

Return a list of `subplotspec` from the given list of Axes.

For an instance of Axes that does not support `subplotspec`, `None` is inserted in the list.

If `grid_spec` is given, `None` is inserted for those not from the given `grid_spec`.

matplotlib.layout_engine.get_tight_layout_figure

```
get_tight_layout_figure(fig, axes_list, subplotspec_list, renderer, pad=1.08,  
h_pad=None, w_pad=None, rect=None)
```

Return subplot parameters for tight-laid-out-figure with specified padding.

Parameters

`fig` : `Figure`
`axes_list` : list of `Axes`
`subplotspec_list` : list of ``.SubplotSpec``
The `subplotspecs` of each Axes.
`renderer` : `renderer`

pad : float
 Padding between the figure edge and the edges of subplots, as a fraction of the font size.

h_pad, w_pad : float
 Padding (height/width) between edges of adjacent subplots. Defaults to *pad*.

rect : tuple (left, bottom, right, top), default: None.
 rectangle in normalized figure coordinates
 that the whole subplots area (including labels) will fit into.
 Defaults to using the entire figure.

Returns

 subplotspec or None
 subplotspec kwargs to be passed to `Figure.subplots_adjust` or
 None if tight_layout could not be accomplished.

matplotlib.legend

`legend(...)`

The legend module defines the Legend class, which is responsible for drawing legends associated with Axes and/or figures.

.. important::

It is unlikely that you would ever create a Legend instance manually. Most users would normally create a legend via the `~.Axes.legend` function. For more details on legends there is also a :ref:`legend guide <legend_guide>`.

The ``Legend`` class is a container of legend handles and legend texts.

The legend handler map specifies how to create legend handles from artists (lines, patches, etc.) in the Axes or figures. Default legend handlers are defined in the :mod:`~matplotlib.legend_handler` module. While not all artist types are covered by the default legend handlers, custom legend handlers can be defined to support arbitrary objects.

See the :ref:`<legend_guide>` for more information.

matplotlib.legend.AnchoredOffsetbox

`AnchoredOffsetbox(loc, *, pad=0.4, borderpad=0.5, child=None, prop=None, frameon=True, bbox_to_anchor=None, bbox_transform=None, **kwargs)`

An OffsetBox placed according to location *loc*.

AnchoredOffsetbox has a single child. When multiple children are needed, use an extra OffsetBox to enclose them. By default, the offset box is anchored against its parent Axes. You may explicitly specify the *bbox_to_anchor*.

matplotlib.legend.Artist

```
Artist()
```

Abstract base class for objects that render into a FigureCanvas.

Typically, all visible elements in a figure are subclasses of Artist.

matplotlib.legend.BarContainer

```
BarContainer(patches, errorbar=None, *, datavalues=None, orientation=None, **kwargs)
```

Container for the artists of bar plots (e.g. created by `.Axes.bar`).

The container can be treated as a tuple of the *patches* themselves. Additionally, you can access these and further parameters by the attributes.

Attributes

`patches` : list of :class:`~matplotlib.patches.Rectangle`
The artists of the bars.

`errorbar` : None or :class:`~matplotlib.container.ErrorbarContainer`
A container for the error bar artists if error bars are present.
None otherwise.

`datavalues` : None or array-like
The underlying data values corresponding to the bars.

`orientation` : {'vertical', 'horizontal'}, default: None
If 'vertical', the bars are assumed to be vertical.
If 'horizontal', the bars are assumed to be horizontal.

matplotlib.legend.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" `[[xmin, ymin], [xmax, ymax]]`.

```
>>> Bbox([[1, 1], [3, 7]])  
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" `(xmin, ymin, xmax, ymax)`

```
>>> Bbox.from_extents(1, 1, 3, 7)  
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" `(xmin, ymin, width, height)`.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting `ignore=True` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify `ignore` explicitly. If not, the default value of `ignore` can be changed at any time by code with access to your Bbox, for example using the method `~.Bbox.ignore`.

****Properties of the `null` bbox****

.. note::

The current behavior of `Bbox.null()` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other

set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])  
Bbox([[ -inf, -inf], [inf, inf]])
```

matplotlib.legend.BboxBase

```
BboxBase(shorthand_name=None)
```

The base class of all bounding boxes.

This class is immutable; `Bbox` is a mutable subclass.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

matplotlib.legend.BboxTransformFrom

```
BboxTransformFrom(boxin, **kwargs)
```

`BboxTransformFrom` linearly transforms points from a given `Bbox` to the unit bounding box.

matplotlib.legend.BboxTransformTo

```
BboxTransformTo(boxout, **kwargs)
```

`BboxTransformTo` is a transformation that linearly transforms points from the unit bounding box to a given `Bbox`.

matplotlib.legend.CircleCollection

```
CircleCollection(sizes, **kwargs)
```

A collection of circles, drawn using splines.

matplotlib.legend.Collection

```
Collection(*, edgecolors=None, facecolors=None, linewidths=None, linestyle='solid',  
capstyle=None, joinstyle=None, antialiaseds=None, offsets=None, offset_transform=None,  
norm=None, cmap=None, colorizer=None, pickradius=5.0, hatch=None, urls=None, zorder=1,  
**kwargs)
```

Base class for Collections. Must be subclassed to be usable.

A Collection represents a sequence of `Patch`s that can be drawn more efficiently together than individually. For example, when a single path is being drawn repeatedly at different offsets, the renderer can typically execute a `draw_marker()` call much more efficiently than a series of repeated calls to `draw_path()` with the offsets put in one-by-one.

Most properties of a collection can be configured per-element. Therefore,

Collections have "plural" versions of many of the properties of a `.Patch`` (e.g. `.Collection.get_paths`` instead of `.Patch.get_path``). Exceptions are the `*zorder*`, `*hatch*`, `*pickradius*`, `*capstyle*` and `*joinstyle*` properties, which can only be set globally for the whole collection.

Besides these exceptions, all properties can be specified as single values (applying to all elements) or sequences of values. The property of the `i`th` element of the collection is::

```
prop[i % len(prop)]
```

Each Collection can optionally be used as its own `.ScalarMappable`` by passing the `*norm*` and `*cmap*` parameters to its constructor. If the Collection's `.ScalarMappable`` matrix ```_A``` has been set (via a call to `.Collection.set_array``), then at draw time this internal scalar mappable will be used to set the ```facecolors``` and ```edgecolors```, ignoring those that were manually passed in.

matplotlib.legend.DraggableLegend

```
DraggableLegend(legend, use_blit=False, update='loc')
```

Helper base class for a draggable artist (legend, offsetbox).

Derived classes must override the following methods::

```
def save_offset(self):
    """
    Called when the object is picked for dragging; should save the
    reference position of the artist.
    """
```

```
def update_offset(self, dx, dy):
    """
    Called during the dragging; (*dx*, *dy*) is the pixel offset from
    the point where the mouse drag started.
    """
```

Optionally, you may override the following method::

```
def finalize_offset(self):
    """Called when the mouse is released."""
```

In the current implementation of `.DraggableLegend`` and `.DraggableAnnotation``, `.update_offset`` places the artists in display coordinates, and `.finalize_offset`` recalculates their position in axes coordinate and set a relevant attribute.

matplotlib.legend.DraggableOffsetBox

```
DraggableOffsetBox(ref_artist, offsetbox, use_blit=False)
```

Helper base class for a draggable artist (legend, offsetbox).

Derived classes must override the following methods::

```
def save_offset(self):
    """
    Called when the object is picked for dragging; should save the
    reference position of the artist.
    """
```

```
def update_offset(self, dx, dy):
    """
    Called during the dragging; (*dx*, *dy*) is the pixel offset from
    the point where the mouse drag started.
    """
```

Optionally, you may override the following method::

```
def finalize_offset(self):
    """Called when the mouse is released."""
```

In the current implementation of `.DraggableLegend`` and `.DraggableAnnotation``, `.update_offset`` places the artists in display coordinates, and `.finalize_offset`` recalculates their position in axes coordinate and set a relevant attribute.

matplotlib.legend.DrawingArea

```
DrawingArea(width, height, xdescent=0.0, ydescent=0.0, clip=False)
```

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed. The children can be clipped at the boundaries of the parent.

matplotlib.legend.ErrorbarContainer

```
ErrorbarContainer(lines, has_xerr=False, has_yerr=False, **kwargs)
```

Container for the artists of error bars (e.g. created by `.Axes.errorbar``).

The container can be treated as the `*lines*` tuple itself. Additionally, you can access these and further parameters by the attributes.

Attributes

lines : tuple

Tuple of ``(data_line, caplines, barlinecols)``.

- data_line : A `~matplotlib.lines.Line2D`` instance of x, y plot markers and/or line.
- caplines : A tuple of `~matplotlib.lines.Line2D`` instances of the error bar caps.
- barlinecols : A tuple of `~matplotlib.collections.LineCollection`` with the horizontal and vertical error ranges.

has_xerr, has_yerr : bool
``True`` if the errorbar has x/y errors.

matplotlib.legend.FancyBboxPatch

```
FancyBboxPatch(xy, width, height, boxstyle='round', *, mutation_scale=1,  
mutation_aspect=1, **kwargs)
```

A fancy box around a rectangle with lower left at `*xy* = (*x*, *y*)` with specified width and height.

`.FancyBboxPatch`` is similar to `.Rectangle``, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the style classes defined in `.BoxStyle``.

matplotlib.legend.FontProperties

```
FontProperties(family=None, style=None, variant=None, weight=None, stretch=None,  
size=None, fname=None, math_fontfamily=None)
```

A class for storing and manipulating font properties.

The font properties are the six properties described in the ``W3C Cascading Style Sheet, Level 1``
<<http://www.w3.org/TR/1998/REC-CSS2-19980512/>> ``_ font`` specification and `*math_fontfamily*` for math fonts:

- family: A list of font names in decreasing order of priority. The items may include a generic font family name, either 'sans-serif', 'serif', 'cursive', 'fantasy', or 'monospace'. In that case, the actual font to be used will be looked up from the associated rcParam during the search process in `.findfont``. Default: `:rc:`font.family``
- style: Either 'normal', 'italic' or 'oblique'. Default: `:rc:`font.style``
- variant: Either 'normal' or 'small-caps'. Default: `:rc:`font.variant``
- stretch: A numeric value in the range 0-1000 or one of 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded'. Default: `:rc:`font.stretch``
- weight: A numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'. Default: `:rc:`font.weight``
- size: Either a relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 10. Default: `:rc:`font.size``
- math_fontfamily: The family of fonts used to render math text.

Supported values are: 'dejavusans', 'dejavuserif', 'cm', 'stix', 'stixsans' and 'custom'. Default: :rc:`mathtext.fontset`

Alternatively, a font may be specified using the absolute path to a font file, by using the `*fname*` kwarg. However, in this case, it is typically simpler to just pass the path (as a ``pathlib.Path``, not a ``str``) to the `*font*` kwarg of the ``Text`` object.

The preferred usage of font sizes is to use the relative values, e.g., 'large', instead of absolute font sizes, e.g., 12. This approach allows all text sizes to be made larger or smaller based on the font manager's default font size.

This class accepts a single positional string as `fontconfig_pattern`, or alternatively individual properties as keyword arguments::

```
FontProperties(pattern)
FontProperties(*, family=None, style=None, variant=None, ...)
```

This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

```
.. _fontconfig: https://www.freedesktop.org/wiki/Software/fontconfig/
.. _pattern:
https://www.freedesktop.org/software/fontconfig/fontconfig-user.html
```

Note that Matplotlib's internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in Matplotlib than in other applications that use fontconfig.

matplotlib.legend.HPacker

```
HPacker(pad=0.0, sep=0.0, width=None, height=None, align='baseline', mode='fixed',
children=None)
```

HPacker packs its children horizontally, automatically adjusting their relative positions at draw time.

.. code-block:: none

```
+-----+
| Child 1 Child 2 Child 3 |
+-----+
```

matplotlib.legend.Legend

```
Legend(parent, handles, labels, *, loc=None, numpoints=None, markerscale=None,
markerfirst=True, reverse=False, scatterpoints=None, scatteryoffsets=None, prop=None,
fontsize=None, labelcolor=None, borderpad=None, labelspacing=None, handlelength=None,
handleheight=None, handletextpad=None, borderaxespad=None, columnspacing=None,
ncols=1, mode=None, fancybox=None, shadow=None, title=None, title_fontsize=None,
framealpha=None, edgecolor=None, facecolor=None, bbox_to_anchor=None,
bbox_transform=None, frameon=None, handler_map=None, title_fontproperties=None,
alignment='center', ncol=1, draggable=False)
```

Place a legend on the figure/axes.

matplotlib.legend.Line2D

```
Line2D(xdata, ydata, *, linewidth=None, linestyle=None, color=None, gapcolor=None,
marker=None, markersize=None, markeredgewidth=None, markeredgewidth=None,
markerfacecolor=None, markerfacecoloralt='none', fillstyle=None, antialiased=None,
dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None,
pickradius=5, drawstyle=None, markevery=None, **kwargs)
```

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create "stepped" lines in various styles.

matplotlib.legend.LineCollection

```
LineCollection(segments, *, zorder=2, **kwargs)
```

Represents a sequence of `.Line2D``s that should be drawn together.

This class extends `.Collection`` to represent a sequence of `.Line2D``s instead of just a sequence of `.Patch``s.

Just as in `.Collection``, each property of a `*LineCollection*` may be either a single value or a list of values. This list is then used cyclically for each element of the `LineCollection`, so the property of the ```i```th element of the collection is::

```
prop[i % len(prop)]
```

The properties of each member of a `*LineCollection*` default to their values in `:rc:`lines.*`` instead of `:rc:`patch.*``, and the property `*colors*` is added in place of `*edgecolors*`.

matplotlib.legend.Patch

```
Patch(*, edgecolor=None, facecolor=None, color=None, linewidth=None, linestyle=None,
antialiased=None, hatch=None, fill=True, capstyle=None, joinstyle=None, **kwargs)
```

A patch is a 2D artist with a face color and an edge color.

If any of `*edgecolor*`, `*facecolor*`, `*linewidth*`, or `*antialiased*` are `*None*`, they default to their rc params setting.

matplotlib.legend.PathCollection

```
PathCollection(paths, sizes=None, **kwargs)
```

A collection of `~.path.Path``s, as created by e.g. `~.Axes.scatter``.

matplotlib.legend.PolyCollection

```
PolyCollection(verts, sizes=None, *, closed=True, **kwargs)
```

Base class for collections that have an array of sizes.

matplotlib.legend.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point **xy** and its **width** and **height**.

The rectangle extends from ``xy[0]`` to ``xy[0] + width`` in x-direction and from ``xy[1]`` to ``xy[1] + height`` in y-direction. ::

```
: +-----+
: | |
: height |
: | |
: (xy)---- width ----+
```

One may picture **xy** as the bottom left corner, but which corner **xy** is actually depends on the direction of the axis and the sign of **width** and **height**; e.g. **xy** would be the bottom right corner if the x-axis was inverted or if **width** was negative.

matplotlib.legend.RegularPolyCollection

```
RegularPolyCollection(numsides, *, rotation=0, sizes=(1,), **kwargs)
```

A collection of n-sided regular polygons.

matplotlib.legend.Shadow

```
Shadow(patch, ox, oy, *, shade=0.7, **kwargs)
```

A patch is a 2D artist with a face color and an edge color.

If any of **edgecolor**, **facecolor**, **linewidth**, or **antialiased** are **None**, they default to their rc params setting.

matplotlib.legend.StemContainer

```
StemContainer(markerline_stemlines_baseline, **kwargs)
```

Container for the artists created in a :meth:`Axes.stem` plot.

The container can be treated like a namedtuple ``(markerline, stemlines, baseline)``.

Attributes

markerline : ~matplotlib.lines.Line2D`

The artist of the markers at the stem heads.

stemlines : ~matplotlib.collections.LineCollection`

The artists of the vertical lines for all stems.

baseline : ~matplotlib.lines.Line2D`

The artist of the horizontal baseline.

matplotlib.legend.StepPatch

```
StepPatch(values, edges, *, orientation='vertical', baseline=0, **kwargs)
```

A path patch describing a stepwise constant function.

By default, the path is not closed and starts and stops at baseline value.

matplotlib.legend.Text

```
Text(x=0, y=0, text='', *, color=None, verticalalignment='baseline',  
horizontalalignment='left', multialignment=None, fontproperties=None, rotation=None,  
linespacing=None, rotation_mode=None, usetex=None, wrap=False,  
transform_rotates_text=False, parse_math=None, antialiased=None, **kwargs)
```

Handle storing and drawing of text in window or data coordinates.

matplotlib.legend.TextArea

```
TextArea(s, *, textprops=None, multilinebaseline=False)
```

The TextArea is a container artist for a single Text instance.

The text is placed at (0, 0) with baseline+left alignment, by default. The width and height of the TextArea instance is the width and height of its child text.

matplotlib.legend.TransformedBbox

```
TransformedBbox(bbox, transform, **kwargs)
```

A `Bbox` that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.

matplotlib.legend.VPacker

```
VPacker(pad=0.0, sep=0.0, width=None, height=None, align='baseline', mode='fixed',  
children=None)
```

VPacker packs its children vertically, automatically adjusting their relative positions at draw time.

.. code-block:: none

```
+-----+  
| Child 1 |  
| Child 2 |  
| Child 3 |  
+-----+
```

matplotlib.legend.allow_rasterization

```
allow_rasterization(draw)
```

Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependent renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

matplotlib.legend.silent_list

```
silent_list(type, seq=None)
```

A list with a short `repr()`.

This is meant to be used for a homogeneous list of artists, so that they don't cause long, meaningless output.

Instead of ::

```
<matplotlib.lines.Line2D object at 0x7f5749fed3c8>,  
<matplotlib.lines.Line2D object at 0x7f5749fed4e0>,  
<matplotlib.lines.Line2D object at 0x7f5758016550>]
```

one will get ::

```
<a list of 3 Line2D objects>
```

If `self.type` is None, the type name is obtained from the first item in the list (if any).

matplotlib.legend_handler

```
legend_handler(...)
```

Default legend handlers.

.. important::

This is a low-level legend API, which most end users do not need.

We recommend that you are familiar with the :ref:`legend guide <legend_guide>` before reading this documentation.

Legend handlers are expected to be a callable object with a following signature::

```
legend_handler(legend, orig_handle, fontsize, handlebox)
```

Where `*legend*` is the legend itself, `*orig_handle*` is the original plot, `*fontsize*` is the fontsize in pixels, and `*handlebox*` is an `.OffsetBox` instance. Within the call, you should create relevant artists (using relevant properties from the `*legend*` and/or `*orig_handle*`) and add them into the `*handlebox*`. The artists need to be scaled according to the `*fontsize*` (note that the size is in pixels, i.e., this is dpi-scaled value).

This module includes definition of several legend handler classes

derived from the base class (HandlerBase) with the following method::

```
def legend_artist(self, legend, orig_handle, fontsize, handlebox)
```

matplotlib.legend_handler.HandlerBase

```
HandlerBase(xpad=0.0, ypad=0.0, update_func=None)
```

A base class for default legend handlers.

The derived classes are meant to override `*create_artists*` method, which has the following signature::

```
def create_artists(self, legend, orig_handle,
xdescent, ydescent, width, height, fontsize,
trans):
```

The overridden method needs to create artists of the given transform that fits in the given dimension (xdescent, ydescent, width, height) that are scaled by fontsize if necessary.

matplotlib.legend_handler.HandlerCircleCollection

```
HandlerCircleCollection(yoffsets=None, sizes=None, **kwargs)
```

Handler for ``CircleCollection``s.

matplotlib.legend_handler.HandlerErrorbar

```
HandlerErrorbar(xerr_size=0.5, yerr_size=None, marker_pad=0.3, numpoints=None,
**kwargs)
```

Handler for Errorbars.

matplotlib.legend_handler.HandlerLine2D

```
HandlerLine2D(marker_pad=0.3, numpoints=None, **kwargs)
```

Handler for ``Line2D`` instances.

See Also

HandlerLine2DCompound : An earlier handler implementation, which used one artist for the line and another for the marker(s).

matplotlib.legend_handler.HandlerLine2DCompound

```
HandlerLine2DCompound(marker_pad=0.3, numpoints=None, **kwargs)
```

Original handler for ``Line2D`` instances, that relies on combining a line-only with a marker-only artist. May be deprecated in the future.

matplotlib.legend_handler.HandlerLineCollection

```
HandlerLineCollection(marker_pad=0.3, numpoints=None, **kwargs)
```

Handler for ``LineCollection`` instances.

matplotlib.legend_handler.HandlerNpoints

```
HandlerNpoints(marker_pad=0.3, numpoints=None, **kwargs)
```

A legend handler that shows `*numpoints*` points in the legend entry.

matplotlib.legend_handler.HandlerNpointsYoffsets

```
HandlerNpointsYoffsets(numpoints=None, yoffsets=None, **kwargs)
```

A legend handler that shows `*numpoints*` in the legend, and allows them to be individually offset in the y-direction.

matplotlib.legend_handler.HandlerPatch

```
HandlerPatch(patch_func=None, **kwargs)
```

Handler for ``Patch`` instances.

matplotlib.legend_handler.HandlerPathCollection

```
HandlerPathCollection(yoffsets=None, sizes=None, **kwargs)
```

Handler for ``PathCollection``s, which are used by ``~.Axes.scatter``.

matplotlib.legend_handler.HandlerPolyCollection

```
HandlerPolyCollection(xpad=0.0, ypad=0.0, update_func=None)
```

Handler for ``PolyCollection`` used in ``~.Axes.fill_between`` and ``~.Axes.stackplot``.

matplotlib.legend_handler.HandlerRegularPolyCollection

```
HandlerRegularPolyCollection(yoffsets=None, sizes=None, **kwargs)
```

Handler for ``RegularPolyCollection``s.

matplotlib.legend_handler.HandlerStem

```
HandlerStem(marker_pad=0.3, numpoints=None, bottom=None, yoffsets=None, **kwargs)
```

Handler for plots produced by ``~.Axes.stem``.

matplotlib.legend_handler.HandlerStepPatch

```
HandlerStepPatch(xpad=0.0, ypad=0.0, update_func=None)
```

Handler for ``~.matplotlib.patches.StepPatch`` instances.

matplotlib.legend_handler.HandlerTuple

```
HandlerTuple(ndivide=1, pad=None, **kwargs)
```

Handler for Tuple.

matplotlib.legend_handler.Line2D

```
Line2D(xdata, ydata, *, linewidth=None, linestyle=None, color=None, gapcolor=None,
marker=None, markersize=None, markeredgewidth=None, markeredgewidth=None,
markerfacecolor=None, markerfacecoloralt='none', fillstyle=None, antialiased=None,
dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None,
pickradius=5, drawstyle=None, markevery=None, **kwargs)
```

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create "stepped" lines in various styles.

matplotlib.legend_handler.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point **xy** and its **width** and **height**.

The rectangle extends from `xy[0]` to `xy[0] + width` in x-direction and from `xy[1]` to `xy[1] + height` in y-direction. ::

```
: +-----+
: |
: height |
: |
: (xy)--- width ----+
```

One may picture **xy** as the bottom left corner, but which corner **xy** is actually depends on the direction of the axis and the sign of **width** and **height**; e.g. **xy** would be the bottom right corner if the x-axis was inverted or if **width** was negative.

matplotlib.legend_handler.update_from_first_child

```
update_from_first_child(tgt, src)
```

No description available.

matplotlib.lines

```
lines(...)
```

2D lines with support for a variety of line styles, markers, colors, etc.

matplotlib.lines.Artist

```
Artist()
```

Abstract base class for objects that render into a FigureCanvas.

Typically, all visible elements in a figure are subclasses of Artist.

matplotlib.lines.AxLine

```
AxLine(xy1, xy2, slope, **kwargs)
```

A helper class that implements `~.Axes.axline`, by recomputing the artist transform at draw time.

matplotlib.lines.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" `[[xmin, ymin], [xmax, ymax]]`.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" `(xmin, ymin, xmax, ymax)`

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" `(xmin, ymin, width, height)`.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting ``ignore=True`` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify ``ignore`` explicitly. If not, the default value of ``ignore`` can be changed at any time by code with access to your Bbox, for example using the method ``~.Bbox.ignore``.

****Properties of the ``null`` bbox****

.. note::

The current behavior of ``Bbox.null()`` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[-inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[-inf, -inf], [inf, inf]])
```

matplotlib.lines.BboxTransformTo

```
BboxTransformTo(boxout, **kwargs)
```

``BboxTransformTo`` is a transformation that linearly transforms points from the unit bounding box to a given ``Bbox``.

matplotlib.lines.CapStyle

```
CapStyle(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
```

Define how the two endpoints (caps) of an unclosed line are drawn.

How to draw the start and end points of lines that represent a closed curve (i.e. that end in a ``~.path.Path.CLOSEPOLY``) is controlled by the line's ``JoinStyle``. For all other lines, how the start and end points are drawn is controlled by the ``*CapStyle``.

For a visual impression of each `*CapStyle*`, view these docs online `<CapStyle>` or run ``CapStyle.demo``.

By default, ``~.backend_bases.GraphicsContextBase`` draws a stroked line as squared off at its endpoints.

****Supported values:****

.. rst-class:: value-list

'butt'

the line is squared off at its endpoint.

'projecting'

the line is squared off as in `*butt*`, but the filled in area extends beyond the endpoint a distance of ``linewidth/2``.

'round'

like `*butt*`, but a semicircular cap is added to the end of the line, of radius ``linewidth/2``.

.. plot::

:alt: Demo of possible CapStyle's

```
from matplotlib._enums import CapStyle
CapStyle.demo()
```

matplotlib.lines.JoinStyle

```
JoinStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the connection between two line segments is drawn.

For a visual impression of each `*JoinStyle*`, view these docs online `<JoinStyle>`, or run ``JoinStyle.demo``.

Lines in Matplotlib are typically defined by a 1D ``~.path.Path`` and a finite ``linewidth``, where the underlying 1D ``~.path.Path`` represents the center of the stroked line.

By default, ``~.backend_bases.GraphicsContextBase`` defines the boundaries of a stroked line to simply be every point within some radius, ``linewidth/2``, away from any point of the center line. However, this results in corners appearing "rounded", which may not be the desired behavior if you are drawing, for example, a polygon or pointed star.

****Supported values:****

.. rst-class:: value-list

'miter'

the "arrow-tip" style. Each boundary of the filled-in area will extend in a straight line parallel to the tangent vector of the centerline at the point it meets the corner, until they meet in a sharp point.

'round'
stokes every point within a radius of ``linewidth/2`` of the center lines.

'bevel'
the "squared-off" style. It can be thought of as a rounded corner where the "circular" part of the corner has been cut off.

.. note::

Very long miter tips are cut off (to form a *bevel*) after a backend-dependent limit called the "miter limit", which specifies the maximum allowed ratio of miter length to line width. For example, the PDF backend uses the default value of 10 specified by the PDF standard, while the SVG backend does not even specify the miter limit, resulting in a default value of 4 per the SVG specification. Matplotlib does not currently allow the user to adjust this parameter.

A more detailed description of the effect of a miter limit can be found in the `Mozilla Developer Docs`_ <<https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-miterlimit>>`_

.. plot::
:alt: Demo of possible JoinStyle's

```
from matplotlib._enums import JoinStyle
JoinStyle.demo()
```

matplotlib.lines.Line2D

```
Line2D(xdata, ydata, *, linewidth=None, linestyle=None, color=None, gapcolor=None,
marker=None, markersize=None, markeredgewidth=None, markeredgewidth=None,
markerfacecolor=None, markerfacecoloralt='none', fillstyle=None, antialiased=None,
dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None,
pickradius=5, drawstyle=None, markevery=None, **kwargs)
```

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create "stepped" lines in various styles.

matplotlib.lines.MarkerStyle

```
MarkerStyle(marker, fillstyle=None, transform=None, capstyle=None, joinstyle=None)
```

A class representing marker types.

Instances are immutable. If you need to change anything, create a new instance.

Attributes

markers : dict

All known markers.

filled_markers : tuple

All known filled markers. This is a subset of *markers*.

fillstyles : tuple
The supported fillstyles.

matplotlib.lines.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- **vertices**: an (N, 2) float array of vertices
- **codes**: an N-length ``numpy.uint8`` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three ``CURVE4`` codes.

The code types are:

- ``STOP`` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- ``MOVETO`` : 1 vertex

Pick up the pen and move to the given vertex.

- ``LINETO`` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If **codes** is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of **codes** being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'.

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.lines.TransformPath

```
TransformPath(path, transform)
```

A `TransformPath` caches a non-affine transformed copy of the `~.path.Path`. This cached copy is automatically updated when the non-affine part of the transform changes.

.. note::

Paths are considered immutable by this class. Any update to the path's vertices/codes will not trigger a transform recomputation.

matplotlib.lines.VertexSelector

```
VertexSelector(line)
```

Manage the callbacks to maintain a list of selected vertices for `.Line2D`. Derived classes should override the `process_selected` method to do something with the picks.

Here is an example which highlights the selected verts with red circles::

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines

class HighlightSelected(lines.VertexSelector):
    def __init__(self, line, fmt='ro', **kwargs):
        super().__init__(line)
        self.markers, = self.axes.plot([], [], fmt, **kwargs)

    def process_selected(self, ind, xs, ys):
        self.markers.set_data(xs, ys)
        self.canvas.draw()

fig, ax = plt.subplots()
x, y = np.random.rand(2, 30)
line, = ax.plot(x, y, 'bs-', picker=5)

selector = HighlightSelected(line)
plt.show()
```

matplotlib.lines.allow_rasterization

```
allow_rasterization(draw)
```

Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependent renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

matplotlib.lines.segment_hits

```
segment_hits(cx, cy, x, y, radius)
```

Return the indices of the segments in the polyline with coordinates (*cx*, *cy*) that are within a distance *radius* of the point (*x*, *y*).

matplotlib.markers

```
markers(...)
```

Functions to handle markers; used by the marker functionality of `~matplotlib.axes.Axes.plot`, `~matplotlib.axes.Axes.scatter`, and `~matplotlib.axes.Axes.errorbar`.

All possible markers are defined here:

```
=====
marker symbol description
=====
'''.'''' |m00| point
','''' |m01| pixel
'''o''' |m02| circle
'''v''' |m03| triangle_down
'''^''' |m04| triangle_up
'''<''' |m05| triangle_left
'''>''' |m06| triangle_right
'''1''' |m07| tri_down
'''2''' |m08| tri_up
'''3''' |m09| tri_left
'''4''' |m10| tri_right
'''8''' |m11| octagon
'''s''' |m12| square
'''p''' |m13| pentagon
'''P''' |m23| plus (filled)
'''*''' |m14| star
'''h''' |m15| hexagon1
'''H''' |m16| hexagon2
'''+'''' |m17| plus
'''x''' |m18| x
'''X''' |m24| x (filled)
'''D''' |m19| diamond
'''d''' |m20| thin_diamond
'''|''' |m21| vline
'''-'''' |m22| hline
'''0''' ('`TICKLEFT`) |m25| tickleft
'''1''' ('`TICKRIGHT`) |m26| tickright
'''2''' ('`TICKUP`) |m27| tickup
'''3''' ('`TICKDOWN`) |m28| tickdown
```

```

``4`` (``CARETLEFT``) |m29| caretleft
``5`` (``CARETRIGHT``) |m30| caretright
``6`` (``CARETUP``) |m31| caretup
``7`` (``CARETDOWN``) |m32| caretdown
``8`` (``CARETLEFTBASE``) |m33| caretleft (centered at base)
``9`` (``CARETRIGHTBASE``) |m34| caretright (centered at base)
``10`` (``CARETUPBASE``) |m35| caretup (centered at base)
``11`` (``CARETDOWNBASE``) |m36| caretdown (centered at base)
``"none"`` or ``None`` nothing
``" "`` or ``""`` nothing
``"$...$"`` |m37| Render the string using mathtext.
E.g. ``"$f$"`` for marker showing the
letter ``f``.
``verts`` A list of (x, y) pairs used for Path
vertices. The center of the marker is
located at (0, 0) and the size is
normalized, such that the created path
is encapsulated inside the unit cell.
``path`` A ~matplotlib.path.Path instance.
``(numsides, 0, angle)`` A regular polygon with ``numsides``
sides, rotated by ``angle``.
``(numsides, 1, angle)`` A star-like symbol with ``numsides``
sides, rotated by ``angle``.
``(numsides, 2, angle)`` An asterisk with ``numsides`` sides,
rotated by ``angle``.
=====

```

Note that special symbols can be defined via the
:ref: STIX math font `<mathtext>`,
e.g. ```"$\u266B$"```. For an overview over the STIX font symbols refer to the
`STIX font table` <http://www.stixfonts.org/allGlyphs.html> `>`_``.
Also see the :doc:`/gallery/text_labels_and_annotations/stix_fonts_demo`.

Integer numbers from ```0``` to ```11``` create lines and triangles. Those are
equally accessible via capitalized variables, like ```CARETDOWNBASE```.
Hence the following are equivalent::

```

plt.plot([1, 2, 3], marker=11)
plt.plot([1, 2, 3], marker=matplotlib.markers.CARETDOWNBASE)

```

Markers join and cap styles can be customized by creating a new instance of
`MarkerStyle`.

A `MarkerStyle` can also have a custom `~matplotlib.transforms.Transform``
allowing it to be arbitrarily rotated or offset.

Examples showing the use of markers:

```

* :doc:`/gallery/lines_bars_and_markers/marker_reference`
* :doc:`/gallery/lines_bars_and_markers/scatter_star_poly`
* :doc:`/gallery/lines_bars_and_markers/multivariate_marker_plot`

```

```

.. |m00| image:: /_static/markers/m00.png
.. |m01| image:: /_static/markers/m01.png
.. |m02| image:: /_static/markers/m02.png
.. |m03| image:: /_static/markers/m03.png

```

```
.. |m04| image:: /_static/markers/m04.png
.. |m05| image:: /_static/markers/m05.png
.. |m06| image:: /_static/markers/m06.png
.. |m07| image:: /_static/markers/m07.png
.. |m08| image:: /_static/markers/m08.png
.. |m09| image:: /_static/markers/m09.png
.. |m10| image:: /_static/markers/m10.png
.. |m11| image:: /_static/markers/m11.png
.. |m12| image:: /_static/markers/m12.png
.. |m13| image:: /_static/markers/m13.png
.. |m14| image:: /_static/markers/m14.png
.. |m15| image:: /_static/markers/m15.png
.. |m16| image:: /_static/markers/m16.png
.. |m17| image:: /_static/markers/m17.png
.. |m18| image:: /_static/markers/m18.png
.. |m19| image:: /_static/markers/m19.png
.. |m20| image:: /_static/markers/m20.png
.. |m21| image:: /_static/markers/m21.png
.. |m22| image:: /_static/markers/m22.png
.. |m23| image:: /_static/markers/m23.png
.. |m24| image:: /_static/markers/m24.png
.. |m25| image:: /_static/markers/m25.png
.. |m26| image:: /_static/markers/m26.png
.. |m27| image:: /_static/markers/m27.png
.. |m28| image:: /_static/markers/m28.png
.. |m29| image:: /_static/markers/m29.png
.. |m30| image:: /_static/markers/m30.png
.. |m31| image:: /_static/markers/m31.png
.. |m32| image:: /_static/markers/m32.png
.. |m33| image:: /_static/markers/m33.png
.. |m34| image:: /_static/markers/m34.png
.. |m35| image:: /_static/markers/m35.png
.. |m36| image:: /_static/markers/m36.png
.. |m37| image:: /_static/markers/m37.png
```

matplotlib.markers.Affine2D

```
Affine2D(matrix=None, **kwargs)
```

A mutable 2D affine transformation.

matplotlib.markers.CapStyle

```
CapStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the two endpoints (caps) of an unclosed line are drawn.

How to draw the start and end points of lines that represent a closed curve (i.e. that end in a `~.path.Path.CLOSEPOLY``) is controlled by the line's ``JoinStyle``. For all other lines, how the start and end points are drawn is controlled by the `*CapStyle*`.

For a visual impression of each `*CapStyle*`, ``view these docs online <CapStyle>`` or run ``CapStyle.demo``.

By default, `~.backend_bases.GraphicsContextBase`` draws a stroked line as squared off at its endpoints.

****Supported values:****

.. rst-class:: value-list

'butt'

the line is squared off at its endpoint.

'projecting'

the line is squared off as in **butt**, but the filled in area extends beyond the endpoint a distance of ```linewidth/2```.

'round'

like **butt**, but a semicircular cap is added to the end of the line, of radius ```linewidth/2```.

.. plot::

:alt: Demo of possible CapStyle's

```
from matplotlib._enums import CapStyle
CapStyle.demo()
```

matplotlib.markers.IdentityTransform

```
IdentityTransform(*args, **kwargs)
```

A special class that does one thing, the identity transform, in a fast way.

matplotlib.markers.JoinStyle

```
JoinStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the connection between two line segments is drawn.

For a visual impression of each **JoinStyle**, `view these docs online <JoinStyle>`, or run `JoinStyle.demo``.

Lines in Matplotlib are typically defined by a 1D `~.path.Path`` and a finite ```linewidth```, where the underlying 1D `~.path.Path`` represents the center of the stroked line.

By default, `~.backend_bases.GraphicsContextBase`` defines the boundaries of a stroked line to simply be every point within some radius, ```linewidth/2```, away from any point of the center line. However, this results in corners appearing "rounded", which may not be the desired behavior if you are drawing, for example, a polygon or pointed star.

****Supported values:****

.. rst-class:: value-list

'miter'

the "arrow-tip" style. Each boundary of the filled-in area will extend in a straight line parallel to the tangent vector of the centerline at the point it meets the corner, until they meet in a sharp point.

'round'

stokes every point within a radius of ``linewidth/2`` of the center lines.

'bevel'

the "squared-off" style. It can be thought of as a rounded corner where the "circular" part of the corner has been cut off.

.. note::

Very long miter tips are cut off (to form a *bevel*) after a backend-dependent limit called the "miter limit", which specifies the maximum allowed ratio of miter length to line width. For example, the PDF backend uses the default value of 10 specified by the PDF standard, while the SVG backend does not even specify the miter limit, resulting in a default value of 4 per the SVG specification. Matplotlib does not currently allow the user to adjust this parameter.

A more detailed description of the effect of a miter limit can be found in the `Mozilla Developer Docs`_ <<https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-miterlimit>>`_

.. plot::

:alt: Demo of possible JoinStyle's

```
from matplotlib._enums import JoinStyle
JoinStyle.demo()
```

matplotlib.markers.MarkerStyle

```
MarkerStyle(marker, fillstyle=None, transform=None, capstyle=None, joinstyle=None)
```

A class representing marker types.

Instances are immutable. If you need to change anything, create a new instance.

Attributes

markers : dict

All known markers.

filled_markers : tuple

All known filled markers. This is a subset of *markers*.

fillstyles : tuple

The supported fillstyles.

matplotlib.markers.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- `*vertices*`: an (N, 2) float array of vertices
- `*codes*`: an N-length ``numpy.uint8`` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three ``CURVE4`` codes.

The code types are:

- ``STOP`` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- ``MOVETO`` : 1 vertex

Pick up the pen and move to the given vertex.

- ``LINETO`` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'.

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.mathtext

```
mathtext(...)
```

A module for parsing a subset of the TeX math syntax and rendering it to a Matplotlib backend.

For a tutorial of its usage, see :ref:`mathtext`. This document is primarily concerned with implementation details.

The module uses `pyparsing_` to parse the TeX expression.

.. _pyparsing: <https://pypi.org/project/pyparsing/>

The Bakoma distribution of the TeX Computer Modern fonts, and STIX fonts are supported. There is experimental support for using arbitrary fonts, but results may vary without proper tweaking and metrics for those fonts.

matplotlib.mathtext.FontProperties

```
FontProperties(family=None, style=None, variant=None, weight=None, stretch=None, size=None, fname=None, math_fontfamily=None)
```

A class for storing and manipulating font properties.

The font properties are the six properties described in the `W3C Cascading Style Sheet, Level 1` <<http://www.w3.org/TR/1998/REC-CSS2-19980512/>>`_ font specification and `*math_fontfamily*` for math fonts:

- family: A list of font names in decreasing order of priority. The items may include a generic font family name, either 'sans-serif', 'serif', 'cursive', 'fantasy', or 'monospace'. In that case, the actual font to be used will be looked up from the associated rcParam during the search process in `.findfont`. Default: `:rc:`font.family``
- style: Either 'normal', 'italic' or 'oblique'. Default: `:rc:`font.style``
- variant: Either 'normal' or 'small-caps'. Default: `:rc:`font.variant``
- stretch: A numeric value in the range 0-1000 or one of 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded'. Default: `:rc:`font.stretch``
- weight: A numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'. Default: `:rc:`font.weight``
- size: Either a relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 10. Default: `:rc:`font.size``
- math_fontfamily: The family of fonts used to render math text.

Supported values are: 'dejavusans', 'dejavuserif', 'cm', 'stix', 'stixsans' and 'custom'. Default: :rc:`mathtext.fontset`

Alternatively, a font may be specified using the absolute path to a font file, by using the `*fname*` kwarg. However, in this case, it is typically simpler to just pass the path (as a ``pathlib.Path``, not a ``str``) to the `*font*` kwarg of the ``Text`` object.

The preferred usage of font sizes is to use the relative values, e.g., 'large', instead of absolute font sizes, e.g., 12. This approach allows all text sizes to be made larger or smaller based on the font manager's default font size.

This class accepts a single positional string as `fontconfig_pattern`, or alternatively individual properties as keyword arguments::

```
FontProperties(pattern)
FontProperties(*, family=None, style=None, variant=None, ...)
```

This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

```
.. _fontconfig: https://www.freedesktop.org/wiki/Software/fontconfig/
.. _pattern:
https://www.freedesktop.org/software/fontconfig/fontconfig-user.html
```

Note that Matplotlib's internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in Matplotlib than in other applications that use fontconfig.

matplotlib.mathtext.LoadFlags

```
LoadFlags(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Flags for ``FT2Font.load_char``, ``FT2Font.load_glyph``, and ``FT2Font.set_text``.

For more information, see `the FreeType documentation`
<https://freetype.org/freetype2/docs/reference/ft2-glyph_retrieval.html#ft_load_xxx>`.

.. versionadded:: 3.10

matplotlib.mathtext.MathTextParser

```
MathTextParser(output)
```

No description available.

matplotlib.mathtext.RasterParse

```
RasterParse(ox: ForwardRef('float'), oy: ForwardRef('float'), width:
ForwardRef('float'), height: ForwardRef('float'), depth: ForwardRef('float'), image:
ForwardRef('FT2Image'))
```

The namedtuple type returned by `MathTextParser("agg").parse(...)`.

Attributes

ox, oy : float

The offsets are always zero.

width, height, depth : float

The global metrics.

image : FT2Image

A raster image.

matplotlib.mathtext.VectorParse

```
VectorParse(width: ForwardRef('float'), height: ForwardRef('float'), depth:
ForwardRef('float'), glyphs: ForwardRef('list[tuple[FT2Font, float, int, float,
float]]'), rects: ForwardRef('list[tuple[float, float, float, float]]'))
```

The namedtuple type returned by `MathTextParser("path").parse(...)`.

Attributes

width, height, depth : float

The global metrics.

glyphs : list

The glyphs including their positions.

rect : list

The list of rectangles.

matplotlib.mathtext.get_unicode_index

```
get_unicode_index(symbol: 'str') -> 'int'
```

Return the integer index (from the Unicode table) of *symbol*.

Parameters

symbol : str

A single (Unicode) character, a TeX command (e.g. `r'\pi'`) or a Type1 symbol name (e.g. `'phi'`).

matplotlib.mathtext.math_to_image

```
math_to_image(s, filename_or_obj, prop=None, dpi=None, format=None, *, color=None)
```

Given a math expression, renders it in a closely-clipped bounding box to an image file.

Parameters

s : str

A math expression. The math portion must be enclosed in dollar signs.

filename_or_obj : str or path-like or file-like

Where to write the image data.

prop : `FontProperties`, optional

The size and style of the text.

dpi : float, optional

The output dpi. If not set, the dpi is determined as for
`.Figure.savefig`.
format : str, optional
The output format, e.g., 'svg', 'pdf', 'ps' or 'png'. If not set, the
format is determined as for `.Figure.savefig`.
color : str, optional
Foreground color, defaults to :rc:`text.color`.

matplotlib.matplotlib_fname

```
matplotlib_fname()
```

Get the location of the config file.

The file location is determined in the following order

- ``\$PWD/matplotlibrc``
- ``\$MATPLOTLIBRC`` if it is not a directory
- ``\$MATPLOTLIBRC/matplotlibrc``
- ``\$MPLCONFIGDIR/matplotlibrc``
- On Linux,
 - ``\$XDG_CONFIG_HOME/matplotlib/matplotlibrc`` (if ``\$XDG_CONFIG_HOME`` is defined)
 - or ``\$HOME/.config/matplotlib/matplotlibrc`` (if ``\$XDG_CONFIG_HOME`` is not defined)
- On other platforms,
 - ``\$HOME/.matplotlib/matplotlibrc`` if ``\$HOME`` is defined
- Lastly, it looks in ``\$MATPLOTLIBDATA/matplotlibrc``, which should always exist.

matplotlib.mlab

```
mlab(...)
```

Numerical Python functions written for compatibility with MATLAB commands with the same names. Most numerical Python functions can be found in the `NumPy`_ and `SciPy`_ libraries. What remains here is code for performing spectral computations and kernel density estimations.

.. _NumPy: <https://numpy.org>
.. _SciPy: <https://www.scipy.org>

Spectral functions

`cohere`
Coherence (normalized cross spectral density)

`csd`
Cross spectral density using Welch's average periodogram

`detrend`
Remove the mean or best fit line from an array

``psd``

Power spectral density using Welch's average periodogram

``specgram``

Spectrogram (spectrum over segments of time)

``complex_spectrum``

Return the complex-valued frequency spectrum of a signal

``magnitude_spectrum``

Return the magnitude of the frequency spectrum of a signal

``angle_spectrum``

Return the angle (wrapped phase) of the frequency spectrum of a signal

``phase_spectrum``

Return the phase (unwrapped angle) of the frequency spectrum of a signal

``detrend_mean``

Remove the mean from a line.

``detrend_linear``

Remove the best fit line from a line.

``detrend_none``

Return the original line.

matplotlib.mlab.GaussianKDE

```
GaussianKDE(dataset, bw_method=None)
```

Representation of a kernel-density estimate using Gaussian kernels.

Parameters

`dataset` : array-like

Datapoints to estimate from. In case of univariate data this is a 1-D array, otherwise a 2D array with shape (# of dims, # of data).

`bw_method` : {'scott', 'silverman'} or float or callable, optional

The method used to calculate the estimator bandwidth. If a float, this will be used directly as ``kde.factor``. If a callable, it should take a ``GaussianKDE`` instance as only parameter and return a float. If None (default), 'scott' is used.

Attributes

`dataset` : ndarray

The dataset passed to the constructor.

`dim` : int

Number of dimensions.

`num_dp` : int

Number of datapoints.

`factor` : float

The bandwidth factor, obtained from ``kde.covariance_factor``, with which

the covariance matrix is multiplied.

covariance : ndarray

The covariance matrix of *dataset*, scaled by the calculated bandwidth (`kde.factor`).

inv_cov : ndarray

The inverse of *covariance*.

Methods

kde.evaluate(points) : ndarray

Evaluate the estimated pdf on a provided set of points.

kde(points) : ndarray

Same as kde.evaluate(points)

matplotlib.mlab.cohere

```
cohere(x, y, NFFT=256, Fs=2, detrend=, window=, noverlap=0, pad_to=None,
sides='default', scale_by_freq=None)
```

The coherence between *x* and *y*. Coherence is the normalized cross spectral density:

.. math::

$$C_{\{xy\}} = \frac{|P_{\{xy\}}|^2}{P_{\{xx\}}P_{\{yy\}}}$$

Parameters

x, y

Array or sequence containing the data

Fs : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window : callable or ndarray, default: ``.window_hanning``

A function or a vector of length *NFFT*. To create window vectors see ``.window_hanning``, ``.window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to : int, optional

The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `~numpy.fft.fft`. The default is None, which sets *pad_to* equal to *NFFT*

NFFT : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

detrend : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The :mod:`~matplotlib.mlab` module defines `.detrend_none`, `.detrend_mean`, and `.detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `.detrend_none`. 'mean' calls `.detrend_mean`. 'linear' calls `.detrend_linear`.

scale_by_freq : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap : int, default: 0 (no overlap)

The number of points of overlap between segments.

Returns

Cxy : 1-D array

The coherence vector.

freqs : 1-D array

The frequencies for the elements in *Cxy*.

See Also

:func:`psd`, :func:`csd` :

For information about the methods used to compute :math:`P_{xy}`,

:math:`P_{xx}` and :math:`P_{yy}`.

matplotlib.mlab.csd

```
csd(x, y, NFFT=None, Fs=None, detrend=None, window=None, noverlap=None, pad_to=None,
    sides=None, scale_by_freq=None)
```

Compute the cross-spectral density.

The cross spectral density :math:`P_{xy}` by Welch's average periodogram method. The vectors *x* and *y* are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of *x* and *y* are averaged over each segment to compute :math:`P_{xy}` , with a scaling to correct for power loss due to windowing.

If $\text{len}(x) < \text{NFFT}$ or $\text{len}(y) < \text{NFFT}$, they will be zero padded to *NFFT*.

Parameters

`x, y` : 1-D arrays or sequences

Arrays or sequences containing the data

`Fs` : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

`window` : callable or ndarray, default: ``.window_hanning``

A function or a vector of length `*NFFT*`. To create window vectors see ``.window_hanning``, ``.window_none``, ``.numpy.blackman``, ``.numpy.hamming``, ``.numpy.bartlett``, ``.scipy.signal``, ``.scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. This can be different from `*NFFT*`, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to ``.numpy.fft.fft``. The default is None, which sets `*pad_to*` equal to `*NFFT*`

`NFFT` : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use `*pad_to*` for this instead.

`detrend` : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the `*detrend*` parameter is a vector, in Matplotlib it is a function. The `:mod:`~matplotlib.mlab`` module defines ``.detrend_none``, ``.detrend_mean``, and ``.detrend_linear``, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls ``.detrend_none``. 'mean' calls ``.detrend_mean``. 'linear' calls ``.detrend_linear``.

`scale_by_freq` : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

`noverlap` : int, default: 0 (no overlap)

The number of points of overlap between segments.

Returns

Pxy : 1-D array

The values for the cross spectrum :math:`P_{xy}` before scaling (real valued)

freqs : 1-D array

The frequencies corresponding to the elements in *Pxy*

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

See Also

psd : equivalent to setting ``y = x``.

matplotlib.mlab.detrend

```
detrend(x, key=None, axis=None)
```

Return *x* with its trend removed.

Parameters

x : array or sequence

Array or sequence containing the data.

key : {'default', 'constant', 'mean', 'linear', 'none'} or function

The detrending algorithm to use. 'default', 'mean', and 'constant' are the same as `detrend_mean`. 'linear' is the same as `detrend_linear`. 'none' is the same as `detrend_none`. The default is 'mean'. See the corresponding functions for more details regarding the algorithms. Can also be a function that carries out the detrend operation.

axis : int

The axis along which to do the detrending.

See Also

`detrend_mean` : Implementation of the 'mean' algorithm.

`detrend_linear` : Implementation of the 'linear' algorithm.

`detrend_none` : Implementation of the 'none' algorithm.

matplotlib.mlab.detrend_linear

```
detrend_linear(y)
```

Return *x* minus best fit line; 'linear' detrending.

Parameters

y : 0-D or 1-D array or sequence

Array or sequence containing the data

See Also

detrend_mean : Another detrend algorithm.

detrend_none : Another detrend algorithm.

detrend : A wrapper around all the detrend algorithms.

matplotlib.mlab.detrend_mean

```
detrend_mean(x, axis=None)
```

Return x minus the mean(x).

Parameters

x : array or sequence

Array or sequence containing the data

Can have any dimensionality

axis : int

The axis along which to take the mean. See `numpy.mean` for a description of this argument.

See Also

detrend_linear : Another detrend algorithm.

detrend_none : Another detrend algorithm.

detrend : A wrapper around all the detrend algorithms.

matplotlib.mlab.detrend_none

```
detrend_none(x, axis=None)
```

Return x : no detrending.

Parameters

x : any object

An object containing the data

axis : int

This parameter is ignored.

It is included for compatibility with detrend_mean

See Also

detrend_mean : Another detrend algorithm.

detrend_linear : Another detrend algorithm.

detrend : A wrapper around all the detrend algorithms.

matplotlib.mlab.psd

```
psd(x, NFFT=None, Fs=None, detrend=None, window=None, noverlap=None, pad_to=None, sides=None, scale_by_freq=None)
```

Compute the power spectral density.

The power spectral density P_{xx} by Welch's average periodogram method. The vector x is divided into $NFFT$ length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The $\frac{1}{N} \sum_{i=1}^N |fft(i)|^2$ of each segment i are averaged to compute P_{xx} .

If $\text{len}(x) < NFFT$, it will be zero padded to $NFFT$.

Parameters

`x` : 1-D array or sequence
Array or sequence containing the data

`Fs` : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `freqs`, in cycles per time unit.

`window` : callable or ndarray, default: ``.window_hanning``
A function or a vector of length $NFFT$. To create window vectors see ``.window_hanning``, ``.window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional
Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional
The number of points to which the data segment is padded when performing the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `n` parameter in the call to `~numpy.fft.fft`. The default is None, which sets `pad_to` equal to $NFFT$.

`NFFT` : int, default: 256
The number of data points used in each block for the FFT. A power 2 is most efficient. This should **NOT** be used to get zero padding, or the scaling of the result will be incorrect; use `pad_to` for this instead.

`detrend` : {'none', 'mean', 'linear'} or callable, default: 'none'
The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the `detrend` parameter is a vector, in Matplotlib it is a function. The `~matplotlib.mlab` module defines ``.detrend_none``, ``.detrend_mean``, and ``.detrend_linear``, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls ``.detrend_none``. 'mean' calls ``.detrend_mean``. 'linear' calls ``.detrend_linear``.

`scale_by_freq` : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap : int, default: 0 (no overlap)
The number of points of overlap between segments.

Returns

Pxx : 1-D array
The values for the power spectrum :math:`P_{xx}` (real valued)

freqs : 1-D array
The frequencies corresponding to the elements in *Pxx*

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

See Also

specgram
`specgram` differs in the default overlap; in not returning the mean of the segment periodograms; and in returning the times of the segments.

magnitude_spectrum : returns the magnitude spectrum.

csd : returns the spectral density between two signals.

matplotlib.mlab.specgram

```
specgram(x, NFFT=None, Fs=None, detrend=None, window=None, noverlap=None, pad_to=None,
sides=None, scale_by_freq=None, mode=None)
```

Compute a spectrogram.

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*.

Parameters

x : array-like
1-D array or sequence.

Fs : float, default: 2
The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window : callable or ndarray, default: ``.window_hanning``
A function or a vector of length *NFFT*. To create window vectors see

``window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`,
`numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a
function is passed as the argument, it must take a data segment as an
argument and return the windowed version of the segment.`

sides : {'default', 'onesided', 'twosided'}, optional
Which sides of the spectrum to return. 'default' is one-sided for real
data and two-sided for complex data. 'onesided' forces the return of a
one-sided spectrum, while 'twosided' forces two-sided.

pad_to : int, optional
The number of points to which the data segment is padded when performing
the FFT. This can be different from `*NFFT*`, which specifies the number
of data points used. While not increasing the actual resolution of the
spectrum (the minimum distance between resolvable peaks), this can give
more points in the plot, allowing for more detail. This corresponds to
the `*n*` parameter in the call to `~numpy.fft.fft`. The default is None,
which sets `*pad_to*` equal to `*NFFT*`

NFFT : int, default: 256
The number of data points used in each block for the FFT. A power 2 is
most efficient. This should *NOT* be used to get zero padding, or the
scaling of the result will be incorrect; use `*pad_to*` for this instead.

detrend : {'none', 'mean', 'linear'} or callable, default: 'none'
The function applied to each segment before fft-ing, designed to remove
the mean or linear trend. Unlike in MATLAB, where the `*detrend*` parameter
is a vector, in Matplotlib it is a function. The `:mod:`~matplotlib.mlab``
module defines `~.detrend_none`, `~.detrend_mean`, and `~.detrend_linear`,
but you can use a custom function as well. You can also use a string to
choose one of the functions: 'none' calls `~.detrend_none`. 'mean' calls
`~.detrend_mean`. 'linear' calls `~.detrend_linear`.

scale_by_freq : bool, default: True
Whether the resulting density values should be scaled by the scaling
frequency, which gives density in units of 1/Hz. This allows for
integration over the returned frequency values. The default is True for
MATLAB compatibility.

noverlap : int, default: 128
The number of points of overlap between blocks.

mode : str, default: 'psd'

What sort of spectrum to use:

'psd'

Returns the power spectral density.

'complex'

Returns the complex-valued frequency spectrum.

'magnitude'

Returns the magnitude spectrum.

'angle'

Returns the phase spectrum without unwrapping.

'phase'

Returns the phase spectrum with unwrapping.

Returns

spectrum : array-like
2D array, columns are the periodograms of successive segments.

freqs : array-like
1-D array, frequencies corresponding to the rows in *spectrum*.

t : array-like
1-D array, the times corresponding to midpoints of segments
(i.e the columns in *spectrum*).

See Also

psd : differs in the overlap and in the return values.
complex_spectrum : similar, but with complex valued frequencies.
magnitude_spectrum : similar single segment when *mode* is 'magnitude'.
angle_spectrum : similar to single segment when *mode* is 'angle'.
phase_spectrum : similar to single segment when *mode* is 'phase'.

Notes

detrend and *scale_by_freq* only apply when *mode* is set to 'psd'.

matplotlib.mlab.window_hanning

`window_hanning(x)`

Return *x* times the Hanning (or Hann) window of len(*x*).

See Also

window_none : Another window algorithm.

matplotlib.mlab.window_none

`window_none(x)`

No window function; simply return *x*.

See Also

window_hanning : Another window algorithm.

matplotlib.offsetbox

`offsetbox(...)`

Container classes for ``Artist``s.

``OffsetBox``

The base of all container artists defined in this module.

``AnchoredOffsetbox``, ``AnchoredText``

Anchor and align an arbitrary ``Artist`` or a text relative to the parent

axes or a specific anchor point.

``DrawingArea``

A container with fixed width and height. Children have a fixed position inside the container and may be clipped.

``HPacker``, ``VParser``

Containers for layouting their children vertically or horizontally.

``PaddedBox``

A container to add a padding around an ``Artist``.

``TextArea``

Contains a single ``Text`` instance.

matplotlib.offsetbox.AnchoredOffsetbox

```
AnchoredOffsetbox(loc, *, pad=0.4, borderpad=0.5, child=None, prop=None, frameon=True,
bbox_to_anchor=None, bbox_transform=None, **kwargs)
```

An `OffsetBox` placed according to location `*loc*`.

`AnchoredOffsetbox` has a single child. When multiple children are needed, use an extra `OffsetBox` to enclose them. By default, the offset box is anchored against its parent Axes. You may explicitly specify the `*bbox_to_anchor*`.

matplotlib.offsetbox.AnchoredText

```
AnchoredText(s, loc, *, pad=0.4, borderpad=0.5, prop=None, **kwargs)
```

`AnchoredOffsetbox` with `Text`.

matplotlib.offsetbox.AnnotationBbox

```
AnnotationBbox(offsetbox, xy, xybox=None, xycoords='data', boxcoords=None, *,
frameon=True, pad=0.4, annotation_clip=None, box_alignment=(0.5, 0.5), bboxprops=None,
arrowprops=None, fontsize=None, **kwargs)
```

Container for an ``OffsetBox`` referring to a specific position `*xy*`.

Optionally an arrow pointing from the offsetbox to `*xy*` can be drawn.

This is like ``Annotation``, but with ``OffsetBox`` instead of ``Text``.

matplotlib.offsetbox.AuxTransformBox

```
AuxTransformBox(aux_transform)
```

An `OffsetBox` with an auxiliary transform.

All child artists are first transformed with `*aux_transform*`, then translated with an offset (the same for all children) so the bounding box of the children matches the drawn box. (In other words, adding an arbitrary translation to `*aux_transform*` has no effect as it will be

cancelled out by the later offsetting.)

`AuxTransformBox`` is similar to `.DrawingArea``, except that the extent of the box is not predetermined but calculated from the window extent of its children, and the extent of the children will be calculated in the transformed coordinate.

matplotlib.offsetbox.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" `[[xmin, ymin], [xmax, ymax]]``.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" `(xmin, ymin, xmax, ymax)``

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" `(xmin, ymin, width, height)``.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting `ignore=True`` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify ``ignore`` explicitly. If not, the default value of ``ignore`` can be changed at any time by code with access to your Bbox, for example using the method `~.Bbox.ignore`.

****Properties of the ``null`` bbox****

.. note::

The current behavior of `Bbox.null()` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[-inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[-inf, -inf], [inf, inf]])
```

matplotlib.offsetbox.BboxBase

```
BboxBase(shorthand_name=None)
```

The base class of all bounding boxes.

This class is immutable; `Bbox` is a mutable subclass.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

matplotlib.offsetbox.BboxImage

```
BboxImage(bbox, *, cmap=None, norm=None, colorizer=None, interpolation=None,
origin=None, filternorm=True, filterrad=4.0, resample=False, **kwargs)
```

The Image class whose size is determined by the given bbox.

matplotlib.offsetbox.DraggableAnnotation

```
DraggableAnnotation(annotation, use_blit=False)
```

Helper base class for a draggable artist (legend, offsetbox).

Derived classes must override the following methods::

```
def save_offset(self):
    """
    Called when the object is picked for dragging; should save the
    reference position of the artist.
    """
```

```
def update_offset(self, dx, dy):
    """
    Called during the dragging; (*dx*, *dy*) is the pixel offset from
    the point where the mouse drag started.
    """
```

Optionally, you may override the following method::

```
def finalize_offset(self):
    """Called when the mouse is released."""
```

In the current implementation of `.DraggableLegend` and .DraggableAnnotation`, .update_offset` places the artists in display coordinates, and .finalize_offset` recalculates their position in axes coordinate and set a relevant attribute.`

matplotlib.offsetbox.DraggableBase

```
DraggableBase(ref_artist, use_blit=False)
```

Helper base class for a draggable artist (legend, offsetbox).

Derived classes must override the following methods::

```
def save_offset(self):
    """
    Called when the object is picked for dragging; should save the
    reference position of the artist.
    """
```

```
def update_offset(self, dx, dy):
    """
    Called during the dragging; (*dx*, *dy*) is the pixel offset from
    the point where the mouse drag started.
    """
```

Optionally, you may override the following method::

```
def finalize_offset(self):
    """Called when the mouse is released."""
```

In the current implementation of `.DraggableLegend`` and ``DraggableAnnotation``, ``update_offset`` places the artists in display coordinates, and ``finalize_offset`` recalculates their position in axes coordinate and set a relevant attribute.

matplotlib.offsetbox.DraggableOffsetBox

```
DraggableOffsetBox(ref_artist, offsetbox, use_blit=False)
```

Helper base class for a draggable artist (legend, offsetbox).

Derived classes must override the following methods::

```
def save_offset(self):
    """
    Called when the object is picked for dragging; should save the
    reference position of the artist.
    """
```

```
def update_offset(self, dx, dy):
    """
    Called during the dragging; (*dx*, *dy*) is the pixel offset from
    the point where the mouse drag started.
    """
```

Optionally, you may override the following method::

```
def finalize_offset(self):
    """Called when the mouse is released."""
```

In the current implementation of `.DraggableLegend`` and ``DraggableAnnotation``, ``update_offset`` places the artists in display coordinates, and ``finalize_offset`` recalculates their position in axes coordinate and set a relevant attribute.

matplotlib.offsetbox.DrawingArea

```
DrawingArea(width, height, xdescent=0.0, ydescent=0.0, clip=False)
```

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed. The children can be clipped at the boundaries of the parent.

matplotlib.offsetbox.FancyArrowPatch

```
FancyArrowPatch(posA=None, posB=None, *, path=None, arrowstyle='simple',
connectionstyle='arc3', patchA=None, patchB=None, shrinkA=2, shrinkB=2,
mutation_scale=1, mutation_aspect=1, **kwargs)
```

A fancy arrow patch.

It draws an arrow using the ``ArrowStyle``. It is primarily used by the ``~.axes.Axes.annotate`` method. For most purposes, use the `annotate` method for drawing arrows.

The head and tail positions are fixed at the specified start and end points of the arrow, but the size and shape (in display coordinates) of the arrow does not change when the axis is moved or zoomed.

matplotlib.offsetbox.FancyBboxPatch

```
FancyBboxPatch(xy, width, height, boxstyle='round', *, mutation_scale=1,
mutation_aspect=1, **kwargs)
```

A fancy box around a rectangle with lower left at `*xy* = (*x*, *y*)` with specified width and height.

`.FancyBboxPatch`` is similar to `.Rectangle``, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the style classes defined in `.BoxStyle``.

matplotlib.offsetbox.FontProperties

```
FontProperties(family=None, style=None, variant=None, weight=None, stretch=None,
size=None, fname=None, math_fontfamily=None)
```

A class for storing and manipulating font properties.

The font properties are the six properties described in the ``W3C Cascading Style Sheet, Level 1``
<<http://www.w3.org/TR/1998/REC-CSS2-19980512/>> ``_ font`` specification and `*math_fontfamily*` for math fonts:

- family: A list of font names in decreasing order of priority. The items may include a generic font family name, either 'sans-serif', 'serif', 'cursive', 'fantasy', or 'monospace'. In that case, the actual font to be used will be looked up from the associated rcParam during the search process in `.findfont``. Default: `:rc:`font.family``
- style: Either 'normal', 'italic' or 'oblique'. Default: `:rc:`font.style``
- variant: Either 'normal' or 'small-caps'. Default: `:rc:`font.variant``
- stretch: A numeric value in the range 0-1000 or one of 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded'. Default: `:rc:`font.stretch``
- weight: A numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'. Default: `:rc:`font.weight``
- size: Either a relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 10. Default: `:rc:`font.size``

- `math_fontfamily`: The family of fonts used to render math text.
Supported values are: 'dejavusans', 'dejavuserif', 'cm', 'stix', 'stixsans' and 'custom'. Default: `:rc:`mathtext.fontset``

Alternatively, a font may be specified using the absolute path to a font file, by using the `*fname*` kwarg. However, in this case, it is typically simpler to just pass the path (as a ``pathlib.Path``, not a ``str``) to the `*font*` kwarg of the ``Text`` object.

The preferred usage of font sizes is to use the relative values, e.g., 'large', instead of absolute font sizes, e.g., 12. This approach allows all text sizes to be made larger or smaller based on the font manager's default font size.

This class accepts a single positional string as `fontconfig_pattern`, or alternatively individual properties as keyword arguments::

```
FontProperties(pattern)
FontProperties(*, family=None, style=None, variant=None, ...)
```

This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

```
.. _fontconfig: https://www.freedesktop.org/wiki/Software/fontconfig/
.. _pattern:
https://www.freedesktop.org/software/fontconfig/fontconfig-user.html
```

Note that Matplotlib's internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in Matplotlib than in other applications that use fontconfig.

matplotlib.offsetbox.HPacker

```
HPacker(pad=0.0, sep=0.0, width=None, height=None, align='baseline', mode='fixed',
children=None)
```

HPacker packs its children horizontally, automatically adjusting their relative positions at draw time.

.. code-block:: none

```
+-----+
| Child 1 Child 2 Child 3 |
+-----+
```

matplotlib.offsetbox.OffsetBox

```
OffsetBox(*args, **kwargs)
```

A simple container artist.

The child artists are meant to be drawn at a relative position to its parent.

Being an artist itself, all parameters are passed on to `.Artist``.

matplotlib.offsetbox.OffsetImage

```
OffsetImage(arr, *, zoom=1, cmap=None, norm=None, interpolation=None, origin=None,
            filternorm=True, filterrad=4.0, resample=False, dpi_cor=True, **kwargs)
```

A simple container artist.

The child artists are meant to be drawn at a relative position to its parent.

Being an artist itself, all parameters are passed on to `.Artist``.

matplotlib.offsetbox.PackerBase

```
PackerBase(pad=0.0, sep=0.0, width=None, height=None, align='baseline', mode='fixed',
           children=None)
```

A simple container artist.

The child artists are meant to be drawn at a relative position to its parent.

Being an artist itself, all parameters are passed on to `.Artist``.

matplotlib.offsetbox.PaddedBox

```
PaddedBox(child, pad=0.0, *, draw_frame=False, patch_attrs=None)
```

A container to add a padding around an `.Artist``.

The `.PaddedBox`` contains a `.FancyBboxPatch`` that is used to visualize it when rendering.

.. code-block:: none

```
+-----+
| |
| |
| |
| <--pad--> Artist |
| ^ |
| pad |
| v |
+-----+
```

Attributes

pad : float

The padding in points.

patch : `.FancyBboxPatch``

When `*draw_frame*` is True, this `.FancyBboxPatch`` is made visible and creates a border around the box.

matplotlib.offsetbox.TextArea

```
TextArea(s, *, textprops=None, multilinebaseline=False)
```

The TextArea is a container artist for a single Text instance.

The text is placed at (0, 0) with baseline+left alignment, by default. The width and height of the TextArea instance is the width and height of its child text.

matplotlib.offsetbox.TransformedBbox

```
TransformedBbox(bbox, transform, **kwargs)
```

A `Bbox` that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.

matplotlib.offsetbox.VPacker

```
VPacker(pad=0.0, sep=0.0, width=None, height=None, align='baseline', mode='fixed', children=None)
```

VPacker packs its children vertically, automatically adjusting their relative positions at draw time.

.. code-block:: none

```
+-----+
| Child 1 |
| Child 2 |
| Child 3 |
+-----+
```

matplotlib.offsetbox.mbbox_artist

```
mbbox_artist(artist, renderer, props=None, fill=True)
```

A debug function to draw a rectangle around the bounding box returned by an artist's `.Artist.get_window_extent` to test whether the artist is returning the correct bbox.

props is a dict of rectangle props with the additional property 'pad' that sets the padding around the bbox in points.

matplotlib.patches

```
patches(...)
```

Patches are `.Artist`s` with a face color and an edge color.

matplotlib.patches.Affine2D

```
Affine2D(matrix=None, **kwargs)
```

A mutable 2D affine transformation.

matplotlib.patches.Annulus

```
Annulus(xy, r, width, angle=0.0, **kwargs)
```

An elliptical annulus.

matplotlib.patches.Arc

```
Arc(xy, width, height, *, angle=0.0, theta1=0.0, theta2=360.0, **kwargs)
```

An elliptical arc, i.e. a segment of an ellipse.

Due to internal optimizations, the arc cannot be filled.

matplotlib.patches.Arrow

```
Arrow(x, y, dx, dy, *, width=1.0, **kwargs)
```

An arrow patch.

matplotlib.patches.ArrowStyle

```
ArrowStyle(stylename, **kwargs)
```

`ArrowStyle` is a container class which defines several arrowstyle classes, which is used to create an arrow path along a given path. These are mainly used with `FancyArrowPatch`.

An arrowstyle object can be either created as::

```
ArrowStyle.Fancy(head_length=.4, head_width=.4, tail_width=.4)
```

or::

```
ArrowStyle("Fancy", head_length=.4, head_width=.4, tail_width=.4)
```

or::

```
ArrowStyle("Fancy", head_length=.4, head_width=.4, tail_width=.4)
```

The following classes are defined

```
=====  
=====  
Class Name Parameters  
=====
```

```
=====  
=====  
Curve ``-`` None
```

```
CurveA ``<-`` head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2,  
angleA=0, angleB=0, scaleA=None, scaleB=None
```

```
CurveB ``->`` head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2,  
angleA=0, angleB=0, scaleA=None, scaleB=None
```

```
CurveAB ``<->`` head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2,  
angleA=0, angleB=0, scaleA=None, scaleB=None
```

```

CurveFilledA ``<|`` head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2,
angleA=0, angleB=0, scaleA=None, scaleB=None
CurveFilledB ``->`` head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2,
angleA=0, angleB=0, scaleA=None, scaleB=None
CurveFilledAB ``<|>`` head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2,
lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None
BracketA ``|`` widthA=1.0, lengthA=0.2, angleA=0
BracketB ``-|`` widthB=1.0, lengthB=0.2, angleB=0
BracketAB ``|``-|`` widthA=1.0, lengthA=0.2, angleA=0, widthB=1.0, lengthB=0.2, angleB=0
BarAB ``|``-|`` widthA=1.0, angleA=0, widthB=1.0, angleB=0
BracketCurve ``|``->`` widthA=1.0, lengthA=0.2, angleA=None
CurveBracket ``<-|`` widthB=1.0, lengthB=0.2, angleB=None
Simple ``simple`` head_length=0.5, head_width=0.5, tail_width=0.2
Fancy ``fancy`` head_length=0.4, head_width=0.4, tail_width=0.4
Wedge ``wedge`` tail_width=0.3, shrink_factor=0.5

```

```

=====
=====

```

For an overview of the visual appearance, see
[:doc:`gallery/text_labels_and_annotations/fancyarrow_demo`](https://matplotlib.org/3.1.0/gallery/text_labels_and_annotations/fancyarrow_demo.html).

An instance of any arrow style class is a callable object,
 whose call signature is::

```
__call__(self, path, mutation_size, linewidth, aspect_ratio=1.)
```

and it returns a tuple of a `.Path`` instance and a boolean value. `*path*` is a `.Path`` instance along which the arrow will be drawn. `*mutation_size*` and `*aspect_ratio*` have the same meaning as in `BoxStyle``. `*linewidth*` is a line width to be stroked. This is meant to be used to correct the location of the head so that it does not overshoot the destination point, but not all classes support it.

Notes

`*angleA*` and `*angleB*` specify the orientation of the bracket, as either a clockwise or counterclockwise angle depending on the arrow type. 0 degrees means perpendicular to the line connecting the arrow's head and tail.

.. plot:: [gallery/text_labels_and_annotations/angles_on_bracket_arrows.py](https://matplotlib.org/3.1.0/gallery/text_labels_and_annotations/angles_on_bracket_arrows.py)

matplotlib.patches.BoxStyle

```
BoxStyle(stylename, **kwargs)
```

`BoxStyle`` is a container class which defines several boxstyle classes, which are used for `FancyBboxPatch``.

A style object can be created as::

```
BoxStyle.Round(pad=0.2)
```

or::

```
BoxStyle("Round", pad=0.2)
```

or::

```
BoxStyle("Round, pad=0.2")
```

The following boxstyle classes are defined.

```
=====
Class Name Parameters
=====
Square ``square`` pad=0.3
Circle ``circle`` pad=0.3
Ellipse ``ellipse`` pad=0.3
LArrow ``larrow`` pad=0.3
RArrow ``rarrow`` pad=0.3
DArrow ``darrow`` pad=0.3
Round ``round`` pad=0.3, rounding_size=None
Round4 ``round4`` pad=0.3, rounding_size=None
Sawtooth ``sawtooth`` pad=0.3, tooth_size=None
Roundtooth ``roundtooth`` pad=0.3, tooth_size=None
=====
```

An instance of a boxstyle class is a callable object, with the signature ::

```
__call__(self, x0, y0, width, height, mutation_size) -> Path
```

x0, **y0**, **width** and **height** specify the location and size of the box to be drawn; **mutation_size** scales the outline properties such as padding.

matplotlib.patches.CapStyle

```
CapStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the two endpoints (caps) of an unclosed line are drawn.

How to draw the start and end points of lines that represent a closed curve (i.e. that end in a `~.path.Path.CLOSEPOLY`) is controlled by the line's `JoinStyle`. For all other lines, how the start and end points are drawn is controlled by the `*CapStyle*`.

For a visual impression of each `*CapStyle*`, view these docs online `<CapStyle>` or run `CapStyle.demo`.

By default, `~.backend_bases.GraphicsContextBase` draws a stroked line as squared off at its endpoints.

****Supported values:****

.. rst-class:: value-list

'butt'

the line is squared off at its endpoint.
'projecting'
the line is squared off as in *butt*, but the filled in area extends beyond the endpoint a distance of ``linewidth/2``.
'round'
like *butt*, but a semicircular cap is added to the end of the line, of radius ``linewidth/2``.

.. plot::
:alt: Demo of possible CapStyle's

```
from matplotlib._enums import CapStyle
CapStyle.demo()
```

matplotlib.patches.Circle

```
Circle(xy, radius=5, **kwargs)
```

A circle patch.

matplotlib.patches.CirclePolygon

```
CirclePolygon(xy, radius=5, *, resolution=20, **kwargs)
```

A polygon-approximation of a circle patch.

matplotlib.patches.ConnectionPatch

```
ConnectionPatch(xyA, xyB, coordsA, coordsB=None, *, axesA=None, axesB=None,
arrowstyle='-', connectionstyle='arc3', patchA=None, patchB=None, shrinkA=0.0,
shrinkB=0.0, mutation_scale=10.0, mutation_aspect=None, clip_on=False, **kwargs)
```

A patch that connects two points (possibly in different Axes).

matplotlib.patches.ConnectionStyle

```
ConnectionStyle(stylename, **kwargs)
```

`ConnectionStyle` is a container class which defines several connectionstyle classes, which is used to create a path between two points. These are mainly used with `FancyArrowPatch`.

A connectionstyle object can be either created as::

```
ConnectionStyle.Arc3(rad=0.2)
```

or::

```
ConnectionStyle("Arc3", rad=0.2)
```

or::

```
ConnectionStyle("Arc3, rad=0.2")
```

The following classes are defined

===== Class Name Parameters =====

```
Arc3 ``arc3`` rad=0.0
Angle3 ``angle3`` angleA=90, angleB=0
Angle ``angle`` angleA=90, angleB=0, rad=0.0
Arc ``arc`` angleA=0, angleB=0, armA=None, armB=None, rad=0.0
Bar ``bar`` armA=0.0, armB=0.0, fraction=0.3, angle=None
=====
```

An instance of any connection style class is a callable object,
whose call signature is::

```
__call__(self, posA, posB,
         patchA=None, patchB=None,
         shrinkA=2., shrinkB=2.)
```

and it returns a `.Path`` instance. `*posA*` and `*posB*` are
tuples of (x, y) coordinates of the two points to be
connected. `*patchA*` (or `*patchB*`) is given, the returned path is
clipped so that it start (or end) from the boundary of the
patch. The path is further shrunk by `*shrinkA*` (or `*shrinkB*`)
which is given in points.

matplotlib.patches.Ellipse

```
Ellipse(xy, width, height, *, angle=0, **kwargs)
```

A scale-free ellipse.

matplotlib.patches.FancyArrow

```
FancyArrow(x, y, dx, dy, *, width=0.001, length_includes_head=False, head_width=None,
           head_length=None, shape='full', overhang=0, head_starts_at_zero=False, **kwargs)
```

Like Arrow, but lets you set head width and head height independently.

matplotlib.patches.FancyArrowPatch

```
FancyArrowPatch(posA=None, posB=None, *, path=None, arrowstyle='simple',
                connectionstyle='arc3', patchA=None, patchB=None, shrinkA=2, shrinkB=2,
                mutation_scale=1, mutation_aspect=1, **kwargs)
```

A fancy arrow patch.

It draws an arrow using the ``ArrowStyle``. It is primarily used by the
``~.axes.Axes.annotate`` method. For most purposes, use the `annotate` method for
drawing arrows.

The head and tail positions are fixed at the specified start and end points
of the arrow, but the size and shape (in display coordinates) of the arrow
does not change when the axis is moved or zoomed.

matplotlib.patches.FancyBboxPatch

```
FancyBboxPatch(xy, width, height, boxstyle='round', *, mutation_scale=1,
mutation_aspect=1, **kwargs)
```

A fancy box around a rectangle with lower left at `*xy* = (*x*, *y*)` with specified width and height.

`.FancyBboxPatch`` is similar to `.Rectangle``, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the style classes defined in `.BoxStyle``.

matplotlib.patches.JoinStyle

```
JoinStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the connection between two line segments is drawn.

For a visual impression of each `*JoinStyle*`, ``view these docs online <JoinStyle>``, or run ``JoinStyle.demo``.

Lines in Matplotlib are typically defined by a 1D ``~.path.Path`` and a finite ``linewidth``, where the underlying 1D ``~.path.Path`` represents the center of the stroked line.

By default, ``~.backend_bases.GraphicsContextBase`` defines the boundaries of a stroked line to simply be every point within some radius, ``linewidth/2``, away from any point of the center line. However, this results in corners appearing "rounded", which may not be the desired behavior if you are drawing, for example, a polygon or pointed star.

****Supported values:****

.. rst-class:: value-list

`'miter'`

the "arrow-tip" style. Each boundary of the filled-in area will extend in a straight line parallel to the tangent vector of the centerline at the point it meets the corner, until they meet in a sharp point.

`'round'`

stokes every point within a radius of ``linewidth/2`` of the center lines.

`'bevel'`

the "squared-off" style. It can be thought of as a rounded corner where the "circular" part of the corner has been cut off.

.. note::

Very long miter tips are cut off (to form a `*bevel*`) after a backend-dependent limit called the "miter limit", which specifies the maximum allowed ratio of miter length to line width. For example, the PDF backend uses the default value of 10 specified by the PDF standard, while the SVG backend does not even specify the miter limit, resulting in a default value of 4 per the SVG specification. Matplotlib does not

currently allow the user to adjust this parameter.

A more detailed description of the effect of a miter limit can be found in the `Mozilla Developer Docs`
<<https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-miterlimit>>`_

.. plot::
:alt: Demo of possible JoinStyle's

```
from matplotlib._enums import JoinStyle
JoinStyle.demo()
```

matplotlib.patches.NonIntersectingPathException

```
NonIntersectingPathException(...)
```

Inappropriate argument value (of correct type).

matplotlib.patches.Patch

```
Patch(*, edgecolor=None, facecolor=None, color=None, linewidth=None, linestyle=None,
      antialiased=None, hatch=None, fill=True, capstyle=None, joinstyle=None, **kwargs)
```

A patch is a 2D artist with a face color and an edge color.

If any of `*edgecolor*`, `*facecolor*`, `*linewidth*`, or `*antialiased*` are `*None*`, they default to their rc params setting.

matplotlib.patches.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- `*vertices*`: an (N, 2) float array of vertices
- `*codes*`: an N-length ``numpy.uint8`` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three ``CURVE4`` codes.

The code types are:

- ``STOP`` : 1 vertex (ignored)
A marker for the end of the entire path (currently not required and ignored)
- ``MOVETO`` : 1 vertex
Pick up the pen and move to the given vertex.
- ``LINETO`` : 1 vertex
Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.patches.PathPatch

```
PathPatch(path, **kwargs)
```

A general polycurve path patch.

matplotlib.patches.Polygon

```
Polygon(xy, *, closed=True, **kwargs)
```

A general polygon patch.

matplotlib.patches.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point `*xy*` and its `*width*` and `*height*`.

The rectangle extends from ``xy[0]`` to ``xy[0] + width`` in x-direction and from ``xy[1]`` to ``xy[1] + height`` in y-direction. ::

```
: +-----+
: |
: height |
```

```
: ||  
: (xy)---- width -----+
```

One may picture **xy** as the bottom left corner, but which corner **xy** is actually depends on the direction of the axis and the sign of **width** and **height**; e.g. **xy** would be the bottom right corner if the x-axis was inverted or if **width** was negative.

matplotlib.patches.RegularPolygon

```
RegularPolygon(xy, numVertices, *, radius=5, orientation=0, **kwargs)
```

A regular polygon patch.

matplotlib.patches.Shadow

```
Shadow(patch, ox, oy, *, shade=0.7, **kwargs)
```

A patch is a 2D artist with a face color and an edge color.

If any of **edgecolor**, **facecolor**, **linewidth**, or **antialiased** are **None**, they default to their rc params setting.

matplotlib.patches.StepPatch

```
StepPatch(values, edges, *, orientation='vertical', baseline=0, **kwargs)
```

A path patch describing a stepwise constant function.

By default, the path is not closed and starts and stops at baseline value.

matplotlib.patches.Wedge

```
Wedge(center, r, theta1, theta2, *, width=None, **kwargs)
```

Wedge shaped patch.

matplotlib.patches.bbox_artist

```
bbox_artist(artist, renderer, props=None, fill=True)
```

A debug function to draw a rectangle around the bounding box returned by an artist's `.Artist.get_window_extent` to test whether the artist is returning the correct bbox.

props is a dict of rectangle props with the additional property 'pad' that sets the padding around the bbox in points.

matplotlib.patches.draw_bbox

```
draw_bbox(bbox, renderer, color='k', trans=None)
```

A debug function to draw a rectangle around the bounding box returned by an artist's `.Artist.get_window_extent` to test whether the artist is returning the correct bbox.

matplotlib.patches.get_cos_sin

```
get_cos_sin(x0, y0, x1, y1)
```

No description available.

matplotlib.patches.get_intersection

```
get_intersection(cx1, cy1, cos_t1, sin_t1, cx2, cy2, cos_t2, sin_t2)
```

Return the intersection between the line through `(*cx1*, *cy1*)` at angle `*t1*` and the line through `(*cx2*, *cy2*)` at angle `*t2*`.

matplotlib.patches.get_parallels

```
get_parallels(bezier2, width)
```

Given the quadratic Bézier control points `*bezier2*`, returns control points of quadratic Bézier lines roughly parallel to given one separated by `*width*`.

matplotlib.patches.inside_circle

```
inside_circle(cx, cy, r)
```

Return a function that checks whether a point is in a circle with center `(*cx*, *cy*)` and radius `*r*`.

The returned function has the signature::

`f(xy: tuple[float, float]) -> bool`

matplotlib.patches.make_wedged_bezier2

```
make_wedged_bezier2(bezier2, width, w1=1.0, wm=0.5, w2=0.0)
```

Being similar to ``get_parallels``, returns control points of two quadratic Bézier lines having a width roughly parallel to given one separated by `*width*`.

matplotlib.patches.split_bezier_intersecting_with_closedpath

```
split_bezier_intersecting_with_closedpath(bezier, inside_closedpath, tolerance=0.01)
```

Split a Bézier curve into two at the intersection with a closed path.

Parameters

`bezier` : (N, 2) array-like

Control points of the Bézier segment. See ``BezierSegment``.

`inside_closedpath` : callable

A function returning True if a given point (x, y) is inside the closed path. See also `.find_bezier_t_intersecting_with_closedpath`.
tolerance : float
The tolerance for the intersection. See also `.find_bezier_t_intersecting_with_closedpath`.

Returns

left, right

Lists of control points for the two Bézier segments.

matplotlib.patches.split_path_inout

```
split_path_inout(path, inside, tolerance=0.01, reorder_inout=False)
```

Divide a path into two segments at the point where `inside(x, y)` becomes False.

matplotlib.path

```
path(...)
```

A module for dealing with the polylines used throughout Matplotlib.

The primary class for polyline handling in Matplotlib is `Path`. Almost all vector drawing makes use of `Path`'s somewhere in the drawing pipeline.

Whilst a `Path` instance itself cannot be drawn, some `.Artist` subclasses, such as `.PathPatch` and `.PathCollection`, can be used for convenient `Path` visualisation.

matplotlib.path.BezierSegment

```
BezierSegment(control_points)
```

A d-dimensional Bézier segment.

Parameters

control_points : (N, d) array

Location of the *N* control points.

matplotlib.path.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- `*vertices*`: an (N, 2) float array of vertices
- `*codes*`: an N-length `numpy.uint8` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three ``CURVE4`` codes.

The code types are:

- ``STOP`` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- ``MOVETO`` : 1 vertex

Pick up the pen and move to the given vertex.

- ``LINETO`` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.path.get_path_collection_extents

```
get_path_collection_extents(master_transform, paths, transforms, offsets,
                             offset_transform)
```

Get bounding box of a ``PathCollection``'s internal objects.

That is, given a sequence of ``Path``'s, ``Transform``'s objects, and offsets, as found in a ``PathCollection``, return the bounding box that encapsulates all of them.

Parameters

master_transform : `~matplotlib.transforms.Transform``
Global transformation applied to all paths.
paths : list of ``Path``
transforms : list of `~matplotlib.transforms.Affine2DBase``
If non-empty, this overrides `*master_transform*`.
offsets : (N, 2) array-like
offset_transform : `~matplotlib.transforms.Affine2DBase``
Transform applied to the offsets before offsetting the path.

Notes

The way that `*paths*`, `*transforms*` and `*offsets*` are combined follows the same method as for collections: each is iterated over independently, so if you have 3 paths (A, B, C), 2 transforms (α , β) and 1 offset (O), their combinations are as follows:

- (A, α , O)
- (B, β , O)
- (C, α , O)

matplotlib.path.simple_linear_interpolation

```
simple_linear_interpolation(a, steps)
```

Resample an array with ```steps - 1``` points between original point pairs.

Along each column of `*a*`, ```(steps - 1)``` points are introduced between each original values; the values are linearly interpolated.

Parameters

a : array, shape (n, ...)
steps : int

Returns

array
shape ```((n - 1) * steps + 1, ...)```

matplotlib.patheffects

```
patheffects(...)
```

Defines classes for path effects. The path effects are supported in ``.Text``, ``.Line2D`` and ``.Patch``.

.. seealso::
:ref:`patheffects_guide`

matplotlib.patheffects.AbstractPathEffect

```
AbstractPathEffect(offset=(0.0, 0.0))
```

A base class for path effects.

Subclasses should override the ``draw_path`` method to add effect functionality.

matplotlib.patheffects.Normal

```
Normal(offset=(0.0, 0.0))
```

The "identity" PathEffect.

The Normal PathEffect's sole purpose is to draw the original artist with no special path effect.

matplotlib.patheffects.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- **vertices**: an (N, 2) float array of vertices
- **codes**: an N-length `numpy.uint8` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three `CURVE4` codes.

The code types are:

- `STOP` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- `MOVETO` : 1 vertex

Pick up the pen and move to the given vertex.

- `LINETO` : 1 vertex

Draw a line from the current position to the given vertex.

- `CURVE3` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- `CURVE4` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- `CLOSEPOLY` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If **codes** is None, it is interpreted as a `MOVETO` followed by a series of `LINETO`.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being `None`.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.patheffects.PathEffectRenderer

```
PathEffectRenderer(path_effects, renderer)
```

Implements a Renderer which contains another renderer.

This proxy then intercepts draw calls, calling the appropriate :class:`AbstractPathEffect` draw method.

.. note::

Not all methods have been overridden on this RendererBase subclass. It may be necessary to add further methods to extend the PathEffects capabilities further.

matplotlib.patheffects.PathPatchEffect

```
PathPatchEffect(offset=(0, 0), **kwargs)
```

Draws a ``PathPatch`` instance whose Path comes from the original PathEffect artist.

matplotlib.patheffects.RendererBase

```
RendererBase()
```

An abstract base class to handle drawing/rendering operations.

The following methods must be implemented in the backend for full functionality (though just implementing ``draw_path`` alone would give a highly capable backend):

- * ``draw_path``
- * ``draw_image``
- * ``draw_gouraud_triangles``

The following methods *should* be implemented in the backend for optimization reasons:

```
* `draw_text`  
* `draw_markers`  
* `draw_path_collection`  
* `draw_quad_mesh`
```

matplotlib.patheffects.SimpleLineShadow

```
SimpleLineShadow(offset=(2, -2), shadow_color='k', alpha=0.3, rho=0.3, **kwargs)
```

A simple shadow via a line.

matplotlib.patheffects.SimplePatchShadow

```
SimplePatchShadow(offset=(2, -2), shadow_rgbFace=None, alpha=None, rho=0.3, **kwargs)
```

A simple shadow via a filled patch.

matplotlib.patheffects.Stroke

```
Stroke(offset=(0, 0), **kwargs)
```

A line based PathEffect which re-draws a stroke.

matplotlib.patheffects.TickedStroke

```
TickedStroke(offset=(0, 0), spacing=10.0, angle=45.0,  
length=np.float64(1.4142135623730951), **kwargs)
```

A line-based PathEffect which draws a path with a ticked style.

This line style is frequently used to represent constraints in optimization. The ticks may be used to indicate that one side of the line is invalid or to represent a closed boundary of a domain (i.e. a wall or the edge of a pipe).

The spacing, length, and angle of ticks can be controlled.

This line style is sometimes referred to as a hatched line.

See also the `:doc:`/gallery/misc/tickdstroke_demo`` example.

matplotlib.patheffects.withSimplePatchShadow

```
withSimplePatchShadow(offset=(2, -2), shadow_rgbFace=None, alpha=None, rho=0.3,  
**kwargs)
```

A shortcut PathEffect for applying `.SimplePatchShadow`` and then drawing the original Artist.

With this class you can use ::

```
artist.set_path_effects([patheffects.withSimplePatchShadow()])
```

as a shortcut for ::

```
artist.set_path_effects([patheffects.SimplePatchShadow(),
patheffects.Normal()])
```

matplotlib.patheffects.withStroke

```
withStroke(offset=(0, 0), **kwargs)
```

A shortcut PathEffect for applying `.Stroke`` and then drawing the original Artist.

With this class you can use ::

```
artist.set_path_effects([patheffects.withStroke()])
```

as a shortcut for ::

```
artist.set_path_effects([patheffects.Stroke(),
patheffects.Normal()])
```

matplotlib.patheffects.withTickedStroke

```
withTickedStroke(offset=(0, 0), spacing=10.0, angle=45.0,
length=np.float64(1.4142135623730951), **kwargs)
```

A shortcut PathEffect for applying `.TickedStroke`` and then drawing the original Artist.

With this class you can use ::

```
artist.set_path_effects([patheffects.withTickedStroke()])
```

as a shortcut for ::

```
artist.set_path_effects([patheffects.TickedStroke(),
patheffects.Normal()])
```

matplotlib.projections

```
projections(...)
```

Non-separable transforms that map from data space to screen space.

Projections are defined as `~.axes.Axes`` subclasses. They include the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- Transformations for the gridlines, ticks and ticklabels. Custom projections will often need to place these elements in special locations, and Matplotlib has a facility to help with doing so.
- Setting up default values (overriding `~.axes.Axes.cla``), since the defaults

for a rectilinear Axes may not be appropriate.

- Defining the shape of the Axes, for example, an elliptical Axes, that will be used to draw the background of the plot and for clipping any data elements.

- Defining custom locators and formatters for the projection. For example, in a geographic projection, it may be more convenient to display the grid in degrees, even if the data is in radians.

- Set up interactive panning and zooming. This is left as an "advanced" feature left to the reader, but there is an example of this for polar plots in ``matplotlib.projections.polar``.

- Any additional methods for additional convenience or features.

Once the projection Axes is defined, it can be used in one of two ways:

- By defining the class attribute ```name```, the projection Axes can be registered with ``matplotlib.projections.register_projection`` and subsequently simply invoked by name::

```
fig.add_subplot(projection="my_proj_name")
```

- For more complex, parameterisable projections, a generic "projection" object may be defined which includes the method ```_as_mpl_axes```. ```_as_mpl_axes``` should take no arguments and return the projection's Axes subclass and a dictionary of additional arguments to pass to the subclass' ```__init__``` method. Subsequently a parameterised projection can be initialised with::

```
fig.add_subplot(projection=MyProjection(param1=param1_value))
```

where `MyProjection` is an object which implements a ```_as_mpl_axes``` method.

A full-fledged and heavily annotated example is in `:doc:`gallery/misc/custom_projection``. The polar plot functionality in ``matplotlib.projections.polar`` may also be of interest.

matplotlib.projections.AitoffAxes

```
AitoffAxes(*args, **kwargs)
```

An abstract base class for geographic projections.

matplotlib.projections.HammerAxes

```
HammerAxes(*args, **kwargs)
```

An abstract base class for geographic projections.

matplotlib.projections.LambertAxes

```
LambertAxes(*args, center_longitude=0, center_latitude=0, **kwargs)
```

An abstract base class for geographic projections.

matplotlib.projections.MollweideAxes

```
MollweideAxes(*args, **kwargs)
```

An abstract base class for geographic projections.

matplotlib.projections.PolarAxes

```
PolarAxes(*args, theta_offset=0, theta_direction=1, rlabel_position=22.5, **kwargs)
```

A polar graph projection, where the input dimensions are **theta**, **r**.

Theta starts pointing east and goes anti-clockwise.

matplotlib.projections.ProjectionRegistry

```
ProjectionRegistry()
```

A mapping of registered projection names to projection classes.

matplotlib.projections.geo

```
geo(...)
```

No description available.

matplotlib.projections.get_projection_class

```
get_projection_class(projection=None)
```

Get a projection class from its name.

If **projection** is None, a standard rectilinear projection is returned.

matplotlib.projections.polar

```
polar(...)
```

No description available.

matplotlib.projections.register_projection

```
register_projection(cls)
```

No description available.

matplotlib.pyplot

```
pylab(...)
```

``pylab`` is a historic interface and its use is strongly discouraged. The equivalent replacement is ``matplotlib.pyplot``. See [:ref:`api_interfaces`](#) for a full overview of Matplotlib interfaces.

``pylab`` was designed to support a MATLAB-like way of working with all plotting related

functions directly available in the global namespace. This was achieved through a wildcard import (`from pylab import *`).

.. warning::

The use of `pylab` is discouraged for the following reasons:

`from pylab import *` imports all the functions from `matplotlib.pyplot`, `numpy`, `numpy.fft`, `numpy.linalg`, and `numpy.random`, and some additional functions into the global namespace.

Such a pattern is considered bad practice in modern python, as it clutters the global namespace. Even more severely, in the case of `pylab`, this will overwrite some builtin functions (e.g. the builtin `sum` will be replaced by `numpy.sum`), which can lead to unexpected behavior.

matplotlib.pylab.Annotation

```
Annotation(text, xy, xytext=None, xycoords='data', textcoords=None, arrowprops=None,
annotation_clip=None, **kwargs)
```

An `.Annotation` is a `.Text` that can refer to a specific position `*xy*`. Optionally an arrow pointing from the text to `*xy*` can be drawn.

Attributes

`xy`

The annotated position.

`xycoords`

The coordinate system for `*xy*`.

`arrow_patch`

A `.FancyArrowPatch` to point from `*xytext*` to `*xy*`.

matplotlib.pylab.Arrow

```
Arrow(x, y, dx, dy, *, width=1.0, **kwargs)
```

An arrow patch.

matplotlib.pylab.Artist

```
Artist()
```

Abstract base class for objects that render into a `FigureCanvas`.

Typically, all visible elements in a figure are subclasses of `Artist`.

matplotlib.pylab.AutoLocator

```
AutoLocator()
```

Place evenly spaced ticks, with the step size and maximum number of ticks chosen automatically.

This is a subclass of `~matplotlib.ticker.MaxNLocator`, with parameters `*nbins = 'auto'` and `*steps = [1, 2, 2.5, 5, 10]`.

matplotlib.pyplot.AxLine

```
AxLine(xy1, xy2, slope, **kwargs)
```

A helper class that implements `~.Axes.axline`, by recomputing the artist transform at draw time.

matplotlib.pyplot.Axes

```
Axes(fig, *args, facecolor=None, frameon=True, sharex=None, sharey=None, label='',  
xscale=None, yscale=None, box_aspect=None, forward_navigation_events='auto', **kwargs)
```

An Axes object encapsulates all the elements of an individual (sub-)plot in a figure.

It contains most of the (sub-)plot elements: `~.axis.Axis`, `~.axis.Tick`, `~.lines.Line2D`, `~.text.Text`, `~.patches.Polygon`, etc., and sets the coordinate system.

Like all visible elements in a figure, Axes is an `.Artist` subclass.

The `Axes` instance supports callbacks through a `callbacks` attribute which is a `~.cbook.CallbackRegistry` instance. The events you can connect to are `'xlim_changed'` and `'ylim_changed'` and the callback will be called with `func(*ax*)` where `*ax*` is the `Axes` instance.

.. note::

As a user, you do not instantiate Axes directly, but use Axes creation methods instead; e.g. from `~.pyplot` or `~.Figure`:
`~.pyplot.subplots`, `~.pyplot.subplot_mosaic` or `~.Figure.add_axes`.

matplotlib.pyplot.BackendFilter

```
BackendFilter(value, names=None, *, module=None, qualname=None, type=None, start=1,  
boundary=None)
```

Filter used with `:meth:`~matplotlib.backends.registry.BackendRegistry.list_builtins``

.. versionadded:: 3.9

matplotlib.pyplot.Button

```
Button(ax, label, image=None, color='0.85', hovercolor='0.95', *, useblit=True)
```

A GUI neutral button.

For the button to remain responsive you must keep a reference to it.
Call `~.on_clicked` to connect to the button.

Attributes

`ax`

The `~.axes.Axes` the button renders into.

`label`

A `.Text`` instance.
color
The color of the button when not hovering.
hovercolor
The color of the button when hovering.

matplotlib.pyplot.Circle

```
Circle(xy, radius=5, **kwargs)
```

A circle patch.

matplotlib.pyplot.Colorizer

```
Colorizer(cmap=None, norm=None)
```

Data to color pipeline.

This pipeline is accessible via `.Colorizer.to_rgba`` and executed via the `.Colorizer.norm`` and `.Colorizer.cmap`` attributes.

Parameters

cmap: `colorbar.Colorbar` or str or None, default: None
The colormap used to color data.

norm: `colors.Normalize` or str or None, default: None
The normalization used to normalize the data

matplotlib.pyplot.ColorizingArtist

```
ColorizingArtist(colorizer, **kwargs)
```

Base class for artists that make map data to color using a `.colorizer.Colorizer``.

The `.colorizer.Colorizer`` applies data normalization before returning RGBA colors from a `~matplotlib.colors.Colormap``.

matplotlib.pyplot.Colormap

```
Colormap(name, N=256)
```

Baseclass for all scalar to RGBA mappings.

Typically, `Colormap` instances are used to convert data values (floats) from the interval `[0, 1]` to the RGBA color that the respective `Colormap` represents. For scaling of data into the `[0, 1]` interval see `matplotlib.colors.Normalize``. Subclasses of `matplotlib.cm.ScalarMappable`` make heavy use of this `data -> normalize -> map-to-color`` processing chain.

matplotlib.pyplot.DateFormatter

```
DateFormatter(fmt, tz=None, *, usetex=None)
```

Format a tick (in days since the epoch) with a
`~datetime.datetime.strftime` format string.

matplotlib.pylab.DateLocator

```
DateLocator(tz=None)
```

Determines the tick locations when plotting dates.

This class is subclassed by other Locators and is not meant to be used on its own.

matplotlib.pylab.DayLocator

```
DayLocator(bymonthday=None, interval=1, tz=None)
```

Make ticks on occurrences of each day of the month. For example, 1, 15, 30.

matplotlib.pylab.Figure

```
Figure(figsize=None, dpi=None, *, facecolor=None, edgecolor=None, linewidth=0.0, frameon=None, subplotpars=None, tight_layout=None, constrained_layout=None, layout=None, **kwargs)
```

The top level container for all the plot elements.

See `matplotlib.figure` for an index of class methods.

Attributes

patch

The `.Rectangle` instance representing the figure background patch.

suppressComposite

For multiple images, the figure will make composite images depending on the renderer option_image_nocomposite function. If *suppressComposite* is a boolean, this will override the renderer.

matplotlib.pylab.FigureBase

```
FigureBase(**kwargs)
```

Base class for `.Figure` and `.SubFigure` containing the methods that add artists to the figure or subfigure, create Axes, etc.

matplotlib.pylab.FigureCanvasBase

```
FigureCanvasBase(figure=None)
```

The canvas the figure renders into.

Attributes

figure : `~matplotlib.figure.Figure`

A high-level figure instance.

matplotlib.pyplot.FigureManagerBase

```
FigureManagerBase(canvas, num)
```

A backend-independent abstraction of a figure container and controller.

The figure manager is used by pyplot to interact with the window in a backend-independent way. It's an adapter for the real (GUI) framework that represents the visual figure on screen.

The figure manager is connected to a specific canvas instance, which in turn is connected to a specific figure instance. To access a figure manager for a given figure in user code, you typically use `fig.canvas.manager`.

GUI backends derive from this class to translate common operations such as `*show*` or `*resize*` to the GUI-specific code. Non-GUI backends do not support these operations and can just use the base class.

The following basic operations are accessible:

Window operations

- `~.FigureManagerBase.show``
- `~.FigureManagerBase.destroy``
- `~.FigureManagerBase.full_screen_toggle``
- `~.FigureManagerBase.resize``
- `~.FigureManagerBase.get_window_title``
- `~.FigureManagerBase.set_window_title``

Key and mouse button press handling

The figure manager sets up default key and mouse button press handling by hooking up the `~.key_press_handler`` to the matplotlib event system. This ensures the same shortcuts and mouse actions across backends.

Other operations

Subclasses will have additional attributes and functions to access additional functionality. This is of course backend-specific. For example, most GUI backends have ```window``` and ```toolbar``` attributes that give access to the native GUI widgets of the respective framework.

Attributes

`canvas : `FigureCanvasBase``
The backend-specific canvas instance.

`num : int or str`
The figure number.

`key_press_handler_id : int`
The default key handler cid, when using the toolmanager.

To disable the default key press handling use::

```
figure.canvas.mpl_disconnect(  
figure.canvas.manager.key_press_handler_id)
```

button_press_handler_id : int

The default mouse button handler cid, when using the toolmanager.

To disable the default button press handling use::

```
figure.canvas.mpl_disconnect(  
figure.canvas.manager.button_press_handler_id)
```

matplotlib.pyplot.FixedFormatter

```
FixedFormatter(seq)
```

Return fixed strings for tick labels based only on position, not value.

.. note::

`FixedFormatter` should only be used together with `FixedLocator`.

Otherwise, the labels may end up in unexpected positions.

matplotlib.pyplot.FixedLocator

```
FixedLocator(locs, nbins=None)
```

Place ticks at a set of fixed values.

If **nbins** is None ticks are placed at all values. Otherwise, the **locs** array of possible positions will be subsampled to keep the number of ticks :math:\leq \text{nbins} + 1. The subsampling will be done to include the smallest absolute value; for example, if zero is included in the array of possibilities, then it will be included in the chosen ticks.

matplotlib.pyplot.FormatStrFormatter

```
FormatStrFormatter(fmt)
```

Use an old-style ('%' operator) format string to format the tick.

The format string should have a single variable format (%) in it.

It will be applied to the value (not the position) of the tick.

Negative numeric values (e.g., -1) will use a dash, not a Unicode minus; use `mathtext` to get a Unicode minus by wrapping the format specifier with `$` (e.g. `"$%g$"`).

matplotlib.pyplot.Formatter

```
Formatter()
```

Create a string based on a tick value and location.

matplotlib.pyplot.FuncFormatter

```
FuncFormatter(func)
```

Use a user-defined function for formatting.

The function should take in two inputs (a tick value ``x`` and a position ``pos``), and return a string containing the corresponding tick label.

matplotlib.pyplot.GridSpec

```
GridSpec(nrows, ncols, figure=None, left=None, bottom=None, right=None, top=None,
wspace=None, hspace=None, width_ratios=None, height_ratios=None)
```

A grid layout to place subplots within a figure.

The location of the grid cells is determined in a similar way to `.SubplotParams` using `*left*`, `*right*`, `*top*`, `*bottom*`, `*wspace*` and `*hspace*`.

Indexing a `GridSpec` instance returns a `.SubplotSpec`.

matplotlib.pyplot.HourLocator

```
HourLocator(byhour=None, interval=1, tz=None)
```

Make ticks on occurrences of each hour.

matplotlib.pyplot.IndexLocator

```
IndexLocator(base, offset)
```

Place ticks at every nth point plotted.

`IndexLocator` assumes index plotting; i.e., that the ticks are placed at integer values in the range between 0 and `len(data)` inclusive.

matplotlib.pyplot.Line2D

```
Line2D(xdata, ydata, *, linewidth=None, linestyle=None, color=None, gapcolor=None,
marker=None, markersize=None, markeredgewidth=None, markeredgewidth=None,
markerfacecolor=None, markerfacecoloralt='none', fillstyle=None, antialiased=None,
dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None,
pickradius=5, drawstyle=None, markevery=None, **kwargs)
```

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the `drawstyle`, e.g., one can create "stepped" lines in various styles.

matplotlib.pyplot.LinearLocator

```
LinearLocator(numticks=None, presets=None)
```

Place ticks at evenly spaced values.

The first time this function is called it will try to set the number of ticks to make a nice tick partitioning. Thereafter, the number of ticks will be fixed so that interactive navigation will be nice

matplotlib.pyplot.Locator

```
Locator()
```

Determine tick locations.

Note that the same locator should not be used across multiple `~matplotlib.axis.Axis`` because the locator stores references to the Axis data and view limits.

matplotlib.pyplot.LogFormatter

```
LogFormatter(base=10.0, labelOnlyBase=False, minor_thresholds=None, linthresh=None)
```

Base class for formatting ticks on a log or symlog scale.

It may be instantiated directly, or subclassed.

Parameters

base : float, default: 10.

Base of the logarithm used in all calculations.

labelOnlyBase : bool, default: False

If True, label ticks only at integer powers of base.

This is normally True for major ticks and False for minor ticks.

minor_thresholds : (subset, all), default: (1, 0.4)

If labelOnlyBase is False, these two numbers control the labeling of ticks that are not at integer powers of base; normally these are the minor ticks. The controlling parameter is the log of the axis data range. In the typical case where base is 10 it is the number of decades spanned by the axis, so we can call it 'numdec'. If ```numdec <= all```, all minor ticks will be labeled. If ```all < numdec <= subset```, then only a subset of minor ticks will be labeled, so as to avoid crowding. If ```numdec > subset``` then no minor ticks will be labeled.

linthresh : None or float, default: None

If a symmetric log scale is in use, its ```linthresh``` parameter must be supplied here.

Notes

The ``set_locs`` method must be called to enable the subsetting logic controlled by the ```minor_thresholds``` parameter.

In some cases such as the colorbar, there is no distinction between major and minor ticks; the tick locations might be set manually, or by a locator that puts ticks at integer powers of base and at intermediate locations. For this situation, disable the `minor_thresholds` logic by using ```minor_thresholds=(np.inf, np.inf)```, so that all ticks will be labeled.

To disable labeling of minor ticks when 'labelOnlyBase' is False, use ```minor_thresholds=(0, 0)```. This is the default for the "classic" style.

Examples

To label a subset of minor ticks when the view limits span up to 2 decades, and all of the ticks when zoomed in to 0.5 decades or less, use ```minor_thresholds=(2, 0.5)```.

To label all minor ticks when the view limits span up to 1.5 decades, use ```minor_thresholds=(1.5, 1.5)```.

matplotlib.pylab.LogFormatterExponent

```
LogFormatterExponent(base=10.0, labelOnlyBase=False, minor_thresholds=None,
linthresh=None)
```

Format values for log axis using ```exponent = log_base(value)```.

matplotlib.pylab.LogFormatterMathtext

```
LogFormatterMathtext(base=10.0, labelOnlyBase=False, minor_thresholds=None,
linthresh=None)
```

Format values for log axis using ```exponent = log_base(value)```.

matplotlib.pylab.LogLocator

```
LogLocator(base=10.0, subs=(1.0,), *, numticks=None)
```

Place logarithmically spaced ticks.

Places ticks at the values ```subs[j] * base**i```.

matplotlib.pylab.MaxNLocator

```
MaxNLocator(nbins=None, **kwargs)
```

Place evenly spaced ticks, with a cap on the total number of ticks.

Finds nice tick locations with no more than `:math:`nbins + 1`` ticks being within the view limits. Locations beyond the limits are added to support autoscaling.

matplotlib.pylab.MinuteLocator

```
MinuteLocator(byminute=None, interval=1, tz=None)
```

Make ticks on occurrences of each minute.

matplotlib.pyplot.MonthLocator

```
MonthLocator(bymonth=None, bymonthday=1, interval=1, tz=None)
```

Make ticks on occurrences of each month, e.g., 1, 3, 12.

matplotlib.pyplot.MouseButton

```
MouseButton(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
```

Enum where members are also (and must be) ints

matplotlib.pyplot.MultipleLocator

```
MultipleLocator(base=1.0, offset=0.0)
```

Place ticks at every integer multiple of a base plus an offset.

matplotlib.pyplot.Normalize

```
Normalize(vmin=None, vmax=None, clip=False)
```

A class which, when called, maps values within the interval ``[vmin, vmax]`` linearly to the interval ``[0.0, 1.0]``. The mapping of values outside ``[vmin, vmax]`` depends on `*clip*`.

Examples

::

```
x = [-2, -1, 0, 1, 2]
```

```
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=False)
norm(x) # [-0.5, 0., 0.5, 1., 1.5]
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=True)
norm(x) # [0., 0., 0.5, 1., 1.]
```

See Also

:ref:`colormapnorms`

matplotlib.pyplot.NullFormatter

```
NullFormatter()
```

Always return the empty string.

matplotlib.pyplot.NullLocator

```
NullLocator()
```

No ticks

matplotlib.pyplot.PolarAxes

```
PolarAxes(*args, theta_offset=0, theta_direction=1, rlabel_position=22.5, **kwargs)
```

A polar graph projection, where the input dimensions are **theta**, **r**.

Theta starts pointing east and goes anti-clockwise.

matplotlib.pyplot.Polygon

```
Polygon(xy, *, closed=True, **kwargs)
```

A general polygon patch.

matplotlib.pyplot.RRuleLocator

```
RRuleLocator(o, tz=None)
```

Determines the tick locations when plotting dates.

This class is subclassed by other Locators and is not meant to be used on its own.

matplotlib.pyplot.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point **xy** and its **width** and **height**.

The rectangle extends from ``xy[0]`` to ``xy[0] + width`` in x-direction and from ``xy[1]`` to ``xy[1] + height`` in y-direction. ::

```
: +-----+
: |
: height |
: |
: (xy)---- width ----+
```

One may picture **xy** as the bottom left corner, but which corner **xy** is actually depends on the direction of the axis and the sign of **width** and **height**; e.g. **xy** would be the bottom right corner if the x-axis was inverted or if **width** was negative.

matplotlib.pyplot.ScalarFormatter

```
ScalarFormatter(useOffset=None, useMathText=None, useLocale=None, *, usetex=None)
```

Format tick values as a number.

Parameters

useOffset : bool or float, default: :rc:`axes.formatter.useoffset`
Whether to use offset notation. See ``.set_useOffset``.
useMathText : bool, default: :rc:`axes.formatter.use_mathtext`
Whether to use fancy math formatting. See ``.set_useMathText``.

`useLocale` : bool, default: `:rc:`axes.formatter.use_locale``.
Whether to use locale settings for decimal sign and positive sign.
See ``set_useLocale``.
`usetex` : bool, default: `:rc:`text.usetex``
To enable/disable the use of TeX's math mode for rendering the numbers in the formatter.

.. versionadded:: 3.10

Notes

In addition to the parameters above, the formatting of scientific vs. floating point representation can be configured via ``set_scientific`` and ``set_powerlimits``).

****Offset notation and scientific notation****

Offset notation and scientific notation look quite similar at first sight. Both split some information from the formatted tick values and display it at the end of the axis.

- The scientific notation splits up the order of magnitude, i.e. a multiplicative scaling factor, e.g. ``1e6``.

- The offset notation separates an additive constant, e.g. ``+1e6``. The offset notation label is always prefixed with a ``+`` or ``-`` sign and is thus distinguishable from the order of magnitude label.

The following plot with x limits ``1_000_000`` to ``1_000_010`` illustrates the different formatting. Note the labels at the right edge of the x axis.

.. plot::

```
lim = (1_000_000, 1_000_010)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, gridspec_kw={'hspace': 2})
ax1.set(title='offset notation', xlim=lim)
ax2.set(title='scientific notation', xlim=lim)
ax2.xaxis.get_major_formatter().set_useOffset(False)
ax3.set(title='floating-point notation', xlim=lim)
ax3.xaxis.get_major_formatter().set_useOffset(False)
ax3.xaxis.get_major_formatter().set_scientific(False)
```

matplotlib.pyplot.SecondLocator

```
SecondLocator(bysecond=None, interval=1, tz=None)
```

Make ticks on occurrences of each second.

matplotlib.pyplot.Slider

```
Slider(ax, label, valmin, valmax, *, valinit=0.5, valfmt=None, closedmin=True,
closedmax=True, slidermin=None, slidermax=None, dragging=True, valstep=None,
orientation='horizontal', initcolor='r', track_color='lightgrey', handle_style=None,
**kwargs)
```

A slider representing a floating point range.

Create a slider from **valmin** to **valmax** in Axes **ax**. For the slider to remain responsive you must maintain a reference to it. Call `:meth:`on_changed`` to connect to the slider event.

Attributes

val : float

Slider value.

matplotlib.pyplot.Subplot

```
Subplot(fig, *args, facecolor=None, frameon=True, sharex=None, sharey=None, label='',
xscale=None, yscale=None, box_aspect=None, forward_navigation_events='auto', **kwargs)
```

An Axes object encapsulates all the elements of an individual (sub-)plot in a figure.

It contains most of the (sub-)plot elements: `~.axis.Axis``, `~.axis.Tick``, `~.lines.Line2D``, `~.text.Text``, `~.patches.Polygon``, etc., and sets the coordinate system.

Like all visible elements in a figure, Axes is an `~.Artist`` subclass.

The `~.Axes`` instance supports callbacks through a `callbacks` attribute which is a `~.cbook.CallbackRegistry`` instance. The events you can connect to are `'xlim_changed'` and `'ylim_changed'` and the callback will be called with `func(*ax*)` where **ax** is the `~.Axes`` instance.

.. note::

As a user, you do not instantiate Axes directly, but use Axes creation methods instead; e.g. from `~.pyplot`` or `~.Figure``: `~.pyplot.subplots``, `~.pyplot.subplot_mosaic`` or `~.Figure.add_axes``.

matplotlib.pyplot.SubplotSpec

```
SubplotSpec(gridspec, num1, num2=None)
```

The location of a subplot in a `~.GridSpec``.

.. note::

Likely, you will never instantiate a `~.SubplotSpec`` yourself. Instead, you will typically obtain one from a `~.GridSpec`` using item-access.

Parameters

gridspec : `~matplotlib.gridspec.GridSpec``

The GridSpec, which the subplot is referencing.

num1, num2 : int

The subplot will occupy the *num1*-th cell of the given *gridspec*. If *num2* is provided, the subplot will span between *num1*-th cell and *num2*-th cell ****inclusive****.

The index starts from 0.

matplotlib.pyplot.Text

```
Text(x=0, y=0, text='', *, color=None, verticalalignment='baseline',
horizontalalignment='left', multialignment=None, fontproperties=None, rotation=None,
linespacing=None, rotation_mode=None, usetex=None, wrap=False,
transform_rotates_text=False, parse_math=None, antialiased=None, **kwargs)
```

Handle storing and drawing of text in window or data coordinates.

matplotlib.pyplot.TickHelper

```
TickHelper()
```

No description available.

matplotlib.pyplot.WeekdayLocator

```
WeekdayLocator(byweekday=1, interval=1, tz=None)
```

Make ticks on occurrences of each weekday.

matplotlib.pyplot.Widget

```
Widget()
```

Abstract base class for GUI neutral widgets.

matplotlib.pyplot.YearLocator

```
YearLocator(base=1, month=1, day=1, tz=None)
```

Make ticks on a given day of each year that is a multiple of base.

Examples::

```
# Tick every year on Jan 1st
```

```
locator = YearLocator()
```

```
# Tick every 5 years on July 4th
```

```
locator = YearLocator(5, month=7, day=4)
```

matplotlib.pyplot.acorr

```
acorr(x: 'ArrayLike', *, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray,
LineCollection | Line2D, Line2D | None]'
```

Plot the autocorrelation of *x*.

Parameters

`x` : array-like

Not run through Matplotlib's unit conversion, so this should be a unit-less array.

`detrend` : callable, default: ``mlab.detrend_none`` (no detrending)

A detrending function applied to `*x*`. It must have the signature ::

`detrend(x: np.ndarray) -> np.ndarray`

`normed` : bool, default: True

If `True``, input vectors are normalised to unit length.

`usevlines` : bool, default: True

Determines the plot style.

If `True``, vertical lines are plotted from 0 to the `acorr` value using ``Axes.vlines``. Additionally, a horizontal line is plotted at `y=0` using ``Axes.axhline``.

If `False``, markers are plotted at the `acorr` values using ``Axes.plot``.

`maxlags` : int, default: 10

Number of lags to show. If `None``, will return all `2 * len(x) - 1`` lags.

Returns

`lags` : array (length `2*maxlags+1``)

The lag vector.

`c` : array (length `2*maxlags+1``)

The auto correlation vector.

`line` : ``LineCollection`` or ``Line2D``

``Artist`` added to the Axes of the correlation:

- ``LineCollection`` if `*usevlines*` is True.

- ``Line2D`` if `*usevlines*` is False.

`b` : `~matplotlib.lines.Line2D`` or None

Horizontal line at 0 if `*usevlines*` is True

None `*usevlines*` is False.

Other Parameters

`linestyle` : `~matplotlib.lines.Line2D`` property, optional

The linestyle for plotting the data points.

Only used if `*usevlines*` is `False``.

`marker` : str, default: 'o'

The marker for plotting the data points.

Only used if `*usevlines*` is `False``.

`data` : indexable object, optional

If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

x

****kwargs**

Additional parameters are passed to `.Axes.vlines`` and `.Axes.axhline`` if `*usevlines*` is ``True``; otherwise they are passed to `.Axes.plot``.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.acorr``.

The cross correlation is performed with ``numpy.correlate`` with ```mode = "full"```.

matplotlib.pyplot.angle_spectrum

```
angle_spectrum(x: 'ArrayLike', *, Fs: 'float | None' = None, Fc: 'int | None' = None, window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the angle spectrum.

Compute the angle spectrum (wrapped phase spectrum) of `*x*`. Data is padded to a length of `*pad_to*` and the windowing function `*window*` is applied to the signal.

Parameters

`x` : 1-D array or sequence
Array or sequence containing the data.

`Fs` : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

`window` : callable or ndarray, default: `.window_hanning``

A function or a vector of length `*NFFT*`. To create window vectors see `.window_hanning``, `.window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to `~numpy.fft.fft`. The default is `None`, which sets `*pad_to*` equal to the length of the input signal (i.e. no padding).

`Fc` : int, default: 0

The center frequency of `*x*`, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

`spectrum` : 1-D array

The values for the angle spectrum in radians (real valued).

`freqs` : 1-D array

The frequencies corresponding to the elements in `*spectrum*`.

`line` : `~matplotlib.lines.Line2D`

The line created by this function.

Other Parameters

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`

`**kwargs`

Keyword arguments control the `~matplotlib.lines.Line2D` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or `None`

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase` or `None`

`clip_on`: bool

`clip_path`: `Patch` or (`Path`, `Transform`) or `None`

`color` or `c`: `~matplotlib.colors.ColorType`

`dash_capstyle`: `~matplotlib.lines.Line2D.CapStyle` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: `~matplotlib.lines.Line2D.JoinStyle` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (`None`, `None`)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `~matplotlib.colors.ColorType` or `None`

`gid`: str

`in_layout`: bool

`label`: object

linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
 linewidth or lw: float
 marker: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``
 markeredgewidth or mec: `:mpltype:`color``
 markeredgewidth or mew: float
 markerfacecolor or mfc: `:mpltype:`color``
 markerfacecoloralt or mfcalt: `:mpltype:`color``
 markersize or ms: float
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]
 mouseover: bool
 path_effects: list of `~.AbstractPathEffect``
 picker: float or callable[[Artist, Event], tuple[bool, dict]]
 pickradius: float
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 solid_capstyle: `~.CapStyle`` or {'butt', 'projecting', 'round'}
 solid_joinstyle: `~.JoinStyle`` or {'miter', 'round', 'bevel'}
 transform: unknown
 url: str
 visible: bool
 xdata: 1D array
 ydata: 1D array
 zorder: float

See Also

magnitude_spectrum
 Plots the magnitudes of the corresponding frequencies.
 phase_spectrum
 Plots the unwrapped version of this function.
 specgram
 Can plot the angle spectrum of segments within the signal in a colormap.

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for `~.axes.Axes.angle_spectrum``.

matplotlib.pyplot.annotate

```

annotate(text: 'str', xy: 'tuple[float, float]', xytext: 'tuple[float, float] | None'
= None, xycoords: 'CoordsType' = 'data', textcoords: 'CoordsType | None' = None,
arrowprops: 'dict[str, Any] | None' = None, annotation_clip: 'bool | None' = None,
**kwargs) -> 'Annotation'

```

Annotate the point `*xy*` with text `*text*`.

In the simplest form, the text is placed at `*xy*`.

Optionally, the text can be displayed in another position `*xytext*`.

An arrow pointing from the text to the annotated point `*xy*` can then

be added by defining `*arrowprops*`.

Parameters

`text` : str

The text of the annotation.

`xy` : (float, float)

The point `*(x, y)*` to annotate. The coordinate system is determined by `*xycoords*`.

`xytext` : (float, float), default: `*xy*`

The position `*(x, y)*` to place the text at. The coordinate system is determined by `*textcoords*`.

`xycoords` : single or two-tuple of str or ``Artist`` or ``Transform`` or callable, default: `'data'`

The coordinate system that `*xy*` is given in. The following types of values are supported:

- One of the following strings:

Value	Description
'figure points'	Points from the lower left of the figure
'figure pixels'	Pixels from the lower left of the figure
'figure fraction'	Fraction of figure from lower left
'subfigure points'	Points from the lower left of the subfigure
'subfigure pixels'	Pixels from the lower left of the subfigure
'subfigure fraction'	Fraction of subfigure from lower left
'axes points'	Points from lower left corner of the Axes
'axes pixels'	Pixels from lower left corner of the Axes
'axes fraction'	Fraction of Axes from lower left
'data'	Use the coordinate system of the object being annotated (default)
'polar'	<code>*(theta, r)*</code> if not native 'data' coordinates

Note that 'subfigure pixels' and 'figure pixels' are the same for the parent figure, so users who want code that is usable in a subfigure can use 'subfigure pixels'.

- An ``Artist``: `*xy*` is interpreted as a fraction of the artist's `~matplotlib.transforms.Bbox``. E.g. `*(0, 0)*` would be the lower left corner of the bounding box and `*(0.5, 1)*` would be the center top of the bounding box.

- A ``Transform`` to transform `*xy*` to screen coordinates.

- A function with one of the following signatures::

```
def transform(renderer) -> Bbox
def transform(renderer) -> Transform
```

where `*renderer*` is a ``RendererBase`` subclass.

The result of the function is interpreted like the ``Artist`` and ``Transform`` cases above.

- A tuple `*(xcoords, ycoords)*` specifying separate coordinate systems for `*x*` and `*y*`. `*xcoords*` and `*ycoords*` must each be of one of the above described types.

See :ref:`plotting-guide-annotation` for more details.

`textcoords` : single or two-tuple of str or ``Artist`` or ``Transform`` or callable, default: value of `*xycoords*`
The coordinate system that `*xytext*` is given in.

All `*xycoords*` values are valid as well as the following strings:

=====	
Value	Description
=====	
'offset points'	Offset, in points, from the <code>*xy*</code> value
'offset pixels'	Offset, in pixels, from the <code>*xy*</code> value
'offset fontsize'	Offset, relative to fontsize, from the <code>*xy*</code> value
=====	

`arrowprops` : dict, optional

The properties used to draw a ``FancyArrowPatch`` arrow between the positions `*xy*` and `*xytext*`. Defaults to None, i.e. no arrow is drawn.

For historical reasons there are two different ways to specify arrows, "simple" and "fancy":

****Simple arrow:****

If `*arrowprops*` does not contain the key 'arrowstyle' the allowed keys are:

=====	
Key	Description
=====	
width	The width of the arrow in points
headwidth	The width of the base of the arrow head in points
headlength	The length of the arrow head in points
shrink	Fraction of total length to shrink from both ends
?	Any <code>`FancyArrowPatch`</code> property
=====	

The arrow is attached to the edge of the text box, the exact position (corners or centers) depending on where it's pointing to.

****Fancy arrow:****

This is used if 'arrowstyle' is provided in the `*arrowprops*`.

Valid keys are the following ``FancyArrowPatch`` parameters:

=====

Key Description

=====

`arrowstyle` The arrow style
`connectionstyle` The connection style
`relpos` See below; default is (0.5, 0.5)
`patchA` Default is bounding box of the text
`patchB` Default is None
`shrinkA` In points. Default is 2 points
`shrinkB` In points. Default is 2 points
`mutation_scale` Default is text size (in points)
`mutation_aspect` Default is 1
? Any ``FancyArrowPatch`` property

=====

The exact starting point position of the arrow is defined by `*relpos*`. It's a tuple of relative coordinates of the text box, where (0, 0) is the lower left corner and (1, 1) is the upper right corner. Values <0 and >1 are supported and specify points outside the text box. By default (0.5, 0.5), so the starting point is centered in the text box.

`annotation_clip` : bool or None, default: None
Whether to clip (i.e. not draw) the annotation when the annotation point `*xy*` is outside the Axes area.

- If `*True*`, the annotation will be clipped when `*xy*` is outside the Axes.
- If `*False*`, the annotation will always be drawn.
- If `*None*`, the annotation will be clipped when `*xy*` is outside the Axes and `*xycoords*` is 'data'.

`**kwargs`
Additional kwargs are passed to ``Text``.

Returns

``Annotation``

See Also

`:ref:`annotations``

Notes

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``axes.Axes.annotate``.

matplotlib.pyplot.arrow

```
arrow(x: 'float', y: 'float', dx: 'float', dy: 'float', **kwargs) -> 'FancyArrow'
```

[*Discouraged*] Add an arrow to the Axes.

This draws an arrow from `` (x, y) `` to `` $(x+dx, y+dy)$ ``.

.. admonition:: Discouraged

The use of this method is discouraged because it is not guaranteed that the arrow renders reasonably. For example, the resulting arrow is affected by the Axes aspect ratio and limits, which may distort the arrow.

Consider using ``~.Axes.annotate`` without a text instead, e.g. ::

```
ax.annotate("", xytext=(0, 0), xy=(0.5, 0.5),
arrowprops=dict(arrowstyle="->"))
```

Parameters

x, y : float

The x and y coordinates of the arrow base.

dx, dy : float

The length of the arrow along x and y direction.

width : float, default: 0.001

Width of full arrow tail.

length_includes_head : bool, default: False

True if head is to be counted in calculating the length.

head_width : float or None, default: 3*width

Total width of the full arrow head.

head_length : float or None, default: 1.5*head_width

Length of arrow head.

shape : {'full', 'left', 'right'}, default: 'full'

Draw the left-half, right-half, or full arrow.

overhang : float, default: 0

Fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

head_starts_at_zero : bool, default: False

If True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

****kwargs**

`.Patch` properties:

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: unknown

animated: bool
 antialiased or aa: bool or None
 capstyle: ``~matplotlib.transforms.BboxBase`` or `{'butt', 'projecting', 'round'}`
 clip_box: ``~matplotlib.transforms.BboxBase`` or None
 clip_on: bool
 clip_path: Patch or (Path, Transform) or None
 color: `:mpltype:`color``
 edgecolor or ec: `:mpltype:`color`` or None
 facecolor or fc: `:mpltype:`color`` or None
 figure: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``
 fill: bool
 gid: str
 hatch: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`
 hatch_linewidth: unknown
 in_layout: bool
 joinstyle: ``~matplotlib.transforms.Transform`` or `{'miter', 'round', 'bevel'}`
 label: object
 linestyle or ls: `{'-', '--', '-.', ':', ''}`, (offset, on-off-seq), ...}
 linewidth or lw: float or None
 mouseover: bool
 path_effects: list of ``~matplotlib.transforms.Transform``
 picker: None or bool or float or callable
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 transform: ``~matplotlib.transforms.Transform``
 url: str
 visible: bool
 zorder: float

Returns

``~matplotlib.transforms.Transform``
 The created ``~matplotlib.transforms.Transform`` object.

Notes

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``~matplotlib.transforms.Transform``.

matplotlib.pyplot.autoscale

```
autoscale(enable: 'bool' = True, axis: "Literal['both', 'x', 'y']" = 'both', tight:
'bool | None' = None) -> 'None'
```

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling.

It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or Axes.

Parameters

enable : bool or None, default: True

True turns autoscaling on, False turns it off.

None leaves the autoscaling state unchanged.

axis : {'both', 'x', 'y'}, default: 'both'

The axis on which to operate. (For 3D Axes, *axis* can also be set to 'z', and 'both' refers to all three Axes.)

tight : bool or None, default: None

If True, first set the margins to zero. Then, this argument is forwarded to `~.axes.Axes.autoscale_view`` (regardless of its value); see the description of its behavior there.

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for `~.axes.Axes.autoscale``.

matplotlib.pyplot.autumn

```
autumn() -> 'None'
```

Set the colormap to 'autumn'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

matplotlib.pyplot.axes

```
axes(arg: 'None | tuple[float, float, float, float]' = None, **kwargs) -> 'matplotlib.axes.Axes'
```

Add an Axes to the current figure and make it the current Axes.

Call signatures::

```
plt.axes()
plt.axes(rect, projection=None, polar=False, **kwargs)
plt.axes(ax)
```

Parameters

arg : None or 4-tuple

The exact behavior of this function depends on the type:

- *None*: A new full window Axes is added using ```subplot(**kwargs)```.
- 4-tuple of floats *rect* = ```(left, bottom, width, height)```. A new Axes is added with dimensions *rect* in normalized (0, 1) units using `~.Figure.add_axes`` on the current figure.

projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional

The projection type of the `~.axes.Axes``. *str* is the name of a custom projection, see `~matplotlib.projections``. The default

None results in a 'rectilinear' projection.

polar : bool, default: False

If True, equivalent to projection='polar'.

sharex, sharey : `~matplotlib.axes.Axes`, optional

Share the x or y `~matplotlib.axis` with sharex and/or sharey.

The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

label : str

A label for the returned Axes.

Returns

`~.axes.Axes`, or a subclass of `~.axes.Axes`

The returned Axes class depends on the projection used. It is

`~.axes.Axes` if rectilinear projection is used and

`.projections.polar.PolarAxes` if polar projection is used.

Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned Axes class. The keyword arguments for the rectilinear Axes class `~.axes.Axes` can be found in the following table but there might also be other keyword arguments if another projection is used, see the actual Axes class.

Properties:

adjustable: {'box', 'datalim'}

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None

anchor: (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...}

animated: bool

aspect: {'auto', 'equal'} or float

autoscale_on: bool

autoscalex_on: unknown

autoscaley_on: unknown

axes_locator: Callable[[Axes, Renderer], Bbox]

axisbelow: bool or 'line'

box_aspect: float or None

clip_box: `~matplotlib.transforms.BboxBase` or None

clip_on: bool

clip_path: Patch or (Path, Transform) or None

facecolor or fc: :mpltype:color

figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`

forward_navigation_events: bool or "auto"

frame_on: bool

gid: str

in_layout: bool

label: object

mouseover: bool

navigate: bool
 navigate_mode: unknown
 path_effects: list of ``AbstractPathEffect``
 picker: None or bool or float or callable
 position: [left, bottom, width, height] or ``~matplotlib.transforms.Bbox``
 prop_cycle: ``~cycler.Cycler``
 rasterization_zorder: float or None
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 subplotspec: unknown
 title: str
 transform: ``~matplotlib.transforms.Transform``
 url: str
 visible: bool
 xbound: (lower: float, upper: float)
 xlabel: str
 xlim: (left: float, right: float)
 xmargin: float greater than -0.5
 xscale: unknown
 xticklabels: unknown
 xticks: unknown
 ybound: (lower: float, upper: float)
 ylabel: str
 ylim: (bottom: float, top: float)
 ymargin: float greater than -0.5
 yscale: unknown
 yticklabels: unknown
 yticks: unknown
 zorder: float

See Also

[.Figure.add_axes](#)
[.pyplot.subplot](#)
[.Figure.add_subplot](#)
[.Figure.subplots](#)
[.pyplot.subplots](#)

Examples

::

```
# Creating a new full window Axes
plt.axes()
```

```
# Creating a new Axes with specified dimensions and a grey background
plt.axes((left, bottom, width, height), facecolor='grey')
```

matplotlib.pyplot.axhline

```
axhline(y: 'float' = 0, xmin: 'float' = 0, xmax: 'float' = 1, **kwargs) -> 'Line2D'
```

Add a horizontal line spanning the whole or fraction of the Axes.

Note: If you want to set x-limits in data coordinates, use
`~.Axes.hlines`` instead.

Parameters

`y` : float, default: 0

y position in :ref:`data coordinates <coordinate-systems>`.

`xmin` : float, default: 0

The start x-position in :ref:`axes coordinates <coordinate-systems>`.

Should be between 0 and 1, 0 being the far left of the plot,

1 the far right of the plot.

`xmax` : float, default: 1

The end x-position in :ref:`axes coordinates <coordinate-systems>`.

Should be between 0 and 1, 0 being the far left of the plot,

1 the far right of the plot.

Returns

`~matplotlib.lines.Line2D``

A `~.Line2D`` specified via two points `((xmin, y))``, `((xmax, y))``.

Its transform is set such that `*x*` is in

:ref:`axes coordinates <coordinate-systems>` and `*y*` is in

:ref:`data coordinates <coordinate-systems>`.

This is still a generic line and the horizontal character is only realized through using identical `*y*` values for both points. Thus, if you want to change the `*y*` value later, you have to provide two values `line.set_ydata([3, 3])``.

Other Parameters

`**kwargs`

Valid keyword arguments are `~.Line2D`` properties, except for `'transform'`:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `~mpltype:`color``

`dash_capstyle`: `~.CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: `~.JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

gapcolor: :mpltype:`color` or None
 gid: str
 in_layout: bool
 label: object
 linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
 linewidth or lw: float
 marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
 markeredgcolor or mec: :mpltype:`color`
 markeredgewidth or mew: float
 markerfacecolor or mfc: :mpltype:`color`
 markerfacecoloralt or mfcalt: :mpltype:`color`
 markersize or ms: float
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]
 mouseover: bool
 path_effects: list of `~.AbstractPathEffect`
 picker: float or callable[[Artist, Event], tuple[bool, dict]]
 pickradius: float
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 solid_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}
 solid_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}
 transform: unknown
 url: str
 visible: bool
 xdata: 1D array
 ydata: 1D array
 zorder: float

See Also

hlines : Add horizontal lines in data coordinates.
 axhspan : Add a horizontal span (rectangle) across the axis.
 axline : Add a line with an arbitrary slope.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.axhline`.

Examples

* draw a thick red hline at 'y' = 0 that spans the xrange::

```
>>> axhline(linewidth=4, color='r')
```

* draw a default hline at 'y' = 1 that spans the xrange::

```
>>> axhline(y=1)
```

* draw a default hline at 'y' = .5 that spans the middle half of the xrange::

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

matplotlib.pyplot.axhspan

```
axhspan(ymin: 'float', ymax: 'float', xmin: 'float' = 0, xmax: 'float' = 1, **kwargs)
-> 'Rectangle'
```

Add a horizontal span (rectangle) across the Axes.

The rectangle spans from *ymin* to *ymax* vertically, and, by default, the whole x-axis horizontally. The x-span can be set using *xmin* (default: 0) and *xmax* (default: 1) which are in axis units; e.g. ```xmin = 0.5``` always refers to the middle of the x-axis regardless of the limits set by `~.Axes.set_xlim``.

Parameters

ymin : float
Lower y-coordinate of the span, in data units.
ymax : float
Upper y-coordinate of the span, in data units.
xmin : float, default: 0
Lower x-coordinate of the span, in x-axis (0-1) units.
xmax : float, default: 1
Upper x-coordinate of the span, in x-axis (0-1) units.

Returns

`~matplotlib.patches.Rectangle``
Horizontal span (rectangle) from (xmin, ymin) to (xmax, ymax).

Other Parameters

****kwargs** : `~matplotlib.patches.Rectangle`` properties

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
alpha: float or None
angle: unknown
animated: bool
antialiased or **aa**: bool or None
bounds: (left, bottom, width, height)
capstyle: `~.CapStyle`` or {'butt', 'projecting', 'round'}
clip_box: `~matplotlib.transforms.BboxBase`` or None
clip_on: bool
clip_path: Patch or (Path, Transform) or None
color: :mpltype:`color`
edgecolor or **ec**: :mpltype:`color` or None
facecolor or **fc**: :mpltype:`color` or None
figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``
fill: bool
gid: str
hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
hatch_linewidth: unknown

height: unknown
 in_layout: bool
 joinstyle: ``JoinStyle`` or `{'miter', 'round', 'bevel'}`
 label: object
 linestyle or ls: `{'-', '--', '-.', ':', ''}`, (offset, on-off-seq), ...}
 linewidth or lw: float or None
 mouseover: bool
 path_effects: list of ``AbstractPathEffect``
 picker: None or bool or float or callable
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 transform: ``~matplotlib.transforms.Transform``
 url: str
 visible: bool
 width: unknown
 x: unknown
 xy: (float, float)
 y: unknown
 zorder: float

See Also

axvspan : Add a vertical span across the Axes.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.axhspan``.

matplotlib.pylab.axis

```
axis(arg: 'tuple[float, float, float, float] | bool | str | None' = None, /, *, emit:
'bool' = True, **kwargs) -> 'tuple[float, float, float, float]'
```

Convenience method to get or set some axis properties.

Call signatures::

```

xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)

```

Parameters

xmin, xmax, ymin, ymax : float, optional

The axis limits to be set. This can also be achieved using ::

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

option : bool or str

If a bool, turns axis lines and labels on or off. If a string,

possible values are:

```
=====
Value Description
=====
```

```
'off' or `False` Hide all axis decorations, i.e. axis labels, spines,
tick marks, tick labels, and grid lines.
This is the same as `~.Axes.set_axis_off()`.
'on' or `True` Do not hide all axis decorations, i.e. axis labels, spines,
tick marks, tick labels, and grid lines.
This is the same as `~.Axes.set_axis_on()`.
'equal' Set equal scaling (i.e., make circles circular) by
changing the axis limits. This is the same as
``ax.set_aspect('equal', adjustable='datalim')``.
Explicit data limits may not be respected in this case.
'scaled' Set equal scaling (i.e., make circles circular) by
changing dimensions of the plot box. This is the same as
``ax.set_aspect('equal', adjustable='box', anchor='C')``.
Additionally, further autoscaling will be disabled.
'tight' Set limits just large enough to show all data, then
disable further autoscaling.
'auto' Automatic scaling (fill plot box with data).
'image' 'scaled' with axis limits equal to data limits.
'square' Square plot; similar to 'scaled', but initially forcing
``xmax-xmin == ymax-ymin``.
=====
```

emit : bool, default: True
Whether observers are notified of the axis limit change.
This option is passed on to `~.Axes.set_xlim`` and
`~.Axes.set_ylim``.

Returns

```
-----
xmin, xmax, ymin, ymax : float
The axis limits.
```

See Also

```
-----
matplotlib.axes.Axes.set_xlim
matplotlib.axes.Axes.set_ylim
```

Notes

```
-----
.. note::
```

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.axis``.

For 3D Axes, this method additionally takes `*zmin*`, `*zmax*` as parameters and likewise returns them.

matplotlib.pyplot.axline

```
axline(xy1: 'tuple[float, float]', xy2: 'tuple[float, float] | None' = None, *, slope:
'float | None' = None, **kwargs) -> 'AxLine'
```

Add an infinitely long straight line.

The line can be defined either by two points **xy1** and **xy2**, or by one point **xy1** and a **slope**.

This draws a straight line "on the screen", regardless of the x and y scales, and is thus also suitable for drawing exponential decays in semilog plots, power laws in loglog plots, etc. However, **slope** should only be used with linear scales; It has no clear meaning for all other scales, and thus the behavior is undefined. Please specify the line using the points **xy1**, **xy2** for non-linear scales.

The **transform** keyword argument only applies to the points **xy1**, **xy2**. The **slope** (if given) is always in data coordinates. This can be used e.g. with `ax.transAxes` for drawing grid lines with a fixed slope.

Parameters

xy1, xy2 : (float, float)

Points for the line to pass through.

Either **xy2** or **slope** has to be given.

slope : float, optional

The slope of the line. Either **xy2** or **slope** has to be given.

Returns

`.AxLine`

Other Parameters

***kwargs*

Valid kwargs are `.Line2D` properties

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None

animated: bool

antialiased or *aa*: bool

clip_box: `~matplotlib.transforms.BboxBase` or None

clip_on: bool

clip_path: Patch or (Path, Transform) or None

color or *c*: `:mpltype:color`

dash_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}

dash_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}

dashes: sequence of floats (on/off ink in points) or (None, None)

data: (2, N) array or two 1D arrays

drawstyle or *ds*: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`

fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}

gapcolor: `:mpltype:color` or None

gid: str

in_layout: bool
 label: object
 linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
 linewidth or lw: float
 marker: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``
 markeredgewidth or mec: `:mpltype:`color``
 markeredgewidth or mew: float
 markerfacecolor or mfc: `:mpltype:`color``
 markerfacecoloralt or mfcalt: `:mpltype:`color``
 markersize or ms: float
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]
 mouseover: bool
 path_effects: list of `~.AbstractPathEffect``
 picker: float or callable[[Artist, Event], tuple[bool, dict]]
 pickradius: float
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 solid_capstyle: `~.CapStyle`` or {'butt', 'projecting', 'round'}
 solid_joinstyle: `~.JoinStyle`` or {'miter', 'round', 'bevel'}
 transform: unknown
 url: str
 visible: bool
 xdata: 1D array
 ydata: 1D array
 zorder: float

See Also

 axhline : for horizontal lines
 axvline : for vertical lines

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for `~.axes.Axes.axline``.

Examples

Draw a thick red line passing through (0, 0) and (1, 1)::

```
>>> axline((0, 0), (1, 1), linewidth=4, color='r')
```

matplotlib.pyplot.axvline

```
axvline(x: 'float' = 0, ymin: 'float' = 0, ymax: 'float' = 1, **kwargs) -> 'Line2D'
```

Add a vertical line spanning the whole or fraction of the Axes.

Note: If you want to set y-limits in data coordinates, use `~.Axes.vlines`` instead.

Parameters

x : float, default: 0

x position in :ref:`data coordinates <coordinate-systems>`.

ymin : float, default: 0

The start y-position in :ref:`axes coordinates <coordinate-systems>`.

Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

ymax : float, default: 1

The end y-position in :ref:`axes coordinates <coordinate-systems>`.

Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

Returns

`~matplotlib.lines.Line2D`

A `.Line2D` specified via two points `((x, ymin))`, `((x, ymax))`.

Its transform is set such that `*x*` is in

:ref:`data coordinates <coordinate-systems>` and `*y*` is in

:ref:`axes coordinates <coordinate-systems>`.

This is still a generic line and the vertical character is only realized through using identical `*x*` values for both points. Thus, if you want to change the `*x*` value later, you have to provide two values `line.set_xdata([3, 3])`.

Other Parameters

****kwargs**

Valid keyword arguments are `.Line2D` properties, except for 'transform':

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:color`

`dash_capstyle`: `.CapStyle` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: `.JoinStyle` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:color` or None

`gid`: str

`in_layout`: bool

`label`: object

linestyle or ls: {'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}
 linewidth or lw: float
 marker: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``
 markeredgewidth or mec: `:mpltype:`color``
 markeredgewidth or mew: float
 markerfacecolor or mfc: `:mpltype:`color``
 markerfacecoloralt or mfcalt: `:mpltype:`color``
 markersize or ms: float
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]
 mouseover: bool
 path_effects: list of `~.AbstractPathEffect``
 picker: float or callable[[Artist, Event], tuple[bool, dict]]
 pickradius: float
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 solid_capstyle: `~.CapStyle`` or {'butt', 'projecting', 'round'}
 solid_joinstyle: `~.JoinStyle`` or {'miter', 'round', 'bevel'}
 transform: unknown
 url: str
 visible: bool
 xdata: 1D array
 ydata: 1D array
 zorder: float

See Also

 vlines : Add vertical lines in data coordinates.
 axvspan : Add a vertical span (rectangle) across the axis.
 axline : Add a line with an arbitrary slope.

Notes

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for `~.axes.Axes.axvline``.

Examples

* draw a thick red vline at `*x* = 0` that spans the yrange::

```
>>> axvline(linewidth=4, color='r')
```

* draw a default vline at `*x* = 1` that spans the yrange::

```
>>> axvline(x=1)
```

* draw a default vline at `*x* = .5` that spans the middle half of the yrange::

```
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

```
axvspan(xmin: 'float', xmax: 'float', ymin: 'float' = 0, ymax: 'float' = 1, **kwargs)
-> 'Rectangle'
```

Add a vertical span (rectangle) across the Axes.

The rectangle spans from `*xmin*` to `*xmax*` horizontally, and, by default, the whole y-axis vertically. The y-span can be set using `*ymin*` (default: 0) and `*ymax*` (default: 1) which are in axis units; e.g. ```ymin = 0.5``` always refers to the middle of the y-axis regardless of the limits set by `~.Axes.set_ylim``.

Parameters

`xmin` : float

Lower x-coordinate of the span, in data units.

`xmax` : float

Upper x-coordinate of the span, in data units.

`ymin` : float, default: 0

Lower y-coordinate of the span, in y-axis units (0-1).

`ymax` : float, default: 1

Upper y-coordinate of the span, in y-axis units (0-1).

Returns

`~matplotlib.patches.Rectangle``

Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

Other Parameters

`**kwargs` : `~matplotlib.patches.Rectangle`` properties

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`angle`: unknown

`animated`: bool

`antialiased` or `aa`: bool or None

`bounds`: (left, bottom, width, height)

`capstyle`: ``.CapStyle`` or `{'butt', 'projecting', 'round'}`

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color`: `:mpltype:`color``

`edgecolor` or `ec`: `:mpltype:`color`` or None

`facecolor` or `fc`: `:mpltype:`color`` or None

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fill`: bool

`gid`: str

`hatch`: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`

`hatch_linewidth`: unknown

`height`: unknown

`in_layout`: bool

`joinstyle`: ``.JoinStyle`` or `{'miter', 'round', 'bevel'}`

label: object
linestyle or ls: {'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}
linewidth or lw: float or None
mouseover: bool
path_effects: list of ``AbstractPathEffect``
picker: None or bool or float or callable
rasterized: bool
sketch_params: (scale: float, length: float, randomness: float)
snap: bool or None
transform: ``~matplotlib.transforms.Transform``
url: str
visible: bool
width: unknown
x: unknown
xy: (float, float)
y: unknown
zorder: float

See Also

axhspan : Add a horizontal span across the Axes.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.axvspan``.

Examples

Draw a vertical, green, translucent rectangle from `x = 1.25` to `x = 1.55` that spans the yrange of the Axes.

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

matplotlib.pyplot.bar

```
bar(x: 'float | ArrayLike', height: 'float | ArrayLike', width: 'float | ArrayLike' = 0.8, bottom: 'float | ArrayLike | None' = None, *, align: "Literal['center', 'edge']" = 'center', data=None, **kwargs) -> 'BarContainer'
```

Make a bar plot.

The bars are positioned at `*x*` with the given `*align*`ment. Their dimensions are given by `*height*` and `*width*`. The vertical baseline is `*bottom*` (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

Parameters

`x` : float or array-like
The x coordinates of the bars. See also `*align*` for the

alignment of the bars to the coordinates.

Bars are often used for categorical data, i.e. string labels below the bars. You can provide a list of strings directly to `*x*`.
```bar(['A', 'B', 'C'], [1, 2, 3])``` is often a shorter and more convenient notation compared to  
```bar(range(3), [1, 2, 3], tick_label=['A', 'B', 'C'])```. They are equivalent as long as the names are unique. The explicit `*tick_label*` notation draws the names in the sequence given. However, when having duplicate values in categorical `*x*` data, these values map to the same numerical x coordinate, and hence the corresponding bars are drawn on top of each other.

`height` : float or array-like
The height(s) of the bars.

Note that if `*bottom*` has units (e.g. `datetime`), `*height*` should be in units that are a difference from the value of `*bottom*` (e.g. `timedelta`).

`width` : float or array-like, default: 0.8
The width(s) of the bars.

Note that if `*x*` has units (e.g. `datetime`), then `*width*` should be in units that are a difference (e.g. `timedelta`) around the `*x*` values.

`bottom` : float or array-like, default: 0
The y coordinate(s) of the bottom side(s) of the bars.

Note that if `*bottom*` has units, then the y-axis will get a Locator and Formatter appropriate for the units (e.g. dates, or categorical).

`align` : {'center', 'edge'}, default: 'center'
Alignment of the bars to the `*x*` coordinates:

- 'center': Center the base on the `*x*` positions.
- 'edge': Align the left edges of the bars with the `*x*` positions.

To align the bars on the right edge pass a negative `*width*` and ```align='edge'```.

Returns

``.BarContainer``
Container with all the bars and optionally errorbars.

Other Parameters

`color` : `:mpltype:`color`` or list of `:mpltype:`color``, optional
The colors of the bar faces. This is an alias for `*facecolor*`.
If both are given, `*facecolor*` takes precedence.

`facecolor` : `:mpltype:`color`` or list of `:mpltype:`color``, optional
The colors of the bar faces.
If both `*color*` and `*facecolor*` are given, `*facecolor*` takes precedence.

`edgecolor` : `mpltype:color` or list of `mpltype:color`, optional
The colors of the bar edges.

`linewidth` : float or array-like, optional
Width of the bar edge(s). If 0, don't draw edges.

`tick_label` : str or list of str, optional
The tick labels of the bars.
Default: None (Use default numeric labels.)

`label` : str or list of str, optional
A single label is attached to the resulting `.BarContainer` as a label for the whole dataset.
If a list is provided, it must be the same length as `*x*` and labels the individual bars. Repeated labels are not de-duplicated and will cause repeated label entries, so this is best used when bars also differ in style (e.g., by passing a list to `*color*`.)

`xerr`, `yerr` : float or array-like of shape(N,) or shape(2, N), optional
If not `*None*`, add horizontal / vertical errorbars to the bar tips.
The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- `*None*`: No errorbar. (Default)

See `:doc:/gallery/statistics/errorbar_features` for an example on the usage of `*xerr*` and `*yerr*`.

`ecolor` : `mpltype:color` or list of `mpltype:color`, default: 'black'
The line color of the errorbars.

`capsize` : float, default: `:rc:errorbar.capsize`
The length of the error bar caps in points.

`error_kw` : dict, optional
Dictionary of keyword arguments to be passed to the `~.Axes.errorbar` method. Values of `*ecolor*` or `*capsize*` defined here take precedence over the independent keyword arguments.

`log` : bool, default: False
If `*True*`, set the y-axis to be log scale.

`data` : indexable object, optional
If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

`**kwargs` : `.Rectangle` properties

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None
 angle: unknown
 animated: bool
 antialiased or aa: bool or None
 bounds: (left, bottom, width, height)
 capstyle: `CapStyle` or `{'butt', 'projecting', 'round'}`
 clip_box: `~matplotlib.transforms.BboxBase` or None
 clip_on: bool
 clip_path: Patch or (Path, Transform) or None
 color: `mpltype:color`
 edgecolor or ec: `mpltype:color` or None
 facecolor or fc: `mpltype:color` or None
 figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`
 fill: bool
 gid: str
 hatch: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`
 hatch_linewidth: unknown
 height: unknown
 in_layout: bool
 joinstyle: `JoinStyle` or `{'miter', 'round', 'bevel'}`
 label: object
 linestyle or ls: `{'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}`
 linewidth or lw: float or None
 mouseover: bool
 path_effects: list of `.AbstractPathEffect`
 picker: None or bool or float or callable
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 transform: `~matplotlib.transforms.Transform`
 url: str
 visible: bool
 width: unknown
 x: unknown
 xy: (float, float)
 y: unknown
 zorder: float

See Also

barh : Plot a horizontal bar plot.

Notes

.. note::

This is the `:ref:pyplot wrapper <pyplot_interface>` for `.axes.Axes.bar`.

Stacked bars can be achieved by passing individual **bottom** values per bar. See `:doc:/gallery/lines_bars_and_markers/bar_stacked`.

matplotlib.pyplot.bar_label

```
bar_label(container: 'BarContainer', labels: 'ArrayLike | None' = None, *, fmt: 'str | Callable[[float], str]' = '%g', label_type: "Literal['center', 'edge']" = 'edge', padding: 'float' = 0, **kwargs) -> 'list[Annotation]'
```

Label a bar plot.

Adds labels to bars in the given ``BarContainer``.

You may need to adjust the axis limits to fit the labels.

Parameters

`container : `BarContainer``

Container with all the bars and optionally errorbars, likely returned from ``bar`` or ``barh``.

`labels : array-like, optional`

A list of label texts, that should be displayed. If not given, the label texts will be the data values formatted with `*fmt*`.

`fmt : str or callable, default: '%g'`

An unnamed %-style or {}-style format string for the label or a function to call with the value as the first argument.

When `*fmt*` is a string and can be interpreted in both formats, %-style takes precedence over {}-style.

`.. versionadded:: 3.7`

Support for {}-style format string and callables.

`label_type : {'edge', 'center'}, default: 'edge'`

The label type. Possible values:

- 'edge': label placed at the end-point of the bar segment, and the value displayed will be the position of that end-point.

- 'center': label placed in the center of the bar segment, and the value displayed will be the length of that segment.

(useful for stacked bars, i.e.,

`:doc:`gallery/lines_bars_and_markers/bar_label_demo``)

`padding : float, default: 0`

Distance of label from the end of the bar, in points.

`**kwargs`

Any remaining keyword arguments are passed through to

``Axes.annotate``. The alignment parameters (

`*horizontalalignment* / *ha*`, `*verticalalignment* / *va*`) are

not supported because the labels are automatically aligned to the bars.

Returns

list of ``Annotation``

A list of ``Annotation`` instances for the labels.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.bar_label``.

matplotlib.pyplot.barbs

```
barbs(*args, data=None, **kwargs) -> 'Barbs'
```

Plot a 2D field of wind barbs.

Call signature::

```
barbs([X, Y], U, V, [C], /, **kwargs)
```

Where `*X*`, `*Y*` define the barb locations, `*U*`, `*V*` define the barb directions, and `*C*` optionally sets the color.

The arguments `*X*`, `*Y*`, `*U*`, `*V*`, `*C*` are positional-only and may be 1D or 2D. `*U*`, `*V*`, `*C*` may be masked arrays, but masked `*X*`, `*Y*` are not supported at present.

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below::

```
: /\
: /\
: /\
: /\
: -----
```

The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

See also https://en.wikipedia.org/wiki/Wind_barb.

Parameters

X, Y : 1D or 2D array-like, optional

The x and y coordinates of the barb locations. See `*pivot*` for how the barbs are drawn to the x, y positions.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of `*U*` and `*V*`.

If **X** and **Y** are 1D but **U**, **V** are 2D, **X**, **Y** are expanded to 2D using ``X, Y = np.meshgrid(X, Y)``. In this case ``len(X)`` and ``len(Y)`` must match the column and row dimensions of **U** and **V**.

U, V : 1D or 2D array-like

The x and y components of the barb shaft.

C : 1D or 2D array-like, optional

Numeric data that defines the barb colors by colormapping via **norm** and **cmap**.

This does not support explicit colors. If you want to set colors directly, use **barbcolor** instead.

length : float, default: 7

Length of the barb in points; the other parts of the barb are scaled against this.

pivot : {'tip', 'middle'} or float, default: 'tip'

The part of the arrow that is anchored to the **X**, **Y** grid. The barb rotates about this point. This can also be a number, which shifts the start of the barb that many points away from grid point.

barbcolor : :mpltype:`color` or color sequence

The color of all parts of the barb except for the flags. This parameter is analogous to the **edgecolor** parameter for polygons, which can be used instead. However this parameter will override *facecolor*.

flagcolor : :mpltype:`color` or color sequence

The color of any flags on the barb. This parameter is analogous to the **facecolor** parameter for polygons, which can be used instead. However, this parameter will override *facecolor*. If this is not set (and **C** has not either) then **flagcolor** will be set to match **barbcolor** so that the barb has a uniform color. If **C** has been set, **flagcolor** has no effect.

sizes : dict, optional

A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

fill_empty : bool, default: False

Whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, the center is transparent.

rounding : bool, default: True

Whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple.

barb_increments : dict, optional

A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

flip_barb : bool or array-like of bool, default: False

Whether the lines and flags should point opposite to normal.

Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere).

A single value is applied to all barbs. Individual barbs can be flipped by passing a bool array of the same size as *U* and *V*.

Returns

barbs : `~matplotlib.quiver.Barbs``

Other Parameters

data : indexable object, optional
If given, all parameters also accept a string ``s``, which is interpreted as `data[s]` if ``s`` is a key in `data`.

****kwargs**

The barbs can further be customized using `.PolyCollection`` keyword arguments:

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: array-like or float or None

animated: bool

antialiased or aa or antialiaseds: bool or list of bools

array: array-like or None

capstyle: `.CapStyle`` or {'butt', 'projecting', 'round'}

clim: (vmin: float, vmax: float)

clip_box: `~matplotlib.transforms.BboxBase`` or None

clip_on: bool

clip_path: Patch or (Path, Transform) or None

cmap: `.Colormap`` or str or None

color: :mpltype:`color` or list of RGBA tuples

edgecolor or ec or edgcolors: :mpltype:`color` or list of :mpltype:`color` or 'face'

facecolor or facecolors or fc: :mpltype:`color` or list of :mpltype:`color`

figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

gid: str

hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}

hatch_linewidth: unknown

in_layout: bool

joinstyle: `.JoinStyle`` or {'miter', 'round', 'bevel'}

label: object
 linestyle or dashes or linestyles or ls: str or tuple or list thereof
 linewidth or linewidths or lw: float or list of floats
 mouseover: bool
 norm: ``_.Normalize`` or str or None
 offset_transform or transOffset: ``_.Transform``
 offsets: (N, 2) or (2,) array-like
 path_effects: list of ``_.AbstractPathEffect``
 paths: list of array-like
 picker: None or bool or float or callable
 pickradius: float
 rasterized: bool
 sizes: ``numpy.ndarray`` or None
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 transform: ``~matplotlib.transforms.Transform``
 url: str
 urls: list of str or None
 verts: list of array-like
 verts_and_codes: unknown
 visible: bool
 zorder: float

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``_.axes.Axes.barbs``.

matplotlib.pyplot.barh

```
barh(y: 'float | ArrayLike', width: 'float | ArrayLike', height: 'float | ArrayLike' =
0.8, left: 'float | ArrayLike | None' = None, *, align: "Literal['center', 'edge']" =
'center', data=None, **kwargs) -> 'BarContainer'
```

Make a horizontal bar plot.

The bars are positioned at **y** with the given **align**ment. Their dimensions are given by **width** and **height**. The horizontal baseline is **left** (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

Parameters

y : float or array-like

The y coordinates of the bars. See also **align** for the alignment of the bars to the coordinates.

Bars are often used for categorical data, i.e. string labels below the bars. You can provide a list of strings directly to **y**.
```barh(['A', 'B', 'C'], [1, 2, 3])``` is often a shorter and more convenient notation compared to

`barh(range(3), [1, 2, 3], tick_label=['A', 'B', 'C'])`. They are equivalent as long as the names are unique. The explicit `tick_label` notation draws the names in the sequence given. However, when having duplicate values in categorical `y` data, these values map to the same numerical `y` coordinate, and hence the corresponding bars are drawn on top of each other.

`width` : float or array-like  
The width(s) of the bars.

Note that if `left` has units (e.g. `datetime`), `width` should be in units that are a difference from the value of `left` (e.g. `timedelta`).

`height` : float or array-like, default: 0.8  
The heights of the bars.

Note that if `y` has units (e.g. `datetime`), then `height` should be in units that are a difference (e.g. `timedelta`) around the `y` values.

`left` : float or array-like, default: 0  
The x coordinates of the left side(s) of the bars.

Note that if `left` has units, then the x-axis will get a `Locator` and `Formatter` appropriate for the units (e.g. `dates`, or `categorical`).

`align` : {'center', 'edge'}, default: 'center'  
Alignment of the base to the `y` coordinates:

- 'center': Center the bars on the `y` positions.
- 'edge': Align the bottom edges of the bars with the `y` positions.

To align the bars on the top edge pass a negative `height` and `align='edge'`.

#### Returns

-----  
`.BarContainer``  
Container with all the bars and optionally errorbars.

#### Other Parameters

-----  
`color` : `mpltype:color`` or list of `mpltype:color``, optional  
The colors of the bar faces.

`edgecolor` : `mpltype:color`` or list of `mpltype:color``, optional  
The colors of the bar edges.

`linewidth` : float or array-like, optional  
Width of the bar edge(s). If 0, don't draw edges.

`tick_label` : str or list of str, optional  
The tick labels of the bars.  
Default: None (Use default numeric labels.)

label : str or list of str, optional

A single label is attached to the resulting `.BarContainer`` as a label for the whole dataset.

If a list is provided, it must be the same length as `*y*` and labels the individual bars. Repeated labels are not de-duplicated and will cause repeated label entries, so this is best used when bars also differ in style (e.g., by passing a list to `*color*`.)

xerr, yerr : float or array-like of shape(N,) or shape(2, N), optional

If not `*None*`, add horizontal / vertical errorbars to the bar tips.

The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- `*None*`: No errorbar. (default)

See `:doc:`gallery/statistics/errorbar_features`` for an example on the usage of `*xerr*` and `*yerr*`.

ecolor : `:mpltype:`color`` or list of `:mpltype:`color``, default: 'black'

The line color of the errorbars.

capsize : float, default: `:rc:`errorbar.capsize``

The length of the error bar caps in points.

error\_kw : dict, optional

Dictionary of keyword arguments to be passed to the `~.Axes.errorbar`` method. Values of `*ecolor*` or `*capsize*` defined here take precedence over the independent keyword arguments.

log : bool, default: False

If ```True```, set the x-axis to be log scale.

data : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

**\*\*kwargs** : `.Rectangle`` properties

Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None

angle: unknown

animated: bool

antialiased or aa: bool or None

bounds: (left, bottom, width, height)

capstyle: `.CapStyle`` or {'butt', 'projecting', 'round'}

clip\_box: `~matplotlib.transforms.BboxBase`` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

color: `:mpltype:`color``

edgecolor or ec: :mpltype:`color` or None  
 facecolor or fc: :mpltype:`color` or None  
 figure: ~matplotlib.figure.Figure` or ~matplotlib.figure.SubFigure`  
 fill: bool  
 gid: str  
 hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '\*'}  
 hatch\_linewidth: unknown  
 height: unknown  
 in\_layout: bool  
 joinstyle: ~.JoinStyle` or {'miter', 'round', 'bevel'}  
 label: object  
 linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}  
 linewidth or lw: float or None  
 mouseover: bool  
 path\_effects: list of ~.AbstractPathEffect`  
 picker: None or bool or float or callable  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 transform: ~matplotlib.transforms.Transform`  
 url: str  
 visible: bool  
 width: unknown  
 x: unknown  
 xy: (float, float)  
 y: unknown  
 zorder: float

See Also

-----  
 bar : Plot a vertical bar plot.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ~.axes.Axes.barh`.

Stacked bars can be achieved by passing individual \*left\* values per bar. See :doc:`gallery/lines\_bars\_and\_markers/horizontal\_barchart\_distribution`.

## matplotlib.pyplot.bone

```
bone() -> 'None'
```

Set the colormap to 'bone'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

## matplotlib.pyplot.box

```
box(on: 'bool | None' = None) -> 'None'
```

Turn the Axes box on or off on the current Axes.

#### Parameters

-----

on : bool or None

The new `~matplotlib.axes.Axes` box` state. If ```None```, toggle the state.

#### See Also

-----

`:meth:`matplotlib.axes.Axes.set_frame_on``

`:meth:`matplotlib.axes.Axes.get_frame_on``

## matplotlib.pyplot.boxplot

```
boxplot(x: 'ArrayLike | Sequence[ArrayLike]', *, notch: 'bool | None' = None, sym:
'str | None' = None, vert: 'bool | None' = None, orientation: "Literal['vertical',
'horizontal']" = 'vertical', whis: 'float | tuple[float, float] | None' = None,
positions: 'ArrayLike | None' = None, widths: 'float | ArrayLike | None' = None,
patch_artist: 'bool | None' = None, bootstrap: 'int | None' = None, usermedians:
'ArrayLike | None' = None, conf_intervals: 'ArrayLike | None' = None, meanline: 'bool
| None' = None, showmeans: 'bool | None' = None, showcaps: 'bool | None' = None,
showbox: 'bool | None' = None, showfliers: 'bool | None' = None, boxprops: 'dict[str,
Any] | None' = None, tick_labels: 'Sequence[str] | None' = None, flierprops:
'dict[str, Any] | None' = None, medianprops: 'dict[str, Any] | None' = None,
meanprops: 'dict[str, Any] | None' = None, capprops: 'dict[str, Any] | None' = None,
whiskerprops: 'dict[str, Any] | None' = None, manage_ticks: 'bool' = True, autorange:
'bool' = False, zorder: 'float | None' = None, capwidths: 'float | ArrayLike | None' =
None, label: 'Sequence[str] | None' = None, data=None) -> 'dict[str, Any]'
```

Draw a box and whisker plot.

The box extends from the first quartile (Q1) to the third quartile (Q3) of the data, with a line at the median.

The whiskers extend from the box to the farthest data point lying within 1.5x the inter-quartile range (IQR) from the box.

Flier points are those past the end of the whiskers.

See [https://en.wikipedia.org/wiki/Box\\_plot](https://en.wikipedia.org/wiki/Box_plot) for reference.

.. code-block:: none

Q1-1.5IQR Q1 median Q3 Q3+1.5IQR

```
|----:----|
o |-----| : |-----| o o
|----:----|
flier <-----> fliers
IQR
```

#### Parameters

-----

x : Array or a sequence of vectors.

The input data. If a 2D array, a boxplot is drawn for each column in `*x*`. If a sequence of 1D arrays, a boxplot is drawn for each array in `*x*`.

notch : bool, default: :rc:`boxplot.notch`  
Whether to draw a notched boxplot (`True`), or a rectangular boxplot (`False`). The notches represent the confidence interval (CI) around the median. The documentation for *\*bootstrap\** describes how the locations of the notches are computed by default, but their locations may also be overridden by setting the *\*conf\_intervals\** parameter.

.. note::

In cases where the values of the CI are less than the lower quartile or greater than the upper quartile, the notches will extend beyond the box, giving it a distinctive "flipped" appearance. This is expected behavior and consistent with other statistical visualization packages.

sym : str, optional  
The default symbol for flier points. An empty string (``) hides the fliers. If *\*None\**, then the fliers default to *'b+'*. More control is provided by the *\*flierprops\** parameter.

vert : bool, optional  
.. deprecated:: 3.11  
Use *\*orientation\** instead.

This is a pending deprecation for 3.10, with full deprecation in 3.11 and removal in 3.13.  
If this is given during the deprecation period, it overrides the *\*orientation\** parameter.

If *True*, plots the boxes vertically.  
If *False*, plots the boxes horizontally.

orientation : {'vertical', 'horizontal'}, default: 'vertical'  
If *'horizontal'*, plots the boxes horizontally.  
Otherwise, plots the boxes vertically.

.. versionadded:: 3.10

whis : float or (float, float), default: 1.5  
The position of the whiskers.

If a float, the lower whisker is at the lowest datum above  $Q1 - whis(Q3 - Q1)$ , and the upper whisker at the highest datum below  $Q3 + whis(Q3 - Q1)$ , where *Q1* and *Q3* are the first and third quartiles. The default value of  $whis = 1.5$  corresponds to Tukey's original definition of boxplots.

If a pair of floats, they indicate the percentiles at which to draw the whiskers (e.g., (5, 95)). In particular, setting this to (0, 100) results in whiskers covering the whole range of the data.

In the edge case where  $Q1 == Q3$ , *\*whis\** is automatically set to (0, 100) (cover the whole range of the data) if *\*autorange\** is

True.

Beyond the whiskers, data are considered outliers and are plotted as individual points.

`bootstrap` : int, optional

Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If `*bootstrap*` is None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, `bootstrap` specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.

`usermedians` : 1D array-like, optional

A 1D array-like of length `len(x)`. Each entry that is not `None` forces the value of the median for the corresponding dataset. For entries that are `None`, the medians are computed by Matplotlib as normal.

`conf_intervals` : array-like, optional

A 2D array-like of shape `(len(x), 2)`. Each entry that is not None forces the location of the corresponding notch (which is only drawn if `*notch*` is `True`). For entries that are `None`, the notches are computed by the method specified by the other parameters (e.g., `*bootstrap*`).

`positions` : array-like, optional

The positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to `range(1, N+1)` where N is the number of boxes to be drawn.

`widths` : float or array-like

The widths of the boxes. The default is 0.5, or `0.15*(distance between extreme positions)`, if that is smaller.

`patch_artist` : bool, default: `rc['boxplot.patchartist']`

If `False` produces boxes with the Line2D artist. Otherwise, boxes are drawn with Patch artists.

`tick_labels` : list of str, optional

The tick labels of each boxplot.

Ticks are always placed at the box `*positions*`. If `*tick_labels*` is given, the ticks are labelled accordingly. Otherwise, they keep their numeric values.

.. versionchanged:: 3.9

Renamed from `*labels*`, which is deprecated since 3.9 and will be removed in 3.11.

`manage_ticks` : bool, default: True

If True, the tick locations and labels will be adjusted to match the boxplot positions.

autorange : bool, default: False

When `True` and the data are distributed such that the 25th and 75th percentiles are equal, `*whis*` is set to (0, 100) such that the whisker ends are at the minimum and maximum of the data.

meanline : bool, default: `rc:boxplot.meanline`

If `True` (and `*showmeans*` is `True`), will try to render the mean as a line spanning the full width of the box according to `*meanprops*` (see below). Not recommended if `*shownotches*` is also `True`. Otherwise, means will be shown as points.

zorder : float, default: `Line2D.zorder = 2`

The zorder of the boxplot.

## Returns

-----

dict

A dictionary mapping each component of the boxplot to a list of the `.Line2D` instances created. That dictionary has the following keys (assuming vertical boxplots):

- `boxes`: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: the vertical lines extending to the most extreme, non-outlier data points.
- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

## Other Parameters

-----

showcaps : bool, default: `rc:boxplot.showcaps`

Show the caps on the ends of whiskers.

showbox : bool, default: `rc:boxplot.showbox`

Show the central box.

showfliers : bool, default: `rc:boxplot.showfliers`

Show the outliers beyond the caps.

showmeans : bool, default: `rc:boxplot.showmeans`

Show the arithmetic means.

capprops : dict, default: None

The style of the caps.

capwidths : float or array, default: None

The widths of the caps.

boxprops : dict, default: None

The style of the box.

whiskerprops : dict, default: None  
 The style of the whiskers.  
 flierprops : dict, default: None  
 The style of the fliers.  
 medianprops : dict, default: None  
 The style of the median.  
 meanprops : dict, default: None  
 The style of the mean.  
 label : str or list of str, optional  
 Legend labels. Use a single string when all boxes have the same style and you only want a single legend entry for them. Use a list of strings to label all boxes individually. To be distinguishable, the boxes should be styled individually, which is currently only possible by modifying the returned artists, see e.g. :doc:`/gallery/statistics/boxplot\_demo`.

In the case of a single string, the legend entry will technically be associated with the first box only. By default, the legend will show the median line (`result["medians"]`); if `patch_artist` is True, the legend will show the box `.Patch` artists (`result["boxes"]`) instead.

.. versionadded:: 3.9

data : indexable object, optional  
 If given, all parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`.

See Also

-----  
`.Axes.bxp` : Draw a boxplot from pre-computed statistics.  
`violinplot` : Draw an estimate of the probability density function.

Notes

-----  
 .. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.boxplot`.

## matplotlib.pyplot.broken\_barh

```
broken_barh(xranges: 'Sequence[tuple[float, float]]', yrange: 'tuple[float, float]',
 *, data=None, **kwargs) -> 'PolyCollection'
```

Plot a horizontal sequence of rectangles.

A rectangle is drawn for each element of `xranges`. All rectangles have the same vertical position and size defined by `yrange`.

Parameters

-----  
`xranges` : sequence of tuples (`xmin`, `xwidth`)  
 The x-positions and extents of the rectangles. For each tuple (`xmin`, `xwidth`) a rectangle is drawn from `xmin` to `xmin` + `xwidth`.  
`yrange` : (`ymin`, `yheight`)

The y-position and extent for all the rectangles.

## Returns

-----  
`~.collections.PolyCollection``

## Other Parameters

-----  
data : indexable object, optional  
If given, all parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``.  
\*\*kwargs : `~.PolyCollection`` properties

Each \*kwargs\* can be either a single argument applying to all rectangles, e.g.::

```
facecolors='black'
```

or a sequence of arguments over which is cycled, e.g.::

```
facecolors=('black', 'blue')
```

would create interleaving black and blue rectangles.

Supported keywords:

## Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: array-like or float or None

animated: bool

antialiased or aa or antialiaseds: bool or list of bools

array: array-like or None

capstyle: `~.CapStyle`` or {'butt', 'projecting', 'round'}

clim: (vmin: float, vmax: float)

clip\_box: `~matplotlib.transforms.BboxBase`` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

cmap: `~.Colormap`` or str or None

color: :mpltype:`color` or list of RGBA tuples

edgecolor or ec or edgecolors: :mpltype:`color` or list of :mpltype:`color` or 'face'

facecolor or facecolors or fc: :mpltype:`color` or list of :mpltype:`color`

figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

gid: str

hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '\*'}

hatch\_linewidth: unknown

in\_layout: bool

joinstyle: `~.JoinStyle`` or {'miter', 'round', 'bevel'}

label: object

linestyle or dashes or linestyles or ls: str or tuple or list thereof

linewidth or linewidths or lw: float or list of floats

mouseover: bool

norm: `~.Normalize`` or str or None

offset\_transform or transOffset: `~.Transform``

offsets: (N, 2) or (2,) array-like

path\_effects: list of ``AbstractPathEffect``  
 paths: list of array-like  
 picker: None or bool or float or callable  
 pickradius: float  
 rasterized: bool  
 sizes: ``numpy.ndarray`` or None  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 transform: ``~matplotlib.transforms.Transform``  
 url: str  
 urls: list of str or None  
 verts: list of array-like  
 verts\_and\_codes: unknown  
 visible: bool  
 zorder: float

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``axes.Axes.broken_barh``.

## matplotlib.pyplot.cla

```
cla() -> 'None'
```

Clear the current Axes.

## matplotlib.pyplot.clabel

```
clabel(CS: 'ContourSet', levels: 'ArrayLike | None' = None, **kwargs) -> 'list[Text]'
```

Label a contour plot.

Adds labels to line contours in given ``ContourSet``.

Parameters

-----

CS : ``ContourSet`` instance

Line contours to label.

levels : array-like, optional

A list of level values, that should be labeled. The list must be a subset of ``CS.levels``. If not given, all levels are labeled.

**\*\*kwargs**

All other parameters are documented in ``~.ContourLabeler.clabel``.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.clabel``.

## matplotlib.pyplot.clf

```
clf() -> 'None'
```

Clear the current figure.

## matplotlib.pyplot.clim

```
clim(vmin: 'float | None' = None, vmax: 'float | None' = None) -> 'None'
```

Set the color limits of the current image.

If either `*vmin*` or `*vmax*` is `None`, the image min/max respectively will be used for color scaling.

If you want to set the clim of multiple images, use `~.ScalarMappable.set_clim`` on every image, for example::

```
for im in gca().get_images():
 im.set_clim(0, 0.5)
```

## matplotlib.pyplot.close

```
close(fig: "None | int | str | Figure | Literal['all']" = None) -> 'None'
```

Close a figure window, and unregister it from pyplot.

Parameters

-----

`fig` : `None` or `int` or `str` or `.Figure``

The figure to close. There are a number of ways to specify this:

- `*None*`: the current figure
- `.Figure``: the given `.Figure`` instance
- ```int```: a figure number
- ```str```: a figure name
- `'all'`: all figures

Notes

-----

pyplot maintains a reference to figures created with `figure()`. When work on the figure is completed, it should be closed, i.e. deregistered from pyplot, to free its memory (see also `:rc:figure.max_open_warning`). Closing a figure window created by `show()` automatically deregisters the figure. For all other use cases, most prominently `savefig()` without `show()`, the figure must be deregistered explicitly using `close()`.

## matplotlib.pyplot.cohere

```
cohere(x: 'ArrayLike', y: 'ArrayLike', *, NFFT: 'int' = 256, Fs: 'float' = 2, Fc:
'int' = 0, detrend: "Literal['none', 'mean', 'linear'] | Callable[[ArrayLike],
ArrayLike]" = , window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike' = , noverlap:
'int' = 0, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided',
'twosided']" = 'default', scale_by_freq: 'bool | None' = None, data=None, **kwargs) ->
'tuple[np.ndarray, np.ndarray]'
```

Plot the coherence between *x* and *y*.

Coherence is the normalized cross spectral density:

.. math::

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

#### Parameters

-----

**Fs** : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *\*freqs\**, in cycles per time unit.

**window** : callable or ndarray, default: ``.window_hanning``

A function or a vector of length *\*NFFT\**. To create window vectors see ``.window_hanning``, ``.window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides** : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad\_to** : int, optional

The number of points to which the data segment is padded when performing the FFT. This can be different from *\*NFFT\**, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *\*n\** parameter in the call to `~numpy.fft.fft``. The default is None, which sets *\*pad\_to\** equal to *\*NFFT\**

**NFFT** : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should *\*NOT\** be used to get zero padding, or the scaling of the result will be incorrect; use *\*pad\_to\** for this instead.

**detrend** : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *\*detrend\** parameter is a vector, in Matplotlib it is a function. The `:mod:`~matplotlib.mlab`` module defines ``.detrend_none``, ``.detrend_mean``, and ``.detrend_linear``, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls ``.detrend_none``. 'mean' calls ``.detrend_mean``. 'linear' calls ``.detrend_linear``.

**scale\_by\_freq** : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap : int, default: 0 (no overlap)  
The number of points of overlap between blocks.

Fc : int, default: 0  
The center frequency of \*x\*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

#### Returns

-----  
Cxy : 1-D array  
The coherence vector.

freqs : 1-D array  
The frequencies for the elements in \*Cxy\*.

#### Other Parameters

-----  
data : indexable object, optional  
If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

\*x\*, \*y\*

\*\*kwargs  
Keyword arguments control the `.Line2D`` properties:

#### Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image  
alpha: float or None  
animated: bool  
antialiased or aa: bool  
clip\_box: `~matplotlib.transforms.BboxBase`` or None  
clip\_on: bool  
clip\_path: Patch or (Path, Transform) or None  
color or c: `:mpltype:`color``  
dash\_capstyle: `.CapStyle`` or {'butt', 'projecting', 'round'}  
dash\_joinstyle: `.JoinStyle`` or {'miter', 'round', 'bevel'}  
dashes: sequence of floats (on/off ink in points) or (None, None)  
data: (2, N) array or two 1D arrays  
drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'  
figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``  
fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}  
gapcolor: `:mpltype:`color`` or None  
gid: str  
in\_layout: bool  
label: object  
linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}  
linewidth or lw: float

marker: marker style string, ``~.path.Path`` or ``~.markers.MarkerStyle``  
 markeredgewidth or mec: `:mpltype:`color``  
 markeredgewidth or mew: float  
 markerfacecolor or mfc: `:mpltype:`color``  
 markerfacecoloralt or mfcalt: `:mpltype:`color``  
 markersize or ms: float  
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]  
 mouseover: bool  
 path\_effects: list of ``~.AbstractPathEffect``  
 picker: float or callable[[Artist, Event], tuple[bool, dict]]  
 pickradius: float  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 solid\_capstyle: ``~.CapStyle`` or {'butt', 'projecting', 'round'}  
 solid\_joinstyle: ``~.JoinStyle`` or {'miter', 'round', 'bevel'}  
 transform: unknown  
 url: str  
 visible: bool  
 xdata: 1D array  
 ydata: 1D array  
 zorder: float

#### Notes

-----

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``~.axes.Axes.cohere``.

#### References

-----

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,  
John Wiley & Sons (1986)

## matplotlib.pyplot.colorbar

```
colorbar(mappable: 'ScalarMappable | ColorizingArtist | None' = None, cax:
'matplotlib.axes.Axes | None' = None, ax: 'matplotlib.axes.Axes |
Iterable[matplotlib.axes.Axes] | None' = None, **kwargs) -> 'Colorbar'
```

Add a colorbar to a plot.

#### Parameters

-----

mappable

The ``matplotlib.cm.ScalarMappable`` (i.e., ``~.AxesImage``, ``~.ContourSet``, etc.) described by this colorbar. This argument is mandatory for the ``~.Figure.colorbar`` method but optional for the ``~.pyplot.colorbar`` function, which sets the default to the current image.

Note that one can create a ``~.ScalarMappable`` "on-the-fly" to generate colorbars not attached to a previously drawn artist, e.g.  
::

`fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap), ax=ax)`

`cax` : `~matplotlib.axes.Axes`, optional

Axes into which the colorbar will be drawn. If `None`, then a new Axes is created and the space for it will be stolen from the Axes(s) specified in `*ax*`.

`ax` : `~matplotlib.axes.Axes` or iterable or `numpy.ndarray` of Axes, optional

The one or more parent Axes from which space for a new colorbar Axes will be stolen. This parameter is only used if `*cax*` is not set.

Defaults to the Axes that contains the mappable used to create the colorbar.

`use_gridspec` : bool, optional

If `*cax*` is `None`, a new `*cax*` is created as an instance of Axes. If `*ax*` is positioned with a `subplotspec` and `*use_gridspec*` is `True`, then `*cax*` is also positioned with a `subplotspec`.

#### Returns

-----  
`colorbar` : `~matplotlib.colorbar.Colorbar`

#### Other Parameters

-----

`location` : None or {'left', 'right', 'top', 'bottom'}

The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the `*orientation*` of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the `*orientation*` if it is set (vertical colorbars on the right, horizontal ones at the bottom), or default to 'right' if `*orientation*` is unset.

`orientation` : None or {'vertical', 'horizontal'}

The orientation of the colorbar. It is preferable to set the `*location*` of the colorbar, as that also determines the `*orientation*`; passing incompatible values for `*location*` and `*orientation*` raises an exception.

`fraction` : float, default: 0.15

Fraction of original Axes to use for colorbar.

`shrink` : float, default: 1.0

Fraction by which to multiply the size of the colorbar.

`aspect` : float, default: 20

Ratio of long to short dimensions.

`pad` : float, default: 0.05 if vertical, 0.15 if horizontal

Fraction of original Axes between colorbar and new image Axes.

`anchor` : (float, float), optional

The anchor point of the colorbar Axes.

Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

`panchor` : (float, float), or `*False*`, optional

The anchor point of the colorbar parent Axes. If `*False*`, the parent axes' anchor will be unchanged.

Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

`extend` : {'neither', 'both', 'min', 'max'}

Make pointed end(s) for out-of-range values (unless 'neither'). These are set for a given colormap using the colormap `set_under` and `set_over` methods.

`extendfrac` : {'None', 'auto', length, lengths}

If set to `*None*`, both the minimum and maximum triangular colorbar extensions will have a length of 5% of the interior colorbar length (this is the default setting).

If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when `*spacing*` is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when `*spacing*` is set to 'proportional').

If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.

`extendrect` : bool

If `*False*` the minimum and maximum colorbar extensions will be triangular (the default). If `*True*` the extensions will be rectangular.

`ticks` : None or list of ticks or Locator

If None, ticks are determined automatically from the input.

`format` : None or str or Formatter

If None, `~.ticker.ScalarFormatter`` is used.

Format strings, e.g., ```"%4.2e"``` or ```"{x:.2e}"```, are supported.

An alternative `~.ticker.Formatter`` may be given instead.

`drawedges` : bool

Whether to draw lines at color boundaries.

`label` : str

The label on the colorbar's long axis.

`boundaries, values` : None or a sequence

If unset, the colormap will be displayed on a 0-1 scale.

If sequences, `*values*` must have a length 1 less than `*boundaries*`. For each region delimited by adjacent entries in `*boundaries*`, the color mapped to the corresponding value in `*values*` will be used. The size of each region is determined by the `*spacing*` parameter.

Normally only useful for indexed colors (i.e. ```norm=NoNorm()```) or other unusual circumstances.

`spacing` : {'uniform', 'proportional'}

For discrete colorbars (`~.BoundaryNorm`` or contours), 'uniform' gives each

color the same space; 'proportional' makes the space proportional to the data interval.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `Figure.colorbar``.

If `*mappable*` is a `~.contour.ContourSet``, its `*extend*` kwarg is included automatically.

The `*shrink*` kwarg provides a simple way to scale the colorbar with respect to the Axes. Note that if `*cax*` is specified, it determines the size of the colorbar, and `*shrink*` and `*aspect*` are ignored.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the Axes properties kwargs.

It is known that some vector graphics viewers (svg and pdf) render white gaps between segments of the colorbar. This is due to bugs in the viewers, not Matplotlib. As a workaround, the colorbar can be rendered with overlapping segments::

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However, this has negative consequences in other circumstances, e.g. with semi-transparent images ( $\alpha < 1$ ) and colorbar extensions; therefore, this workaround is not used by default (see issue #1188).

## matplotlib.pyplot.connect

```
connect(s: 'str', func: 'Callable[[Event], Any]') -> 'int'
```

Bind function `*func*` to event `*s*`.

#### Parameters

-----

`s` : str

One of the following events ids:

- 'button\_press\_event'
- 'button\_release\_event'
- 'draw\_event'
- 'key\_press\_event'
- 'key\_release\_event'
- 'motion\_notify\_event'
- 'pick\_event'
- 'resize\_event'
- 'scroll\_event'

- 'figure\_enter\_event',
- 'figure\_leave\_event',
- 'axes\_enter\_event',
- 'axes\_leave\_event'
- 'close\_event'.

func : callable

The callback function to be executed, which must have the signature::

```
def func(event: Event) -> Any
```

For the location events (button and key press/release), if the mouse is over the Axes, the ``inaxes`` attribute of the event will be set to the `~matplotlib.axes.Axes` the event occurs is over, and additionally, the variables ``xdata`` and ``ydata`` attributes will be set to the mouse location in data coordinates. See `.KeyEvent` and `.MouseEvent` for more info.

.. note::

If func is a method, this only stores a weak reference to the method. Thus, the figure does not influence the lifetime of the associated object. Usually, you want to make sure that the object is kept alive throughout the lifetime of the figure by holding a reference to it.

Returns

-----

cid

A connection id that can be used with `.FigureCanvasBase.mpl_disconnect`.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.FigureCanvasBase.mpl_connect`.

Examples

-----

::

```
def on_press(event):
 print('you pressed', event.button, event.xdata, event.ydata)
```

```
cid = canvas.mpl_connect('button_press_event', on_press)
```

## matplotlib.pyplot.contour

```
contour(*args, data=None, **kwargs) -> 'QuadContourSet'
```

Plot contour lines.

Call signature::

```
contour([X, Y,] Z, /, [levels], **kwargs)
```

The arguments `*X*`, `*Y*`, `*Z*` are positional-only.

``~.contour`` and ``~.contourf`` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

#### Parameters

-----

`X`, `Y` : array-like, optional

The coordinates of the values in `*Z*`.

`*X*` and `*Y*` must both be 2D with the same shape as `*Z*` (e.g. created via ``numpy.meshgrid``), or they must both be 1-D such that ``len(X) == N`` is the number of columns in `*Z*` and ``len(Y) == M`` is the number of rows in `*Z*`.

`*X*` and `*Y*` must both be ordered monotonically.

If not given, they are assumed to be integer indices, i.e.

``X = range(N)``, ``Y = range(M)``.

`Z` : (M, N) array-like

The height values over which the contour is drawn. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

`levels` : int or array-like, optional

Determines the number and positions of the contour lines / regions.

If an int `*n*`, use ``~matplotlib.ticker.MaxNLocator``, which tries to automatically choose no more than `*n+1*` "nice" contour levels between minimum and maximum numeric values of `*Z*`.

If array-like, draw contour lines at the specified levels.

The values must be in increasing order.

#### Returns

-----

``~.contour.QuadContourSet``

#### Other Parameters

-----

`corner_mask` : bool, default: `:rc:`contour.corner_mask``

Enable/disable corner masking, which only has an effect if `*Z*` is a masked array. If ``False``, any quad touching a masked point is masked out. If ``True``, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

`colors` : `:mpltype:`color`` or list of `:mpltype:`color``, optional

The colors of the levels, i.e. the lines for ``~.contour`` and the

areas for ``contourf``.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, a single color may be used in place of one-element lists, i.e. ```red``` instead of ```[red]``` to color all levels with the same color.

.. versionchanged:: 3.10

Previously a single color had to be expressed as a string, but now any valid color format may be passed.

By default (value `*None*`), the colormap specified by `*cmap*` will be used.

`alpha` : float, default: 1

The alpha blending value, between 0 (transparent) and 1 (opaque).

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*colors*` is set.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `~matplotlib.colors.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `~matplotlib.colors.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin`, `vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`colorizer` : `~matplotlib.colorbar.Colorizer`` or None, default: None

The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

This parameter is ignored if `*colors*` is set.

`origin` : {'None', 'upper', 'lower', 'image'}, default: None  
Determines the orientation and exact position of `*Z*` by specifying the position of ```Z[0, 0]```. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: ```Z[0, 0]``` is at  $X=0$ ,  $Y=0$  in the lower left corner.
- `'lower'`: ```Z[0, 0]``` is at  $X=0.5$ ,  $Y=0.5$  in the lower left corner.
- `'upper'`: ```Z[0, 0]``` is at  $X=N+0.5$ ,  $Y=0.5$  in the upper left corner.
- `'image'`: Use the value from `:rc:``image.origin```.

`extent` : (`x0`, `x1`, `y0`, `y1`), optional

If `*origin*` is not `*None*`, then `*extent*` is interpreted as in ``.imshow``: it gives the outer pixel boundaries. In this case, the position of `Z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then (`*x0*`, `*y0*`) is the position of `Z[0, 0]`, and (`*x1*`, `*y1*`) is the position of `Z[-1, -1]`.

This argument is ignored if `*X*` and `*Y*` are specified in the call to `contour`.

`locator` : `ticker.Locator` subclass, optional

The locator is used to determine the contour levels if they are not given explicitly via `*levels*`.  
Defaults to ``.ticker.MaxNLocator``.

`extend` : {'neither', 'both', 'min', 'max'}, default: 'neither'

Determines the ```contourf```-coloring of values that are outside the `*levels*` range.

If `'neither'`, values outside the `*levels*` range are not colored.  
If `'min'`, `'max'` or `'both'`, color the values below, above or below and above the `*levels*` range.

Values below ```min(levels)``` and above ```max(levels)``` are mapped to the under/over values of the ``.Colormap``. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using ``.Colormap.set_under`` and ``.Colormap.set_over``.

.. note::

An existing ``.QuadContourSet`` does not get notified if properties of its colormap are changed. Therefore, an explicit call ``.ContourSet.changed()`` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the ``.QuadContourSet`` because it internally calls ``.ContourSet.changed()``.

Example::

```

x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()

```

xunits, yunits : registered units, optional  
 Override axis units by specifying an instance of a  
 :class:`matplotlib.units.ConversionInterface`.

antialiased : bool, optional  
 Enable antialiasing, overriding the defaults. For  
 filled contours, the default is `*False*`. For line contours,  
 it is taken from :rc:`lines.antialiased`.

nchunk : int >= 0, optional  
 If 0, no subdivision of the domain. Specify a positive integer to  
 divide the domain into subdomains of `*nchunk*` by `*nchunk*` quads.  
 Chunking reduces the maximum length of polygons generated by the  
 contouring algorithm which reduces the rendering workload passed  
 on to the backend and also requires slightly less RAM. It can  
 however introduce rendering artifacts at chunk boundaries depending  
 on the backend, the `*antialiased*` flag and value of `*alpha*`.

linewidths : float or array-like, default: :rc:`contour.linewidth`  
 \*Only applies to\* ``contour``.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with  
 the linewidths in the order specified.

If None, this falls back to :rc:`lines.linewidth`.

linestyles : {`*None*`, 'solid', 'dashed', 'dashdot', 'dotted'}, optional  
 \*Only applies to\* ``contour``.

If `*linestyles*` is `*None*`, the default is 'solid' unless the lines are  
 monochrome. In that case, negative contours will instead take their  
 linestyle from the `*negative_linestyles*` argument.

`*linestyles*` can also be an iterable of the above strings specifying a set  
 of linestyles to be used. If this iterable is shorter than the number of  
 contour levels it will be repeated as necessary.

negative\_linestyles : {`*None*`, 'solid', 'dashed', 'dashdot', 'dotted'}, optional  
 \*Only applies to\* ``contour``.

If `*linestyles*` is `*None*` and the lines are monochrome, this argument

specifies the line style for negative contours.

If `*negative_linestyles*` is `*None*`, the default is taken from `:rc:`contour.negative_linestyle``.

`*negative_linestyles*` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

`hatches` : list[str], optional  
\*Only applies to\* ``contourf``.

A list of cross hatch patterns to use on the filled areas.  
If None, no hatching will be added to the contour.

`algorithm` : {'mpl2005', 'mpl2014', 'serial', 'threaded'}, optional  
Which contouring algorithm to use to calculate the contour lines and polygons. The algorithms are implemented in ``ContourPy`` <<https://github.com/contourpy/contourpy>>`, consult the ``ContourPy`` documentation <<https://contourpy.readthedocs.io>>` for further information.

The default is taken from `:rc:`contour.algorithm``.

`clip_path` : ``~matplotlib.patches.Patch`` or ``Path`` or ``TransformedPath``  
Set the clip path. See ``~matplotlib.artist.Artist.set_clip_path``.

.. versionadded:: 3.8

`data` : indexable object, optional  
If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

Notes  
-----

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``axes.Axes.contour``.

1. ``contourf`` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to ``contour``.

2. ``contourf`` fills intervals that are closed at the top; that is, for boundaries `*z1*` and `*z2*`, the filled region is::

$$z1 < Z \leq z2$$

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

3. ``contour`` and ``contourf`` use a ``marching squares`` <[https://en.wikipedia.org/wiki/Marching\\_squares](https://en.wikipedia.org/wiki/Marching_squares)>` algorithm to compute contour locations. More information can be found in

`ContourPy documentation <<https://contourpy.readthedocs.io>>`\_.

## matplotlib.pyplot.contourf

```
contourf(*args, data=None, **kwargs) -> 'QuadContourSet'
```

Plot filled contours.

Call signature::

```
contourf([X, Y,] Z, /, [levels], **kwargs)
```

The arguments `*X*`, `*Y*`, `*Z*` are positional-only.

``~.contour`` and ``~.contourf`` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

### Parameters

-----  
`X, Y` : array-like, optional

The coordinates of the values in `*Z*`.

`*X*` and `*Y*` must both be 2D with the same shape as `*Z*` (e.g. created via ``numpy.meshgrid``), or they must both be 1-D such that ``len(X) == N`` is the number of columns in `*Z*` and ``len(Y) == M`` is the number of rows in `*Z*`.

`*X*` and `*Y*` must both be ordered monotonically.

If not given, they are assumed to be integer indices, i.e.

``X = range(N)``, ``Y = range(M)``.

`Z` : (M, N) array-like

The height values over which the contour is drawn. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

`levels` : int or array-like, optional

Determines the number and positions of the contour lines / regions.

If an int `*n*`, use ``~matplotlib.ticker.MaxNLocator``, which tries to automatically choose no more than `*n+1*` "nice" contour levels between minimum and maximum numeric values of `*Z*`.

If array-like, draw contour lines at the specified levels.

The values must be in increasing order.

### Returns

-----  
``~.contour.QuadContourSet``

### Other Parameters

-----  
`corner_mask` : bool, default: `:rc:`contour.corner_mask``

Enable/disable corner masking, which only has an effect if `*Z*` is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

`colors` : `mpltype:color` or list of `mpltype:color`, optional  
The colors of the levels, i.e. the lines for `.contour` and the areas for `.contourf`.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, a single color may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color.

.. versionchanged:: 3.10

Previously a single color had to be expressed as a string, but now any valid color format may be passed.

By default (value `*None*`), the colormap specified by `*cmap*` will be used.

`alpha` : float, default: 1  
The alpha blending value, between 0 (transparent) and 1 (opaque).

`cmap` : str or `~matplotlib.colors.Colormap`, default: `:rc:image.cmap`  
The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*colors*` is set.

`norm` : str or `~matplotlib.colors.Normalize`, optional  
The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()`. In that case, a suitable `.Normalize` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin`, `vmax` : float, optional  
When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a `str` `*norm*` name together with `*vmin*/vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`colorizer : ~matplotlib.colorbar.Colorizer` or None, default: None`  
The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

This parameter is ignored if `*colors*` is set.

`origin : {'None', 'upper', 'lower', 'image'}, default: None`  
Determines the orientation and exact position of `*Z*` by specifying the position of ```Z[0, 0]```. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: ```Z[0, 0]``` is at  $X=0$ ,  $Y=0$  in the lower left corner.
- `'lower'`: ```Z[0, 0]``` is at  $X=0.5$ ,  $Y=0.5$  in the lower left corner.
- `'upper'`: ```Z[0, 0]``` is at  $X=N+0.5$ ,  $Y=0.5$  in the upper left corner.
- `'image'`: Use the value from `:rc:'image.origin'`.

`extent : (x0, x1, y0, y1), optional`  
If `*origin*` is not `*None*`, then `*extent*` is interpreted as in ``.imshow``: it gives the outer pixel boundaries. In this case, the position of `Z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then `(*x0*, *y0*)` is the position of `Z[0, 0]`, and `(*x1*, *y1*)` is the position of `Z[-1, -1]`.

This argument is ignored if `*X*` and `*Y*` are specified in the call to `contour`.

`locator : ticker.Locator subclass, optional`  
The locator is used to determine the contour levels if they are not given explicitly via `*levels*`.  
Defaults to `~.ticker.MaxNLocator``.

`extend : {'neither', 'both', 'min', 'max'}, default: 'neither'`  
Determines the ```contourf```-coloring of values that are outside the `*levels*` range.

If `'neither'`, values outside the `*levels*` range are not colored.  
If `'min'`, `'max'` or `'both'`, color the values below, above or below and above the `*levels*` range.

Values below ```min(levels)``` and above ```max(levels)``` are mapped to the under/over values of the ``.Colormap``. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using ``.Colormap.set_under`` and ``.Colormap.set_over``.

.. note::

An existing ``.QuadContourSet`` does not get notified if

properties of its colormap are changed. Therefore, an explicit call `~.ContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the `~.QuadContourSet` because it internally calls `~.ContourSet.changed()`.

Example::

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
 colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

xunits, yunits : registered units, optional  
Override axis units by specifying an instance of a  
:class:`matplotlib.units.ConversionInterface`.

antialiased : bool, optional  
Enable antialiasing, overriding the defaults. For  
filled contours, the default is `*False*`. For line contours,  
it is taken from `:rc:`lines.antialiased``.

nchunk : int >= 0, optional  
If 0, no subdivision of the domain. Specify a positive integer to  
divide the domain into subdomains of `*nchunk*` by `*nchunk*` quads.  
Chunking reduces the maximum length of polygons generated by the  
contouring algorithm which reduces the rendering workload passed  
on to the backend and also requires slightly less RAM. It can  
however introduce rendering artifacts at chunk boundaries depending  
on the backend, the `*antialiased*` flag and value of `*alpha*`.

linewidths : float or array-like, default: `:rc:`contour.linewidth``  
`*Only applies to* ~.contour`.`

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with  
the linewidths in the order specified.

If None, this falls back to `:rc:`lines.linewidth``.

linestyles : {`*None*`, 'solid', 'dashed', 'dashdot', 'dotted'}, optional  
`*Only applies to* ~.contour`.`

If `*linestyles*` is `*None*`, the default is 'solid' unless the lines are  
monochrome. In that case, negative contours will instead take their  
linestyle from the `*negative_linestyles*` argument.

`*linestyles*` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

`negative_linestyles` : {`*None*`, `'solid'`, `'dashed'`, `'dashdot'`, `'dotted'`}, optional  
`*Only applies to* `contour`.`

If `*linestyles*` is `*None*` and the lines are monochrome, this argument specifies the line style for negative contours.

If `*negative_linestyles*` is `*None*`, the default is taken from `:rc:`contour.negative_linestyle``.

`*negative_linestyles*` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

`hatches` : list[str], optional  
`*Only applies to* `contourf`.`

A list of cross hatch patterns to use on the filled areas.  
If None, no hatching will be added to the contour.

`algorithm` : {`'mpl2005'`, `'mpl2014'`, `'serial'`, `'threaded'`}, optional  
Which contouring algorithm to use to calculate the contour lines and polygons. The algorithms are implemented in  
``ContourPy`` <<https://github.com/contourpy/contourpy>>`, consult the  
``ContourPy`` documentation <<https://contourpy.readthedocs.io>>` for further information.

The default is taken from `:rc:`contour.algorithm``.

`clip_path` : `~matplotlib.patches.Patch`` or ``Path`` or ``TransformedPath``  
Set the clip path. See `~matplotlib.artist.Artist.set_clip_path``.

.. versionadded:: 3.8

`data` : indexable object, optional  
If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

## Notes

-----

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``axes.Axes.contourf``.

1. ``contourf`` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to ``contour``.

2. ``contourf`` fills intervals that are closed at the top; that is, for boundaries `*z1*` and `*z2*`, the filled region is::

$z_1 < Z \leq z_2$

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

3. `.contour` and `.contourf` use a `marching squares` [https://en.wikipedia.org/wiki/Marching\\_squares](https://en.wikipedia.org/wiki/Marching_squares) algorithm to compute contour locations. More information can be found in `ContourPy` documentation <https://contourpy.readthedocs.io>.

## matplotlib.pyplot.cool

```
cool() -> 'None'
```

Set the colormap to 'cool'.

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

## matplotlib.pyplot.copper

```
copper() -> 'None'
```

Set the colormap to 'copper'.

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

## matplotlib.pyplot.csd

```
csd(x: 'ArrayLike', y: 'ArrayLike', *, NFFT: 'int | None' = None, Fs: 'float | None' = None, Fc: 'int | None' = None, detrend: "Literal['none', 'mean', 'linear'] | Callable[[ArrayLike], ArrayLike] | None" = None, window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, noverlap: 'int | None' = None, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None, scale_by_freq: 'bool | None' = None, return_line: 'bool | None' = None, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray] | tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the cross-spectral density.

The cross spectral density  $P_{xy}$  by Welch's average periodogram method. The vectors  $x$  and  $y$  are divided into  $NFFT$  length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The product of the direct FFTs of  $x$  and  $y$  are averaged over each segment to compute  $P_{xy}$ , with a scaling to correct for power loss due to windowing.

If  $\text{len}(x) < NFFT$  or  $\text{len}(y) < NFFT$ , they will be zero padded to  $NFFT$ .

Parameters

-----

$x, y$  : 1-D arrays or sequences

Arrays or sequences containing the data.

`Fs` : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

`window` : callable or ndarray, default: ``window_hanning``

A function or a vector of length `*NFFT*`. To create window vectors see ``window_hanning``, ``window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. This can be different from `*NFFT*`, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to ``~numpy.fft.fft``. The default is None, which sets `*pad_to*` equal to `*NFFT*`

`NFFT` : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use `*pad_to*` for this instead.

`detrend` : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the `*detrend*` parameter is a vector, in Matplotlib it is a function. The `:mod:`~matplotlib.mlab`` module defines ``detrend_none``, ``detrend_mean``, and ``detrend_linear``, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls ``detrend_none``. 'mean' calls ``detrend_mean``. 'linear' calls ``detrend_linear``.

`scale_by_freq` : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

`noverlap` : int, default: 0 (no overlap)

The number of points of overlap between segments.

`Fc` : int, default: 0

The center frequency of `*x*`, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

return\_line : bool, default: False  
Whether to include the line object plotted in the returned values.

#### Returns

-----  
Pxy : 1-D array  
The values for the cross spectrum :math:`P\_{xy}` before scaling  
(complex valued).

freqs : 1-D array  
The frequencies corresponding to the elements in \*Pxy\*.

line : ~matplotlib.lines.Line2D  
The line created by this function.  
Only returned if \*return\_line\* is True.

#### Other Parameters

-----  
data : indexable object, optional  
If given, the following parameters also accept a string ``s``, which is  
interpreted as ``data[s]`` if ``s`` is a key in ``data``:

\*x\*, \*y\*

\*\*kwargs  
Keyword arguments control the ~.Line2D` properties:

#### Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array  
and two offsets from the bottom left corner of the image

alpha: float or None

animated: bool

antialiased or aa: bool

clip\_box: ~matplotlib.transforms.BboxBase` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

color or c: :mpltype:`color`

dash\_capstyle: ~.CapStyle` or {'butt', 'projecting', 'round'}

dash\_joinstyle: ~.JoinStyle` or {'miter', 'round', 'bevel'}

dashes: sequence of floats (on/off ink in points) or (None, None)

data: (2, N) array or two 1D arrays

drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

figure: ~matplotlib.figure.Figure` or ~matplotlib.figure.SubFigure`

fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}

gapcolor: :mpltype:`color` or None

gid: str

in\_layout: bool

label: object

linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

linewidth or lw: float

marker: marker style string, ~.path.Path` or ~.markers.MarkerStyle`

markeredgecolor or mec: :mpltype:`color`

markeredgewidth or mew: float

markerfacecolor or mfc: :mpltype:`color`

markerfacecoloralt or mfcalt: :mpltype:`color`

markersize or ms: float  
markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]  
mouseover: bool  
path\_effects: list of ``AbstractPathEffect``  
picker: float or callable[[Artist, Event], tuple[bool, dict]]  
pickradius: float  
rasterized: bool  
sketch\_params: (scale: float, length: float, randomness: float)  
snap: bool or None  
solid\_capstyle: ``CapStyle`` or {'butt', 'projecting', 'round'}  
solid\_joinstyle: ``JoinStyle`` or {'miter', 'round', 'bevel'}  
transform: unknown  
url: str  
visible: bool  
xdata: 1D array  
ydata: 1D array  
zorder: float

#### See Also

-----  
psd : is equivalent to setting ``y = x``.

#### Notes

-----  
  
.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``axes.Axes.csd``.

For plotting, the power is plotted as  
:math:`10 \log\_{10}(P\_{xy})` for decibels, though :math:`P\_{xy}` itself  
is returned.

#### References

-----  
Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,  
John Wiley & Sons (1986)

## matplotlib.pylab.date2num

```
date2num(d)
```

Convert datetime objects to Matplotlib dates.

#### Parameters

-----  
d : ``datetime.datetime`` or ``numpy.datetime64`` or sequences of these

#### Returns

-----  
float or sequence of floats  
Number of days since the epoch. See ``get_epoch`` for the  
epoch, which can be changed by :code:`rc: date.epoch` or ``set_epoch``. If  
the epoch is "1970-01-01T00:00:00" (default) then noon Jan 1 1970

("1970-01-01T12:00:00") returns 0.5.

#### Notes

-----

The Gregorian calendar is assumed; this is not universal practice.  
For details see the module docstring.

## matplotlib.pyplot.datestr2num

```
datestr2num(d, default=None)
```

Convert a date string to a datenum using ``dateutil.parser.parse``.

#### Parameters

-----

d : str or sequence of str  
The dates to convert.

default : datetime.datetime, optional  
The default date to use when fields are missing in `*d*`.

## matplotlib.pyplot.delaxes

```
delaxes(ax: 'matplotlib.axes.Axes | None' = None) -> 'None'
```

Remove an `~.axes.Axes`` (defaulting to the current Axes) from its figure.

## matplotlib.pyplot.detrend

```
detrend(x, key=None, axis=None)
```

Return `*x*` with its trend removed.

#### Parameters

-----

x : array or sequence  
Array or sequence containing the data.

key : {'default', 'constant', 'mean', 'linear', 'none'} or function  
The detrending algorithm to use. 'default', 'mean', and 'constant' are the same as ``detrend_mean``. 'linear' is the same as ``detrend_linear``. 'none' is the same as ``detrend_none``. The default is 'mean'. See the corresponding functions for more details regarding the algorithms. Can also be a function that carries out the detrend operation.

axis : int  
The axis along which to do the detrending.

#### See Also

-----

detrend\_mean : Implementation of the 'mean' algorithm.  
detrend\_linear : Implementation of the 'linear' algorithm.  
detrend\_none : Implementation of the 'none' algorithm.

## matplotlib.pyplot.detrend\_linear

---

```
detrend_linear(y)
```

Return  $x$  minus best fit line; 'linear' detrending.

### Parameters

-----

y : 0-D or 1-D array or sequence

Array or sequence containing the data

### See Also

-----

detrend\_mean : Another detrend algorithm.

detrend\_none : Another detrend algorithm.

detrend : A wrapper around all the detrend algorithms.

## matplotlib.pyplot.detrend\_mean

---

```
detrend_mean(x, axis=None)
```

Return  $x$  minus the mean( $x$ ).

### Parameters

-----

x : array or sequence

Array or sequence containing the data

Can have any dimensionality

axis : int

The axis along which to take the mean. See ``numpy.mean`` for a description of this argument.

### See Also

-----

detrend\_linear : Another detrend algorithm.

detrend\_none : Another detrend algorithm.

detrend : A wrapper around all the detrend algorithms.

## matplotlib.pyplot.detrend\_none

---

```
detrend_none(x, axis=None)
```

Return  $x$ : no detrending.

### Parameters

-----

x : any object

An object containing the data

axis : int

This parameter is ignored.

It is included for compatibility with `detrend_mean`

### See Also

-----

detrend\_mean : Another detrend algorithm.  
detrend\_linear : Another detrend algorithm.  
detrend : A wrapper around all the detrend algorithms.

## matplotlib.pyplot.disconnect

```
disconnect(cid: 'int') -> 'None'
```

Disconnect the callback with id \*cid\*.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `FigureCanvasBase.mpl_disconnect``.

Examples

-----

::

```
cid = canvas.mpl_connect('button_press_event', on_press)
... later
canvas.mpl_disconnect(cid)
```

## matplotlib.pyplot.drange

```
drange(dstart, dend, delta)
```

Return a sequence of equally spaced Matplotlib dates.

The dates start at \*dstart\* and reach up to, but not including \*dend\*.  
They are spaced by \*delta\*.

Parameters

-----

dstart, dend : `~datetime.datetime``  
The date limits.  
delta : `~datetime.timedelta``  
Spacing of the dates.

Returns

-----

`~numpy.array``  
A list floats representing Matplotlib dates.

## matplotlib.pyplot.draw

```
draw() -> 'None'
```

Redraw the current figure.

This is used to update a figure that has been altered, but not automatically re-drawn. If interactive mode is on (via `~.ion()``), this

should be only rarely needed, but there may be ways to modify the state of a figure without marking it as "stale". Please report these cases as bugs.

This is equivalent to calling `fig.canvas.draw_idle()`, where `fig` is the current figure.

See Also

-----  
`.FigureCanvasBase.draw_idle`  
`.FigureCanvasBase.draw`

## matplotlib.pyplot.ecdf

```
ecdf(x: 'ArrayLike', weights: 'ArrayLike | None' = None, *, complementary: 'bool' = False, orientation: "Literal['vertical', 'horizontal']" = 'vertical', compress: 'bool' = False, data=None, **kwargs) -> 'Line2D'
```

Compute and plot the empirical cumulative distribution function of `*x`.

.. versionadded:: 3.8

Parameters

-----  
`x` : 1d array-like

The input data. Infinite entries are kept (and move the relevant end of the ecdf from 0/1), but NaNs and masked values are errors.

`weights` : 1d array-like or None, default: None

The weights of the entries; must have the same shape as `*x`.

Weights corresponding to NaN data points are dropped, and then the remaining weights are normalized to sum to 1. If unset, all entries have the same weight.

`complementary` : bool, default: False

Whether to plot a cumulative distribution function, which increases from 0 to 1 (the default), or a complementary cumulative distribution function, which decreases from 1 to 0.

`orientation` : {"vertical", "horizontal"}, default: "vertical"

Whether the entries are plotted along the x-axis ("vertical", the default) or the y-axis ("horizontal"). This parameter takes the same values as in `~.Axes.hist`.

`compress` : bool, default: False

Whether multiple entries with the same values are grouped together (with a summed weight) before plotting. This is mainly useful if `*x` contains many identical data points, to decrease the rendering complexity of the plot. If `*x` contains no duplicate points, this has no effect and just uses some time and memory.

Other Parameters

-----  
`data` : indexable object, optional

If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`:

`*x*, *weights*`

`**kwargs`

Keyword arguments control the ``Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: ``~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, ``~.path.Path`` or ``~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

`markersize` or `ms`: float

`markevery`: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

`mouseover`: bool

`path_effects`: list of ``AbstractPathEffect``

`picker`: float or callable[[Artist, Event], tuple[bool, dict]]

`pickradius`: float

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`solid_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`solid_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`transform`: unknown

`url`: str

`visible`: bool

`xdata`: 1D array

`ydata`: 1D array

`zorder`: float

Returns

-----

``Line2D``

## Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.ecdf`.

The ecdf plot can be thought of as a cumulative histogram with one bin per data entry; i.e. it reports on the entire dataset without any arbitrary binning.

If `*x*` contains NaNs or masked entries, either remove them first from the array (if they should not taken into account), or replace them by `-inf` or `+inf` (if they should be sorted at the beginning or the end of the array).

## matplotlib.pyplot.errorbar

```
errorbar(x: 'float | ArrayLike', y: 'float | ArrayLike', yerr: 'float | ArrayLike | None' = None, xerr: 'float | ArrayLike | None' = None, fmt: 'str' = '', *, ecolor: 'ColorType | None' = None, elinewidth: 'float | None' = None, capsize: 'float | None' = None, barsabove: 'bool' = False, lolims: 'bool | ArrayLike' = False, uplims: 'bool | ArrayLike' = False, xlolims: 'bool | ArrayLike' = False, xuplims: 'bool | ArrayLike' = False, errorevery: 'int | tuple[int, int]' = 1, capthick: 'float | None' = None, data=None, **kwargs) -> 'ErrorbarContainer'
```

Plot `y` versus `x` as lines and/or markers with attached errorbars.

`*x*`, `*y*` define the data locations, `*xerr*`, `*yerr*` define the errorbar sizes. By default, this draws the data markers/lines as well as the errorbars. Use `fmt='none'` to draw errorbars without any data markers.

.. versionadded:: 3.7

Caps and error lines are drawn in polar coordinates on polar plots.

## Parameters

-----

`x`, `y` : float or array-like

The data positions.

`xerr`, `yerr` : float or array-like, shape(N,) or shape(2, N), optional

The errorbar sizes:

- scalar: Symmetric +/- values for all data points.
- shape(N,): Symmetric +/-values for each data point.
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- `*None*`: No errorbar.

All values must be `>= 0`.

See :doc:`/gallery/statistics/errorbar\_features`

for an example on the usage of ``xerr`` and ``yerr``.

`fmt` : str, default: ""

The format for the data points / data lines. See ``.plot`` for details.

Use 'none' (case-insensitive) to plot errorbars without any data markers.

`ecolor` : :mpltype:`color`, default: None

The color of the errorbar lines. If None, use the color of the line connecting the markers.

`elinewidth` : float, default: None

The linewidth of the errorbar lines. If None, the linewidth of the current style is used.

`capsize` : float, default: :rc:`errorbar.capsize`

The length of the error bar caps in points.

`capthick` : float, default: None

An alias to the keyword argument `*markeredgewidth*` (a.k.a. `*mew*`). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if `*mew*` or `*markeredgewidth*` are given, then they will over-ride `*capthick*`. This may change in future releases.

`barsabove` : bool, default: False

If True, will plot the errorbars above the plot symbols. Default is below.

`lolims`, `uplims`, `xlolims`, `xuplims` : bool or array-like, default: False

These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. `*lims*`-arguments may be scalars, or array-likes of the same length as `*xerr*` and `*yerr*`. To use limits with inverted axes, `~.Axes.set_xlim`` or `~.Axes.set_ylim`` must be called before `:meth:`errorbar``. Note the tricky parameter names: setting e.g. `*lolims*` to True means that the y-value is a `*lower*` limit of the True value, so, only an `*upward*`-pointing arrow will be drawn!

`errorevery` : int or (int, int), default: 1

draws error bars on a subset of the data. `*errorevery* = N` draws error bars on the points `(x[::N], y[::N])`.

`*errorevery* = (start, N)` draws error bars on the points

`(x[start::N], y[start::N])`. e.g. `errorevery=(6, 3)`

adds error bars to the data at `(x[6], x[9], x[12], x[15], ...)`.

Used to avoid overlapping error bars when two series share x-axis values.

Returns

-----

``.ErrorbarContainer``

The container contains:

- `data_line` : A `~matplotlib.lines.Line2D`` instance of x, y plot markers and/or line.
- `caplines` : A tuple of `~matplotlib.lines.Line2D`` instances of the error bar caps.
- `barlinecols` : A tuple of `~.LineCollection`` with the horizontal and vertical error ranges.

#### Other Parameters

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*y*`, `*xerr*`, `*yerr*`

`**kwargs`

All other keyword arguments are passed on to the `~.Axes.plot`` call drawing the markers. For example, this code makes big red squares with thick green edges::

```
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s', mfc='red',
mec='green', ms=20, mew=4)
```

where `*mfc*`, `*mec*`, `*ms*` and `*mew*` are aliases for the longer property names, `*markerfacecolor*`, `*markeredgecolor*`, `*markersize*` and `*markeredgewidth*`.

Valid kwargs for the marker properties are:

- `*dashes*`
- `*dash_capstyle*`
- `*dash_joinstyle*`
- `*drawstyle*`
- `*fillstyle*`
- `*linestyle*`
- `*marker*`
- `*markeredgecolor*`
- `*markeredgewidth*`
- `*markerfacecolor*`
- `*markerfacecoloralt*`
- `*markersize*`
- `*markevery*`
- `*solid_capstyle*`
- `*solid_joinstyle*`

Refer to the corresponding `~.Line2D`` property for more details:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or `None`  
`clip_on`: `bool`  
`clip_path`: `Path` or `(Path, Transform)` or `None`  
`color` or `c`: `:mpltype:`color``  
`dash_capstyle`: ``.CapStyle`` or `{'butt', 'projecting', 'round'}`  
`dash_joinstyle`: ``.JoinStyle`` or `{'miter', 'round', 'bevel'}`  
`dashes`: sequence of floats (on/off ink in points) or `(None, None)`  
`data`: `(2, N)` array or two 1D arrays  
`drawstyle` or `ds`: `{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}`, default: `'default'`  
`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``  
`fillstyle`: `{'full', 'left', 'right', 'bottom', 'top', 'none'}`  
`gapcolor`: `:mpltype:`color`` or `None`  
`gid`: `str`  
`in_layout`: `bool`  
`label`: `object`  
`linestyle` or `ls`: `{'-'', '--', '-.', ':', ''}`, (offset, on-off-seq), ...}  
`linewidth` or `lw`: `float`  
`marker`: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``  
`markeredgecolor` or `mec`: `:mpltype:`color``  
`markeredgewidth` or `mew`: `float`  
`markerfacecolor` or `mfc`: `:mpltype:`color``  
`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``  
`markersize` or `ms`: `float`  
`markevery`: `None` or `int` or `(int, int)` or `slice` or `list[int]` or `float` or `(float, float)` or `list[bool]`  
`mouseover`: `bool`  
`path_effects`: list of ``.AbstractPathEffect``  
`picker`: `float` or callable[[`Artist`, `Event`], `tuple[bool, dict]`]  
`pickradius`: `float`  
`rasterized`: `bool`  
`sketch_params`: (scale: `float`, length: `float`, randomness: `float`)  
`snap`: `bool` or `None`  
`solid_capstyle`: ``.CapStyle`` or `{'butt', 'projecting', 'round'}`  
`solid_joinstyle`: ``.JoinStyle`` or `{'miter', 'round', 'bevel'}`  
`transform`: `unknown`  
`url`: `str`  
`visible`: `bool`  
`xdata`: 1D array  
`ydata`: 1D array  
`zorder`: `float`

## Notes

-----

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``.axes.Axes.errorbar``.

## matplotlib.pyplot.eventplot

```

eventplot(positions: 'ArrayLike | Sequence[ArrayLike]', *, orientation:
 "Literal['horizontal', 'vertical']" = 'horizontal', lineoffsets: 'float |
 Sequence[float]' = 1, linelengths: 'float | Sequence[float]' = 1, linewidths: 'float |
 Sequence[float] | None' = None, colors: 'ColorType | Sequence[ColorType] | None' =
 None, alpha: 'float | Sequence[float] | None' = None, linestyles: 'LineStyleType |
 Sequence[LineStyleType]' = 'solid', data=None, **kwargs) -> 'EventCollection'

```

Plot identical parallel lines at the given positions.

This type of plot is commonly used in neuroscience for representing neural events, where it is usually called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

#### Parameters

-----

`positions` : array-like or list of array-like

A 1D array-like defines the positions of one sequence of events.

Multiple groups of events may be passed as a list of array-likes. Each group can be styled independently by passing lists of values to `*lineoffsets*`, `*linelengths*`, `*linewidths*`, `*colors*` and `*linestyles*`.

Note that `*positions*` can be a 2D array, but in practice different event groups usually have different counts so that one will use a list of different-length arrays rather than a 2D array.

`orientation` : {'horizontal', 'vertical'}, default: 'horizontal'

The direction of the event sequence:

- 'horizontal': the events are arranged horizontally.

The indicator lines are vertical.

- 'vertical': the events are arranged vertically.

The indicator lines are horizontal.

`lineoffsets` : float or array-like, default: 1

The offset of the center of the lines from the origin, in the direction orthogonal to `*orientation*`.

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

`linelengths` : float or array-like, default: 1

The total height of the lines (i.e. the lines stretches from ```lineoffset - linelength/2``` to ```lineoffset + linelength/2```).

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

`linewidths` : float or array-like, default: `:rc:`lines.linewidth``

The line width(s) of the event lines, in points.

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

`colors` : `:mplttype:`color`` or list of color, default: `:rc:`lines.color``

The color(s) of the event lines.

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

`alpha` : float or array-like, default: 1

The alpha blending value(s), between 0 (transparent) and 1 (opaque).

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

`linestyles` : str or tuple or list of such values, default: 'solid'

Default is 'solid'. Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form::

(offset, onoffseq),

where `*onoffseq*` is an even length tuple of on and off ink in points.

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

`data` : indexable object, optional

If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`:

`*positions*`, `*lineoffsets*`, `*linelengths*`, `*linewidths*`, `*colors*`, `*linestyles*`

**\*\*kwargs**

Other keyword arguments are line collection properties. See `.LineCollection`` for a list of the valid properties.

Returns

-----

list of `.EventCollection``

The `.EventCollection`` that were added.

Notes

-----

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for `.axes.Axes.eventplot``.

For `*linelengths*`, `*linewidths*`, `*colors*`, `*alpha*` and `*linestyles*`, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as `*positions*`, and each value will be applied to the corresponding row of the array.

Examples

-----

.. plot:: gallery/lines\_bars\_and\_markers/eventplot\_demo.py

## matplotlib.pyplot.figaspect

```
figaspect(arg)
```

Calculate the width and height for a figure with a specified aspect ratio.

While the height is taken from `:rc:`figure.figsize``, the width is adjusted to match the desired aspect ratio. Additionally, it is ensured that the width is in the range `[4., 16.]` and the height is in the range `[2., 16.]`. If necessary, the default height is adjusted to ensure this.

### Parameters

-----

`arg` : float or 2D array

If a float, this defines the aspect ratio (i.e. the ratio height / width).

In case of an array the aspect ratio is number of rows / number of columns, so that the array could be fitted in the figure undistorted.

### Returns

-----

size : (2,) array

The width and height of the figure in inches.

### Notes

-----

If you want to create an Axes within the figure, that still preserves the aspect ratio, be sure to create it with equal width and height. See examples below.

Thanks to Fernando Perez for this function.

### Examples

-----

Make a figure twice as tall as it is wide::

```
w, h = figaspect(2.)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

Make a figure with the proper aspect for an array::

```
A = rand(5, 3)
w, h = figaspect(A)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

## matplotlib.pyplot.figimage

```
figimage(X: 'ArrayLike', xo: 'int' = 0, yo: 'int' = 0, alpha: 'float | None' = None,
norm: 'str | Normalize | None' = None, cmap: 'str | Colormap | None' = None, vmin:
'float | None' = None, vmax: 'float | None' = None, origin: "Literal['upper', 'lower']
| None" = None, resize: 'bool' = False, *, colorizer: 'Colorizer | None' = None,
**kwargs) -> 'FigureImage'
```

Add a non-resampled image to the figure.

The image is attached to the lower or upper left corner depending on `*origin*`.

#### Parameters

-----

`X`

The image data. This is an array of one of the following shapes:

- (M, N): an image with scalar data. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

`xo, yo : int`

The `*x*/y*` image offset in pixels.

`alpha : None or float`

The alpha blending value.

`cmap : str or ~matplotlib.colors.Colormap`, default: `:rc:image.cmap`

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*X*` is RGB(A).

`norm : str or ~matplotlib.colors.Normalize`, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize` or one of its subclasses (see `:ref:colormapnorms`).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()`. In that case, a suitable `.Normalize` subclass is dynamically generated and instantiated.

This parameter is ignored if `*X*` is RGB(A).

`vmin, vmax : float`, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a `str` *norm*` name together with `*vmin*/vmax*` is acceptable).

This parameter is ignored if *\*X\** is RGB(A).

`origin : {'upper', 'lower'}, default: :rc:`image.origin``  
Indicates where the [0, 0] index of the array is in the upper left or lower left corner of the Axes.

`resize : bool`  
If *\*True\**, resize the figure to match the given image size.

`colorizer : ~matplotlib.colorbar.Colorizer` or None, default: None`  
The Colorizer object used to map color to data. If None, a Colorizer object is created from a *\*norm\** and *\*cmap\**.

This parameter is ignored if *\*X\** is RGB(A).

#### Returns

-----  
``matplotlib.image.FigureImage``

#### Other Parameters

-----  
**\*\*kwargs**  
Additional kwargs are ``Artist`` kwargs passed on to ``FigureImage``.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``Figure.figimage``.

`figimage` complements the Axes image (``~matplotlib.axes.Axes.imshow``) which will be resampled to fit the current Axes. If you want a resampled image to fill the entire figure, you can define an ``~matplotlib.axes.Axes`` with extent [0, 0, 1, 1].

#### Examples

-----

::

```
f = plt.figure()
nx = int(f.get_figwidth() * f.dpi)
ny = int(f.get_figheight() * f.dpi)
data = np.random.random((ny, nx))
f.figimage(data)
plt.show()
```

## matplotlib.pyplot.figlegend

```
figlegend(*args, **kwargs) -> 'Legend'
```

Place a legend on the figure.

Call signatures::

```
figlegend()
figlegend(handles, labels)
figlegend(handles=handles)
figlegend(labels)
```

The call signatures correspond to the following different ways to use this method:

**\*\*1. Automatic detection of elements to be shown in the legend\*\***

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the :meth:`~.Artist.set\_label` method on the artist::

```
plt.plot([1, 2, 3], label='Inline label')
plt.figlegend()
```

or::

```
line, = plt.plot([1, 2, 3])
line.set_label('Label via method')
plt.figlegend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling `Figure.legend` without any arguments and without setting the labels manually will result in no legend being drawn.

**\*\*2. Explicitly listing the artists and labels in the legend\*\***

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively::

```
plt.figlegend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

**\*\*3. Explicitly listing the artists in the legend\*\***

This is similar to 2, but the labels are taken from the artists' label properties. Example::

```
line1, = ax1.plot([1, 2, 3], label='label1')
line2, = ax2.plot([1, 2, 3], label='label2')
plt.figlegend(handles=[line1, line2])
```

**\*\*4. Labeling existing plot elements\*\***

.. admonition:: Discouraged

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

To make a legend for all artists on all Axes, call this function with an iterable of strings, one for each legend item. For example::

```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot([1, 3, 5], color='blue')
ax2.plot([2, 4, 6], color='red')
plt.figlegend(['the blues', 'the reds'])
```

#### Parameters

handles : list of `Artist`, optional

A list of Artists (lines, patches) to be added to the legend.

Use this together with `*labels*`, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

labels : list of str, optional

A list of labels to show next to the artists.

Use this together with `*handles*`, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

#### Returns

`~matplotlib.legend.Legend`

#### Other Parameters

loc : str or pair of floats, default: 'upper right'

The location of the legend.

The strings `''upper left''`, `''upper right''`, `''lower left''`, `''lower right''` place the legend at the corresponding corner of the figure.

The strings `''upper center''`, `''lower center''`, `''center left''`, `''center right''` place the legend at the center of the corresponding edge of the figure.

The string `''center''` places the legend at the center of the figure.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in figure coordinates (in which case `*bbox_to_anchor*`

will be ignored).

For back-compatibility, ``'center right'`` (but no other location) can also be spelled ``'right'``, and each "string" location can also be given as a numeric value:

```
=====
Location String Location Code
=====
'best' (Axes only) 0
'upper right' 1
'upper left' 2
'lower left' 3
'lower right' 4
'right' 5
'center left' 6
'center right' 7
'lower center' 8
'upper center' 9
'center' 10
=====
```

If a figure is using the constrained layout manager, the string codes of the `*loc*` keyword argument can get better layout behaviour using the prefix 'outside'. There is ambiguity at the corners, so 'outside upper right' will make space for the legend above the rest of the axes in the layout, and 'outside right upper' will make space on the right side of the layout. In addition to the values of `*loc*` listed above, we have 'outside right upper', 'outside right lower', 'outside left upper', and 'outside left lower'. See :ref:`legend\_guide` for more details.

`bbox_to_anchor` : `.BboxBase``, 2-tuple, or 4-tuple of floats  
Box that is used to position the legend in conjunction with `*loc*`.  
Defaults to ```axes.bbox``` (if called as a method to `.Axes.legend``) or ```figure.bbox``` (if ```figure.legend```). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by `*bbox_transform*`, with the default transform Axes or Figure coordinates, depending on which ```legend``` is called.

If a 4-tuple or `.BboxBase`` is given, then it specifies the bbox ```(x, y, width, height)``` that the legend is placed in.  
To put the legend in the best location in the bottom right quadrant of the Axes (or figure)::

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple ```(x, y)``` places the corner of the legend specified by `*loc*` at `x, y`. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used::

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

`ncols` : int, default: 1

The number of columns that the legend has.

For backward compatibility, the spelling `*ncol*` is also supported but it is discouraged. If both are given, `*ncols*` takes precedence.

`prop` : None or `~matplotlib.font_manager.FontProperties`` or dict

The font properties of the legend. If None (default), the current

`:data:`matplotlib.rcParams`` will be used.

`fontsize` : int or `{'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}`

The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if `*prop*` is not specified.

`labelcolor` : str or list, default: `:rc:`legend.labelcolor``

The color of the text in the legend. Either a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using `:rc:`legend.labelcolor``. If None, use `:rc:`text.color``.

`numpoints` : int, default: `:rc:`legend.numpoints``

The number of marker points in the legend when creating a legend entry for a `.Line2D`` (line).

`scatterpoints` : int, default: `:rc:`legend.scatterpoints``

The number of marker points in the legend when creating a legend entry for a `.PathCollection`` (scatter plot).

`scatteryoffsets` : iterable of floats, default: ```[0.375, 0.5, 0.3125]```

The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to ```[0.5]```.

`markerscale` : float, default: `:rc:`legend.markerscale``

The relative size of legend markers compared to the originally drawn ones.

`markerfirst` : bool, default: True

If `*True*`, legend marker is placed to the left of the legend label.

If `*False*`, legend marker is placed to the right of the legend label.

`reverse` : bool, default: False

If `*True*`, the legend labels are displayed in reverse order from the input.

If `*False*`, the legend labels are displayed in the same order as the input.

.. versionadded:: 3.7

`frameon` : bool, default: `:rc:`legend.frameon``

Whether the legend should be drawn on a patch (frame).

`fancybox` : bool, default: `:rc:`legend.fancybox``

Whether round edges should be enabled around the ``FancyBboxPatch`` which makes up the legend's background.

`shadow` : None, bool or dict, default: `:rc:`legend.shadow``

Whether to draw a shadow behind the legend.

The shadow can be configured using ``Patch`` keywords.

Customization via `:rc:`legend.shadow`` is currently not supported.

`framealpha` : float, default: `:rc:`legend.framealpha``

The alpha transparency of the legend's background.

If `*shadow*` is activated and `*framealpha*` is ```None```, the default value is ignored.

`facecolor` : "inherit" or color, default: `:rc:`legend.facecolor``

The legend's background color.

If ```"inherit"```, use `:rc:`axes.facecolor``.

`edgecolor` : "inherit" or color, default: `:rc:`legend.edgecolor``

The legend's background patch edge color.

If ```"inherit"```, use `:rc:`axes.edgecolor``.

`mode` : {"expand", None}

If `*mode*` is set to ```"expand"``` the legend will be horizontally expanded to fill the Axes area (or `*bbox_to_anchor*` if defines the legend's size).

`bbox_transform` : None or `~matplotlib.transforms.Transform``

The transform for the bounding box (`*bbox_to_anchor*`). For a value of ```None``` (default) the Axes'

`:data:~matplotlib.axes.Axes.transAxes`` transform will be used.

`title` : str or None

The legend's title. Default is no title (```None```).

`title_fontproperties` : None or `~matplotlib.font_manager.FontProperties`` or dict

The font properties of the legend's title. If None (default), the

`*title_fontsize*` argument will be used if present; if `*title_fontsize*` is also None, the current `:rc:`legend.title_fontsize`` will be used.

`title_fontsize` : int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default:

`:rc:`legend.title_fontsize``

The font size of the legend's title.

Note: This cannot be combined with `*title_fontproperties*`. If you want to set the fontsize alongside other font properties, use the `*size*` parameter in `*title_fontproperties*`.

`alignment` : {'center', 'left', 'right'}, default: 'center'

The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

`borderpad` : float, default: `:rc:`legend.borderpad``

The fractional whitespace inside the legend border, in font-size units.

`labelspacing` : float, default: `:rc:`legend.labelspacing``

The vertical space between the legend entries, in font-size units.

`handlelength` : float, default: `:rc:`legend.handlelength``  
The length of the legend handles, in font-size units.

`handleheight` : float, default: `:rc:`legend.handleheight``  
The height of the legend handles, in font-size units.

`handletextpad` : float, default: `:rc:`legend.handletextpad``  
The pad between the legend handle and text, in font-size units.

`borderaxespad` : float, default: `:rc:`legend.borderaxespad``  
The pad between the Axes and legend border, in font-size units.

`columnspacing` : float, default: `:rc:`legend.columnspacing``  
The spacing between columns, in font-size units.

`handler_map` : dict or None  
The custom dictionary mapping instances or types to a legend handler. This `*handler_map*` updates the default handler map found at ``matplotlib.legend.Legend.get_legend_handler_map``.

`draggable` : bool, default: False  
Whether the legend can be dragged with the mouse.

See Also

-----  
`.Axes.legend`

Notes

-----  
Some artists are not supported by this function. See `:ref:`legend_guide`` for details.

## matplotlib.pyplot.figure\_exists

```
figure_exists(num: 'int | str') -> 'bool'
```

Return whether the figure with the given id exists.

Parameters

-----  
`num` : int or str  
A figure identifier.

Returns

-----  
bool  
Whether or not a figure with id `*num*` exists.

## matplotlib.pyplot.figtext

```
figtext(x: 'float', y: 'float', s: 'str', fontdict: 'dict[str, Any] | None' = None, **kwargs) -> 'Text'
```

Add text to figure.

#### Parameters

-----

`x, y : float`

The position to place the text. By default, this is in figure coordinates, floats in [0, 1]. The coordinate system can be changed using the `*transform*` keyword.

`s : str`

The text string.

`fontdict : dict, optional`

A dictionary to override the default text properties. If not given, the defaults are determined by `:rc:`font.*``. Properties passed as `*kwargs*` override the corresponding ones given in `*fontdict*`.

#### Returns

-----

``~.text.Text``

#### Other Parameters

-----

`**kwargs : `~matplotlib.text.Text` properties`

Other miscellaneous text parameters.

#### Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased`: bool

`backgroundcolor`: `:mpltype:`color``

`bbox`: dict with properties for ``~.patches.FancyBboxPatch``

`clip_box`: unknown

`clip_on`: unknown

`clip_path`: unknown

`color` or `c`: `:mpltype:`color``

`figure`: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

`fontfamily` or `family` or `fontname`: {`'FONTNAME'`, `'serif'`, `'sans-serif'`, `'cursive'`, `'fantasy'`, `'monospace'`}

`fontproperties` or `font` or `font_properties`: ``~.font_manager.FontProperties`` or ``str`` or ``pathlib.Path``

`fontsize` or `size`: float or {`'xx-small'`, `'x-small'`, `'small'`, `'medium'`, `'large'`, `'x-large'`, `'xx-large'`}

`fontstretch` or `stretch`: {a numeric value in range 0-1000, `'ultra-condensed'`, `'extra-condensed'`, `'condensed'`, `'semi-condensed'`, `'normal'`, `'semi-expanded'`, `'expanded'`, `'extra-expanded'`, `'ultra-expanded'`}

`fontstyle` or `style`: {`'normal'`, `'italic'`, `'oblique'`}

`fontvariant` or `variant`: {`'normal'`, `'small-caps'`}

`fontweight` or `weight`: {a numeric value in range 0-1000, `'ultralight'`, `'light'`, `'normal'`, `'regular'`, `'book'`, `'medium'`, `'roman'`, `'semibold'`, `'demibold'`, `'demi'`, `'bold'`, `'heavy'`, `'extra bold'`, `'black'`}

`gid`: str

`horizontalalignment` or `ha`: {`'left'`, `'center'`, `'right'`}

`in_layout`: bool

`label`: object

`linespacing`: float (multiple of font size)

`math_fontfamily`: str

`mouseover`: bool

`multialignment` or `ma`: {`'left'`, `'right'`, `'center'`}

parse\_math: bool  
 path\_effects: list of ``~.AbstractPathEffect``  
 picker: None or bool or float or callable  
 position: (float, float)  
 rasterized: bool  
 rotation: float or {'vertical', 'horizontal'}  
 rotation\_mode: {None, 'default', 'anchor'}  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 text: object  
 transform: ``~matplotlib.transforms.Transform``  
 transform\_rotates\_text: bool  
 url: str  
 usetex: bool, default: `:rc:`text.usetex``  
 verticalalignment or va: {'baseline', 'bottom', 'center', 'center\_baseline', 'top'}  
 visible: bool  
 wrap: bool  
 x: float  
 y: float  
 zorder: float

#### See Also

-----  
[.Axes.text](#)  
[.pyplot.text](#)

#### Notes

-----

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for ``~.Figure.text``.

## matplotlib.pyplot.figure

```
figure(num: 'int | str | Figure | SubFigure | None' = None, figsize: 'ArrayLike | None' = None, dpi: 'float | None' = None, *, facecolor: 'ColorType | None' = None, edgecolor: 'ColorType | None' = None, frameon: 'bool' = True, FigureClass: 'type[Figure]' = , clear: 'bool' = False, **kwargs) -> 'Figure'
```

Create a new figure, or activate an existing figure.

#### Parameters

-----

num : int or str or ``~.Figure`` or ``~.SubFigure``, optional  
 A unique identifier for the figure.

If a figure with that identifier already exists, this figure is made active and returned. An integer refers to the ```Figure.number``` attribute, a string refers to the figure label.

If there is no figure with the identifier or `*num*` is not given, a new figure is created, made active and returned. If `*num*` is an int, it will be used for the ```Figure.number``` attribute, otherwise, an auto-generated integer value is used (starting at 1 and incremented

for each new figure). If `*num*` is a string, the figure label and the window title is set to this value. If `num` is a `SubFigure`, its parent `Figure` is activated.

`figsize` : (float, float), default: `rc('figure.figsize')`  
Width, height in inches.

`dpi` : float, default: `rc('figure.dpi')`  
The resolution of the figure in dots-per-inch.

`facecolor` : `mpltype:'color'`, default: `rc('figure.facecolor')`  
The background color.

`edgecolor` : `mpltype:'color'`, default: `rc('figure.edgecolor')`  
The border color.

`frameon` : bool, default: True  
If False, suppress drawing the figure frame.

`FigureClass` : subclass of `~matplotlib.figure.Figure`  
If set, an instance of this subclass will be created, rather than a plain `.Figure`.

`clear` : bool, default: False  
If True and the figure already exists, then it is cleared.

`layout` : {'constrained', 'compressed', 'tight', 'none', `.LayoutEngine`, None}, default: None  
The layout mechanism for positioning of plot elements to avoid overlapping Axes decorations (labels, ticks, etc). Note that layout managers can measurably slow down figure display.

- 'constrained': The constrained layout solver adjusts Axes sizes to avoid overlapping Axes decorations. Can handle complex plot layouts and colorbars, and is thus recommended.

See `:ref:'constrainedlayout_guide'` for examples.

- 'compressed': uses the same algorithm as 'constrained', but removes extra space between fixed-aspect-ratio Axes. Best for simple grids of Axes.

- 'tight': Use the tight layout mechanism. This is a relatively simple algorithm that adjusts the subplot parameters so that decorations do not overlap. See `.Figure.set_tight_layout` for further details.

- 'none': Do not use a layout engine.

- A `.LayoutEngine` instance. Builtin layout classes are `.ConstrainedLayoutEngine` and `.TightLayoutEngine`, more easily accessible by 'constrained' and 'tight'. Passing an instance allows third parties to provide their own layout engine.

If not given, fall back to using the parameters `*tight_layout*` and

\*constrained\_layout\*, including their config defaults  
:rc:`figure.autolayout` and :rc:`figure.constrained\_layout.use`.

**\*\*kwargs**

Additional keyword arguments are passed to the `Figure` constructor.

**Returns**

-----  
`~matplotlib.figure.Figure`

**Notes**

-----  
A newly created figure is passed to the `~.FigureCanvasBase.new_manager` method or the `new_figure_manager` function provided by the current backend, which install a canvas and a manager on the figure.

Once this is done, :rc:`figure.hooks` are called, one at a time, on the figure; these hooks allow arbitrary customization of the figure (e.g., attaching callbacks) or of associated elements (e.g., modifying the toolbar). See :doc:`/gallery/user\_interfaces/mplcvd` for an example of toolbar customization.

If you are creating many figures, make sure you explicitly call `~.pyplot.close` on the figures you are not using, because this will enable pyplot to properly clean up the memory.

`~matplotlib.rcParams` defines the default values, which can be modified in the `matplotlibrc` file.

## matplotlib.pyplot.fill

```
fill(*args, data=None, **kwargs) -> 'list[Polygon]'
```

Plot filled polygons.

**Parameters**

-----  
**\*args** : sequence of x, y, [color]  
Each polygon is defined by the lists of *\*x\** and *\*y\** positions of its nodes, optionally followed by a *\*color\** specifier. See :mod:`matplotlib.colors` for supported color specifiers. The standard color cycle is used for polygons without a color specifier.

You can plot multiple polygons by providing multiple *\*x\**, *\*y\**, *\*[color]\** groups.

For example, each of the following is legal::

```
ax.fill(x, y) # a polygon with default color
ax.fill(x, y, "b") # a blue polygon
ax.fill(x, y, x2, y2) # two polygons
ax.fill(x, y, "b", x2, y2, "r") # a blue and a red polygon
```

data : indexable object, optional

An object with labelled data. If given, provide the label names to plot in `*x*` and `*y*`, e.g.::

```
ax.fill("time", "signal",
data={"time": [0, 1, 2], "signal": [0, 1, 0]})
```

Returns

-----  
list of `~matplotlib.patches.Polygon``

Other Parameters

-----  
`**kwargs` : `~matplotlib.patches.Polygon`` properties

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.fill``.

Use :meth:`fill\_between` if you would like to fill the region between two curves.

## matplotlib.pyplot.fill\_between

```
fill_between(x: 'ArrayLike', y1: 'ArrayLike | float', y2: 'ArrayLike | float' = 0,
where: 'Sequence[bool] | None' = None, interpolate: 'bool' = False, step:
'Literal['pre', 'post', 'mid'] | None' = None, *, data=None, **kwargs) ->
'FillBetweenPolyCollection'
```

Fill the area between two horizontal curves.

The curves are defined by the points `(*x*, *y1*)` and `(*x*, *y2*)`. This creates one or multiple polygons describing the filled area.

You may exclude some horizontal sections from filling using `*where*`.

By default, the edges connect the given points directly. Use `*step*` if the filling should be a step function, i.e. constant in between `*x*`.

Parameters

-----

x : array-like

The x coordinates of the nodes defining the curves.

y1 : array-like or float

The y coordinates of the nodes defining the first curve.

y2 : array-like or float, default: 0

The y coordinates of the nodes defining the second curve.

where : array-like of bool, optional

Define *\*where\** to exclude some horizontal regions from being filled.

The filled regions are defined by the coordinates ``x[where]``.

More precisely, fill between ``x[i]`` and ``x[i+1]`` if

``where[i] and where[i+1]``. Note that this definition implies

that an isolated *\*True\** value between two *\*False\** values in *\*where\**

will not result in filling. Both sides of the *\*True\** position

remain unfilled due to the adjacent *\*False\** values.

interpolate : bool, default: False

This option is only relevant if *\*where\** is used and the two curves

are crossing each other.

Semantically, *\*where\** is often used for *\*y1\* > \*y2\** or

similar. By default, the nodes of the polygon defining the filled

region will only be placed at the positions in the *\*x\** array.

Such a polygon cannot describe the above semantics close to the

intersection. The x-sections containing the intersection are

simply clipped.

Setting *\*interpolate\** to *\*True\** will calculate the actual

intersection point and extend the filled region up to this point.

step : {'pre', 'post', 'mid'}, optional

Define *\*step\** if the filling should be a step function,

i.e. constant in between *\*x\**. The value determines where the

step will occur:

- 'pre': The y value is continued constantly to the left from

every *\*x\** position, i.e. the interval ``x[i-1], x[i]``

has the value ``y[i]``.

- 'post': The y value is continued constantly to the right from

every *\*x\** position, i.e. the interval ``x[i], x[i+1]``

has the value ``y[i]``.

- 'mid': Steps occur half-way between the *\*x\** positions.

Returns

-----

`.FillBetweenPolyCollection`

A *`.FillBetweenPolyCollection`* containing the plotted polygons.

Other Parameters

-----

data : indexable object, optional

If given, the following parameters also accept a string *``s``*, which is

interpreted as *``data[s]``* if *``s``* is a key in *``data``*:

*\*x\**, *\*y1\**, *\*y2\**, *\*where\**

**\*\*kwargs**

All other keyword arguments are passed on to

*`.FillBetweenPolyCollection`*. They control the *`.Polygon`* properties:

Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array

and two offsets from the bottom left corner of the image  
 alpha: array-like or float or None  
 animated: bool  
 antialiased or aa or antialiaseds: bool or list of bools  
 array: array-like or None  
 capstyle: ``~matplotlib.transforms.BboxBase`` or `{'butt', 'projecting', 'round'}`  
 clim: (vmin: float, vmax: float)  
 clip\_box: ``~matplotlib.transforms.BboxBase`` or None  
 clip\_on: bool  
 clip\_path: Patch or (Path, Transform) or None  
 cmap: ``~matplotlib.colors.Colormap`` or str or None  
 color: :mpltype:color or list of RGBA tuples  
 data: array-like  
 edgecolor or ec or edgecolors: :mpltype:color or list of :mpltype:color or 'face'  
 facecolor or facecolors or fc: :mpltype:color or list of :mpltype:color  
 figure: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``  
 gid: str  
 hatch: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`  
 hatch\_linewidth: unknown  
 in\_layout: bool  
 joinstyle: ``~matplotlib.transforms.Transform`` or `{'miter', 'round', 'bevel'}`  
 label: object  
 linestyle or dashes or linestyles or ls: str or tuple or list thereof  
 linewidth or linewidths or lw: float or list of floats  
 mouseover: bool  
 norm: ``~matplotlib.colors.Normalize`` or str or None  
 offset\_transform or transOffset: ``~matplotlib.transforms.Transform``  
 offsets: (N, 2) or (2,) array-like  
 path\_effects: list of ``~matplotlib.transforms.Transform``  
 paths: list of array-like  
 picker: None or bool or float or callable  
 pickradius: float  
 rasterized: bool  
 sizes: ``numpy.ndarray`` or None  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 transform: ``~matplotlib.transforms.Transform``  
 url: str  
 urls: list of str or None  
 verts: list of array-like  
 verts\_and\_codes: unknown  
 visible: bool  
 zorder: float

## See Also

-----

`fill_between` : Fill between two sets of y-values.

`fill_betweenx` : Fill between two sets of x-values.

## Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``~matplotlib.axes.Axes.fill_between``.

## matplotlib.pyplot.fill\_betweenx

```
fill_betweenx(y: 'ArrayLike', x1: 'ArrayLike | float', x2: 'ArrayLike | float' = 0,
where: 'Sequence[bool] | None' = None, step: "Literal['pre', 'post', 'mid'] | None" =
None, interpolate: 'bool' = False, *, data=None, **kwargs) ->
'FillBetweenPolyCollection'
```

Fill the area between two vertical curves.

The curves are defined by the points  $(y, x1)$  and  $(y, x2)$ . This creates one or multiple polygons describing the filled area.

You may exclude some vertical sections from filling using `*where*`.

By default, the edges connect the given points directly. Use `*step*` if the filling should be a step function, i.e. constant in between `*y*`.

### Parameters

-----

`y` : array-like

The y coordinates of the nodes defining the curves.

`x1` : array-like or float

The x coordinates of the nodes defining the first curve.

`x2` : array-like or float, default: 0

The x coordinates of the nodes defining the second curve.

`where` : array-like of bool, optional

Define `*where*` to exclude some vertical regions from being filled.

The filled regions are defined by the coordinates `y[where]`.

More precisely, fill between `y[i]` and `y[i+1]` if

`where[i]` and `where[i+1]`. Note that this definition implies that an isolated `*True*` value between two `*False*` values in `*where*` will not result in filling. Both sides of the `*True*` position remain unfilled due to the adjacent `*False*` values.

`interpolate` : bool, default: False

This option is only relevant if `*where*` is used and the two curves are crossing each other.

Semantically, `*where*` is often used for `*x1* > *x2*` or

similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the `*y*` array.

Such a polygon cannot describe the above semantics close to the intersection. The y-sections containing the intersection are simply clipped.

Setting `*interpolate*` to `*True*` will calculate the actual intersection point and extend the filled region up to this point.

`step` : {'pre', 'post', 'mid'}, optional

Define `*step*` if the filling should be a step function,

i.e. constant in between *\*y\**. The value determines where the step will occur:

- 'pre': The x value is continued constantly to the left from every *\*y\** position, i.e. the interval `[(y[i-1], y[i]]` has the value `x[i]`.
- 'post': The y value is continued constantly to the right from every *\*y\** position, i.e. the interval `[y[i], y[i+1])` has the value `x[i]`.
- 'mid': Steps occur half-way between the *\*y\** positions.

#### Returns

`.FillBetweenPolyCollection``  
A `.FillBetweenPolyCollection`` containing the plotted polygons.

#### Other Parameters

data : indexable object, optional

If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`:

*\*y\*, \*x1\*, \*x2\*, \*where\**

#### \*\*kwargs

All other keyword arguments are passed on to `.FillBetweenPolyCollection``. They control the `.Polygon`` properties:

#### Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: array-like or float or None

animated: bool

antialiased or aa or antialiaseds: bool or list of bools

array: array-like or None

capstyle: `.CapStyle`` or {'butt', 'projecting', 'round'}

clim: (vmin: float, vmax: float)

clip\_box: `~matplotlib.transforms.BboxBase`` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

cmap: `.Colormap`` or str or None

color: `:mpltype:`color`` or list of RGBA tuples

data: array-like

edgecolor or ec or edgecolors: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'

facecolor or facecolors or fc: `:mpltype:`color`` or list of `:mpltype:`color``

figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

gid: str

hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '\*'}

hatch\_linewidth: unknown

in\_layout: bool

joinstyle: `.JoinStyle`` or {'miter', 'round', 'bevel'}

label: object

linestyle or dashes or linestyles or ls: str or tuple or list thereof

linewidth or linewidths or lw: float or list of floats

mouseover: bool

norm: ``Normalize`` or str or None  
offset\_transform or transOffset: ``Transform``  
offsets: (N, 2) or (2,) array-like  
path\_effects: list of ``AbstractPathEffect``  
paths: list of array-like  
picker: None or bool or float or callable  
pickradius: float  
rasterized: bool  
sizes: ``numpy.ndarray`` or None  
sketch\_params: (scale: float, length: float, randomness: float)  
snap: bool or None  
transform: ``~matplotlib.transforms.Transform``  
url: str  
urls: list of str or None  
verts: list of array-like  
verts\_and\_codes: unknown  
visible: bool  
zorder: float

#### See Also

-----

`fill_between` : Fill between two sets of y-values.

`fill_betweenx` : Fill between two sets of x-values.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``axes.Axes.fill_betweenx``.

## matplotlib.pyplot.findobj

```
findobj(o: 'Artist | None' = None, match: 'Callable[[Artist], bool] | type[Artist] | None' = None, include_self: 'bool' = True) -> 'list[Artist]'
```

Find artist objects.

Recursively find all ``Artist`` instances contained in the artist.

#### Parameters

-----

match

A filter criterion for the matches. This can be

- `*None*`: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool``. The result will only contain artists for which the function returns `*True*`.
- A class instance: e.g., ``Line2D``. The result will only contain artists of this class or its subclasses (`isinstance`` check).

include\_self : bool

Include `*self*` in the list to be checked for a match.

Returns

-----

list of ``Artist``

## matplotlib.pyplot.flag

```
flag() -> 'None'
```

Set the colormap to 'flag'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.flatten

```
flatten(seq, scalarp=)
```

Return a generator of flattened nested containers.

For example:

```
>>> from matplotlib.cbook import flatten
>>> l = (('John', ['Hunter']), (1, 23), [[[42, (5, 23)],]])
>>> print(list(flatten(l)))
['John', 'Hunter', 1, 23, 42, 5, 23]
```

By: Composite of Holger Krekel and Luther Blissett

From: <https://code.activestate.com/recipes/121294-simple-generator-for-flattening-nested-containers/>  
and Recipe 1.12 in cookbook

## matplotlib.pyplot.gca

```
gca() -> 'Axes'
```

Get the current Axes.

If there is currently no Axes on this Figure, a new one is created using ``Figure.add_subplot``. (To test whether there is currently an Axes on a Figure, check whether ```figure.axes``` is empty. To test whether there is currently a Figure on the pyplot figure stack, check whether ``pyplot.get_fignums()`` is empty.)

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``Figure.gca``.

## matplotlib.pyplot.gcf

```
gcf() -> 'Figure'
```

Get the current figure.

If there is currently no figure on the pyplot figure stack, a new one is created using `~.pyplot.figure()`. (To test whether there is currently a figure on the pyplot figure stack, check whether `~.pyplot.get_fignums()` is empty.)

## matplotlib.pyplot.gci

```
gci() -> 'ColorizingArtist | None'
```

Get the current colorable artist.

Specifically, returns the current `~.ScalarMappable` instance (`~.Image` created by `imshow` or `figimage`, `~.Collection` created by `pcolor` or `scatter`, etc.), or `*None*` if no such instance has been defined.

The current image is an attribute of the current Axes, or the nearest earlier Axes in the current figure that contains an image.

Notes

-----

Historically, the only colorable artists were images; hence the name ```gci``` (get current image).

## matplotlib.pyplot.get

```
get(obj, *args, **kwargs)
```

Return the value of an `~.Artist`'s `*property*`, or print all of them.

Parameters

-----

`obj` : `~matplotlib.artist.Artist`

The queried artist; e.g., a `~.Line2D`, a `~.Text`, or an `~.axes.Axes`.

`property` : str or None, default: None

If `*property*` is 'somename', this function returns ```obj.get_somename()```.

If it's None (or unset), it `*prints*` all gettable properties from `*obj*`. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

See Also

-----

setp

Notes

-----

.. note::

This is equivalent to ``matplotlib.artist.getp``.

## matplotlib.pyplot.get\_backend

```
get_backend(*, auto_select=True)
```

Return the name of the current backend.

Parameters

-----

`auto_select` : bool, default: True

Whether to trigger backend resolution if no backend has been selected so far. If True, this ensures that a valid backend is returned. If False, this returns None if no backend has been selected so far.

.. versionadded:: 3.10

.. admonition:: Provisional

The `*auto_select*` flag is provisional. It may be changed or removed without prior warning.

See Also

-----

`matplotlib.use`

## matplotlib.pyplot.get\_cmap

```
get_cmap(name: 'Colormap | str | None' = None, lut: 'int | None' = None) -> 'Colormap'
```

Get a colormap instance, defaulting to rc values if `*name*` is None.

Parameters

-----

`name` : `~matplotlib.colors.Colormap`` or str or None, default: None

If a `~matplotlib.colors.Colormap`` instance, it will be returned. Otherwise, the name of a colormap known to Matplotlib, which will be resampled by `*lut*`. The default, None, means `:rc:`image.cmap``.

`lut` : int or None, default: None

If `*name*` is not already a Colormap instance and `*lut*` is not None, the colormap will be resampled to have `*lut*` entries in the lookup table.

Returns

-----

Colormap

## matplotlib.pyplot.get\_current\_fig\_manager

```
get_current_fig_manager() -> 'FigureManagerBase | None'
```

Return the figure manager of the current figure.

The figure manager is a container for the actual backend-dependent window that displays the figure on screen.

If no current figure exists, a new one is created, and its figure manager is returned.

Returns

-----  
`.FigureManagerBase` or backend-dependent subclass thereof

## matplotlib.pyplot.get\_figlabels

```
get_figlabels() -> 'list[Any]'
```

Return a list of existing figure labels.

## matplotlib.pyplot.get\_fignums

```
get_fignums() -> 'list[int]'
```

Return a list of existing figure numbers.

## matplotlib.pyplot.get\_plot\_commands

```
get_plot_commands() -> 'list[str]'
```

[\*Deprecated\*] Get a sorted list of all of the plotting commands.

Notes

-----  
.. deprecated:: 3.7

## matplotlib.pyplot.get\_scale\_names

```
get_scale_names()
```

Return the names of the available scales.

## matplotlib.pyplot.getp

```
getp(obj, *args, **kwargs)
```

Return the value of an `.Artist`'s \*property\*, or print all of them.

Parameters

-----  
obj : `~matplotlib.artist.Artist`  
The queried artist; e.g., a `.Line2D`, a `.Text`, or an `~.axes.Axes`.

property : str or None, default: None

If \*property\* is 'somename', this function returns

```
``obj.get_somename()``.
```

If it's None (or unset), it \*prints\* all gettable properties from \*obj\*. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

See Also

-----

setp

Notes

-----

.. note::

This is equivalent to ``matplotlib.artist.getp``.

## matplotlib.pyplot.ginput

```
ginput(n: 'int' = 1, timeout: 'float' = 30, show_clicks: 'bool' = True, mouse_add:
'MouseButton' = , mouse_pop: 'MouseButton' = , mouse_stop: 'MouseButton' =) ->
'list[tuple[int, int]]'
```

Blocking call to interact with a figure.

Wait until the user clicks \*n\* times on the figure, and return the coordinates of each click in a list.

There are three possible interactions:

- Add a point.
- Remove the most recently added point.
- Stop the interaction and return the points added so far.

The actions are assigned to mouse buttons via the arguments \*mouse\_add\*, \*mouse\_pop\* and \*mouse\_stop\*.

Parameters

-----

n : int, default: 1

Number of mouse clicks to accumulate. If negative, accumulate clicks until the input is terminated manually.

timeout : float, default: 30 seconds

Number of seconds to wait before timing out. If zero or negative will never time out.

show\_clicks : bool, default: True

If True, show a red cross at the location of each click.

mouse\_add : ``MouseButton`` or None, default: ``MouseButton.LEFT``

Mouse button used to add points.

`mouse_pop` : ``MouseButton`` or `None`, default: ``MouseButton.RIGHT``

Mouse button used to remove the most recently added point.

`mouse_stop` : ``MouseButton`` or `None`, default: ``MouseButton.MIDDLE``

Mouse button used to stop input.

Returns

-----

list of tuples

A list of the clicked (x, y) coordinates.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``Figure.ginput``.

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right-clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

## matplotlib.pyplot.gray

```
gray() -> 'None'
```

Set the colormap to 'gray'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.grid

```
grid(visible: 'bool | None' = None, which: "Literal['major', 'minor', 'both']" = 'major', axis: "Literal['both', 'x', 'y']" = 'both', **kwargs) -> 'None'
```

Configure the grid lines.

Parameters

-----

`visible` : bool or `None`, optional

Whether to show the grid lines. If any `*kwargs*` are supplied, it is assumed you want the grid on and `*visible*` will be set to `True`.

If `*visible*` is `*None*` and there are no `*kwargs*`, this toggles the visibility of the lines.

`which` : {'major', 'minor', 'both'}, optional

The grid lines to apply the changes on.

`axis` : {'both', 'x', 'y'}, optional

The axis to apply the changes on.

**\*\*kwargs :** `~matplotlib.lines.Line2D`` properties

Define the line properties of the grid, e.g.::

```
grid(color='r', linestyle='-', linewidth=2)
```

Valid keyword arguments are:

Properties:

**agg\_filter:** a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

**alpha:** float or None

**animated:** bool

**antialiased** or **aa:** bool

**clip\_box:** `~matplotlib.transforms.BboxBase`` or None

**clip\_on:** bool

**clip\_path:** Patch or (Path, Transform) or None

**color** or **c:** `:mpltype:`color``

**dash\_capstyle:** ``.CapStyle`` or {'butt', 'projecting', 'round'}

**dash\_joinstyle:** ``.JoinStyle`` or {'miter', 'round', 'bevel'}

**dashes:** sequence of floats (on/off ink in points) or (None, None)

**data:** (2, N) array or two 1D arrays

**drawstyle** or **ds:** {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

**figure:** `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

**fillstyle:** {'full', 'left', 'right', 'bottom', 'top', 'none'}

**gapcolor:** `:mpltype:`color`` or None

**gid:** str

**in\_layout:** bool

**label:** object

**linestyle** or **ls:** {'-', '--', '-.', ':', ''}, (offset, on-off-seq), ...}

**linewidth** or **lw:** float

**marker:** marker style string, `~.path.Path`` or `~.markers.MarkerStyle``

**markeredgecolor** or **mec:** `:mpltype:`color``

**markeredgewidth** or **mew:** float

**markerfacecolor** or **mfc:** `:mpltype:`color``

**markerfacecoloralt** or **mfcalt:** `:mpltype:`color``

**markersize** or **ms:** float

**markevery:** None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

**mouseover:** bool

**path\_effects:** list of `~.AbstractPathEffect``

**picker:** float or callable[[Artist, Event], tuple[bool, dict]]

**pickradius:** float

**rasterized:** bool

**sketch\_params:** (scale: float, length: float, randomness: float)

**snap:** bool or None

**solid\_capstyle:** ``.CapStyle`` or {'butt', 'projecting', 'round'}

**solid\_joinstyle:** ``.JoinStyle`` or {'miter', 'round', 'bevel'}

**transform:** unknown

**url:** str

**visible:** bool

**xdata:** 1D array

**ydata:** 1D array

**zorder:** float

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.grid``.

The axis is drawn as a unit, so the effective zorder for drawing the grid is determined by the zorder of each axis, not by the zorder of the `.Line2D`` objects comprising the grid. Therefore, to set grid zorder, use `.set_axisbelow`` or, for more control, call the `~.Artist.set_zorder`` method of each axis.

## matplotlib.pyplot.hexbin

```
hexbin(x: 'ArrayLike', y: 'ArrayLike', C: 'ArrayLike | None' = None, *, gridsize: 'int | tuple[int, int]' = 100, bins: "Literal['log'] | int | Sequence[float] | None" = None, xscale: "Literal['linear', 'log']" = 'linear', yscale: "Literal['linear', 'log']" = 'linear', extent: 'tuple[float, float, float, float] | None' = None, cmap: 'str | Colormap | None' = None, norm: 'str | Normalize | None' = None, vmin: 'float | None' = None, vmax: 'float | None' = None, alpha: 'float | None' = None, linewidths: 'float | None' = None, edgecolors: "Literal['face', 'none'] | ColorType" = 'face', reduce_C_function: 'Callable[[np.ndarray | list[float]], float]' = , mincnt: 'int | None' = None, marginals: 'bool' = False, colorizer: 'Colorizer | None' = None, data=None, **kwargs) -> 'PolyCollection'
```

Make a 2D hexagonal binning plot of points `*x*`, `*y*`.

If `*C*` is `*None*`, the value of the hexagon is determined by the number of points in the hexagon. Otherwise, `*C*` specifies values at the coordinate `(x[i], y[i])`. For each hexagon, these values are reduced using `*reduce_C_function*`.

### Parameters

-----

`x, y` : array-like

The data positions. `*x*` and `*y*` must be of the same length.

`C` : array-like, optional

If given, these values are accumulated in the bins. Otherwise, every point has a value of 1. Must be of the same length as `*x*` and `*y*`.

`gridsize` : int or (int, int), default: 100

If a single int, the number of hexagons in the `*x*`-direction.

The number of hexagons in the `*y*`-direction is chosen such that the hexagons are approximately regular.

Alternatively, if a tuple `(*nx*, *ny*)`, the number of hexagons in the `*x*`-direction and the `*y*`-direction. In the `*y*`-direction, counting is done along vertically aligned hexagons, not along the zig-zag chains of hexagons; see the following illustration.

.. plot::

```
import numpy
import matplotlib.pyplot as plt
```

```

np.random.seed(19680801)
n= 300
x = np.random.standard_normal(n)
y = np.random.standard_normal(n)

fig, ax = plt.subplots(figsize=(4, 4))
h = ax.hexbin(x, y, gridsize=(5, 3))
hx, hy = h.get_offsets().T
ax.plot(hx[24::3], hy[24::3], 'ro-')
ax.plot(hx[-3:], hy[-3:], 'ro-')
ax.set_title('gridsize=(5, 3)')
ax.axis('off')

```

To get approximately regular hexagons, choose  
 $n_x = \sqrt{3}, n_y$ .

bins : 'log' or int or sequence, default: None  
 Discretization of the hexagon values.

- If *\*None\**, no binning is applied; the color of each hexagon directly corresponds to its count value.
- If 'log', use a logarithmic scale for the colormap. Internally,  $\log_{10}(i+1)$  is used to determine the hexagon color. This is equivalent to `norm=LogNorm()`.
- If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.
- If a sequence of values, the values of the lower bound of the bins to be used.

xscale : {'linear', 'log'}, default: 'linear'  
 Use a linear or log10 scale on the horizontal axis.

yscale : {'linear', 'log'}, default: 'linear'  
 Use a linear or log10 scale on the vertical axis.

mincnt : int >= 0, default: *\*None\**  
 If not *\*None\**, only display cells with at least *\*mincnt\** number of points in the cell.

marginals : bool, default: *\*False\**  
 If marginals is *\*True\**, plot the marginal density as colormapped rectangles along the bottom of the x-axis and left of the y-axis.

extent : 4-tuple of float, default: *\*None\**  
 The limits of the bins (xmin, xmax, ymin, ymax).  
 The default assigns the limits based on *\*gridsize\**, *\*x\**, *\*y\**, *\*xscale\** and *\*yscale\**.

If *\*xscale\** or *\*yscale\** is set to 'log', the limits are expected to be the exponent for a power of 10. E.g. for x-limits of 1 and 50 in 'linear' scale and y-limits of 10 and 1000 in 'log' scale, enter (1, 50, 1, 3).

## Returns

-----

`~matplotlib.collections.PolyCollection``

A `~.PolyCollection`` defining the hexagonal bins.

- `~.PolyCollection.get_offsets`` contains a Mx2 array containing the x, y positions of the M hexagon centers in data coordinates.

- `~.PolyCollection.get_array`` contains the values of the M hexagons.

If *\*marginals\** is *\*True\**, horizontal bar and vertical bar (both `PolyCollections`) will be attached to the return collection as attributes *\*hbar\** and *\*vbar\**.

## Other Parameters

-----

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The `Colormap` instance or registered colormap name used to map scalar data to colors.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *\*cmap\**. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `~.Normalize`` or one of its subclasses

(see `:ref:`colormapnorms``).

- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``.

In that case, a suitable `~.Normalize`` subclass is dynamically generated and instantiated.

`vmin`, `vmax` : float, optional

When using scalar data and no explicit *\*norm\**, *\*vmin\** and *\*vmax\** define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *\*vmin\*/vmax\** when a *\*norm\** instance is given (but using a `~str`` *\*norm\** name together with *\*vmin\*/vmax\** is acceptable).

`alpha` : float between 0 and 1, optional

The alpha blending value, between 0 (transparent) and 1 (opaque).

`linewidths` : float, default: *\*None\**

If *\*None\**, defaults to `:rc:`patch.linewidth``.

`edgecolors` : {'face', 'none', *\*None\**} or color, default: 'face'

The color of the hexagon edges. Possible values are:

- 'face': Draw the edges in the same color as the fill color.

- 'none': No edges are drawn. This can sometimes lead to unsightly unpainted pixels between the hexagons.

- *\*None\**: Draw outlines in the default color.

- An explicit color.

reduce\_C\_function : callable, default: ``numpy.mean``  
The function to aggregate `*C*` within the bins. It is ignored if `*C*` is not given. This must have the signature::

```
def reduce_C_function(C: array) -> float
```

Commonly used functions are:

- ``numpy.mean``: average of the points
- ``numpy.sum``: integral of the point values
- ``numpy.amax``: value taken from the largest point

By default will only reduce cells with at least 1 point because some reduction functions (such as ``numpy.amax``) will error/warn with empty input. Changing `*mincnt*` will adjust the cutoff, and if set to 0 will pass empty input to the reduction function.

colorizer : ``~matplotlib.colorbar.Colorizer`` or None, default: None  
The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

data : indexable object, optional

If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

`*x*, *y*, *C*`

`**kwargs` : ``~matplotlib.collections.PolyCollection`` properties  
All other keyword arguments are passed on to ``~matplotlib.collections.PolyCollection``:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: array-like or float or None

`animated`: bool

`antialiased` or `aa` or `antialiaseds`: bool or list of bools

`array`: array-like or None

`capstyle`: ``~matplotlib.collections.PolyCollection`` or `{'butt', 'projecting', 'round'}`

`clim`: (vmin: float, vmax: float)

`clip_box`: ``~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`cmap`: ``~matplotlib.colors.Colormap`` or str or None

`color`: `:mpltype:`color`` or list of RGBA tuples

`edgecolor` or `ec` or `edgecolors`: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'

`facecolor` or `facecolors` or `fc`: `:mpltype:`color`` or list of `:mpltype:`color``

`figure`: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

`gid`: str

`hatch`: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`

`hatch_linewidth`: unknown

`in_layout`: bool

`joinstyle`: ``~matplotlib.collections.PolyCollection`` or `{'miter', 'round', 'bevel'}`

`label`: object

`linestyle` or `dashes` or `linestyles` or `ls`: str or tuple or list thereof

linewidth or linewidths or lw: float or list of floats  
 mouseover: bool  
 norm: ``~.Normalize`` or str or None  
 offset\_transform or transOffset: ``~.Transform``  
 offsets: (N, 2) or (2,) array-like  
 path\_effects: list of ``~.AbstractPathEffect``  
 paths: list of array-like  
 picker: None or bool or float or callable  
 pickradius: float  
 rasterized: bool  
 sizes: ``numpy.ndarray`` or None  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 transform: ``~matplotlib.transforms.Transform``  
 url: str  
 urls: list of str or None  
 verts: list of array-like  
 verts\_and\_codes: unknown  
 visible: bool  
 zorder: float

See Also

-----  
 hist2d : 2D histogram rectangular bins

Notes

-----  
 .. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``~.axes.Axes.hexbin``.

## matplotlib.pyplot.hist

```

hist(x: 'ArrayLike | Sequence[ArrayLike]', bins: 'int | Sequence[float] | str | None'
 = None, *, range: 'tuple[float, float] | None' = None, density: 'bool' = False,
 weights: 'ArrayLike | None' = None, cumulative: 'bool | float' = False, bottom:
 'ArrayLike | float | None' = None, histtype: "Literal['bar', 'barstacked', 'step',
 'stepfilled']" = 'bar', align: "Literal['left', 'mid', 'right']" = 'mid', orientation:
 "Literal['vertical', 'horizontal']" = 'vertical', rwidth: 'float | None' = None, log:
 'bool' = False, color: 'ColorType | Sequence[ColorType] | None' = None, label: 'str |
 Sequence[str] | None' = None, stacked: 'bool' = False, data=None, **kwargs) ->
 'tuple[np.ndarray | list[np.ndarray], np.ndarray, BarContainer | Polygon |
 list[BarContainer | Polygon]]'

```

Compute and plot a histogram.

This method uses ``numpy.histogram`` to bin the data in `*x*` and count the number of values in each bin, then draws the distribution either as a ``~.BarContainer`` or ``~.Polygon``. The `*bins*`, `*range*`, `*density*`, and `*weights*` parameters are forwarded to ``numpy.histogram``.

If the data has already been binned and counted, use ``~.bar`` or ``~.stairs`` to plot the distribution::

```
counts, bins = np.histogram(x)
```

```
plt.stairs(counts, bins)
```

Alternatively, plot pre-computed bins and counts using ``hist()`` by treating each bin as a single point with a weight equal to its count::

```
plt.hist(bins[:-1], bins, weights=counts)
```

The data input *\*x\** can be a singular array, a list of datasets of potentially different lengths (*[\*x0\*, \*x1\*, ...]*), or a 2D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form. If the input is an array, then the return value is a tuple (*\*n\*, \*bins\*, \*patches\**); if the input is a sequence of arrays, then the return value is a tuple (*[\*n0\*, \*n1\*, ...], \*bins\*, [\*patches0\*, \*patches1\*, ...]*).

Masked arrays are not supported.

#### Parameters

-----

*x* : (n,) array or sequence of (n,) arrays

Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.

*bins* : int or sequence or str, default: :rc:`hist.bins`

If *\*bins\** is an integer, it defines the number of equal-width bins in the range.

If *\*bins\** is a sequence, it defines the bin edges, including the left edge of the first bin and the right edge of the last bin; in this case, bins may be unequally spaced. All but the last (righthand-most) bin is half-open. In other words, if *\*bins\** is::

```
[1, 2, 3, 4]
```

then the first bin is ``[1, 2)`` (including 1, but excluding 2) and the second ``[2, 3)``. The last bin, however, is ``[3, 4]``, which *\*includes\** 4.

If *\*bins\** is a string, it is one of the binning strategies supported by `numpy.histogram_bin_edges`: 'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'.

*range* : tuple or None, default: None

The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, *\*range\** is ``(x.min(), x.max())``. Range has no effect if *\*bins\** is a sequence.

If *\*bins\** is a sequence or *\*range\** is specified, autoscaling is based on the specified bin range instead of the range of *x*.

*density* : bool, default: False

If ``True``, draw and return a probability density: each bin will display the bin's raw count divided by the total number of counts *\*and the bin width\**

(`density = counts / (sum(counts) \* np.diff(bins))`),  
so that the area under the histogram integrates to 1  
(`np.sum(density \* np.diff(bins)) == 1`).

If *\*stacked\** is also `True`, the sum of the histograms is normalized to 1.

*weights* : (n,) array-like or None, default: None  
An array of weights, of the same shape as *\*x\**. Each value in *\*x\** only contributes its associated weight towards the bin count (instead of 1). If *\*density\** is `True`, the weights are normalized, so that the integral of the density over the range remains 1.

*cumulative* : bool or -1, default: False  
If `True`, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints.

If *\*density\** is also `True` then the histogram is normalized such that the last bin equals 1.

If *\*cumulative\** is a number less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if *\*density\** is also `True`, then the histogram is normalized such that the first bin equals 1.

*bottom* : array-like or float, default: 0  
Location of the bottom of each bin, i.e. bins are drawn from `bottom` to `bottom + hist(x, bins)`. If a scalar, the bottom of each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

*histtype* : {'bar', 'barstacked', 'step', 'stepfilled'}, default: 'bar'  
The type of histogram to draw.

- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

*align* : {'left', 'mid', 'right'}, default: 'mid'  
The horizontal alignment of the histogram bars.

- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

*orientation* : {'vertical', 'horizontal'}, default: 'vertical'  
If 'horizontal', `~.Axes.barh` will be used for bar-type histograms and the *\*bottom\** kwarg will be the left edges.

`rwidth` : float or None, default: None  
The relative width of the bars as a fraction of the bin width. If  
``None``, automatically compute the width.

Ignored if `*histtype*` is 'step' or 'stepfilled'.

`log` : bool, default: False  
If ``True``, the histogram axis will be set to a log scale.

`color` : :mpltype:`color` or list of :mpltype:`color` or None, default: None  
Color or sequence of colors, one per dataset. Default (``None``)  
uses the standard line color sequence.

`label` : str or list of str, optional  
String, or sequence of strings to match multiple datasets. Bar  
charts yield multiple patches per dataset, but only the first gets  
the label, so that `~.Axes.legend`` will work as expected.

`stacked` : bool, default: False  
If ``True``, multiple data are stacked on top of each other If  
``False`` multiple data are arranged side by side if `histtype` is  
'bar' or on top of each other if `histtype` is 'step'

#### Returns

-----  
`n` : array or list of arrays  
The values of the histogram bins. See `*density*` and `*weights*` for a  
description of the possible semantics. If input `*x*` is an array,  
then this is an array of length `*nbins*`. If input is a sequence of  
arrays ```[data1, data2, ...]```, then this is a list of arrays with  
the values of the histograms for each of the arrays in the same  
order. The dtype of the array `*n*` (or of its element arrays) will  
always be float even if no weighting or normalization is used.

`bins` : array  
The edges of the bins. Length `nbins + 1` (`nbins` left edges and right  
edge of last bin). Always a single array even when multiple data  
sets are passed in.

`patches` : `~.BarContainer`` or list of a single `~.Polygon`` or list of such objects  
Container of individual artists used to create the histogram  
or list of such containers if there are multiple input datasets.

#### Other Parameters

-----  
`data` : indexable object, optional  
If given, the following parameters also accept a string ```s```, which is  
interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*weights*`

`**kwargs`  
`~matplotlib.patches.Patch`` properties. The following properties  
additionally accept a sequence of values corresponding to the  
datasets in `*x*`:

\*edgecolor\*, \*facecolor\*, \*linewidth\*, \*linestyle\*, \*hatch\*.

.. versionadded:: 3.10

Allowing sequences of values in above listed Patch properties.

See Also

-----

hist2d : 2D histogram with rectangular bins

hexbin : 2D histogram with hexagonal bins

stairs : Plot a pre-computed histogram

bar : Plot a pre-computed histogram

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.hist`.

For large numbers of bins (>1000), plotting can be significantly accelerated by using `~.Axes.stairs` to plot a pre-computed histogram (``plt.stairs(np.histogram(data))``), or by setting `*histtype*` to `'step'` or `'stepfilled'` rather than `'bar'` or `'barstacked'`.

## matplotlib.pyplot.hist2d

```
hist2d(x: 'ArrayLike', y: 'ArrayLike', bins: 'None | int | tuple[int, int] | ArrayLike | tuple[ArrayLike, ArrayLike]' = 10, *, range: 'ArrayLike | None' = None, density: 'bool' = False, weights: 'ArrayLike | None' = None, cmin: 'float | None' = None, cmax: 'float | None' = None, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray, np.ndarray, QuadMesh]'
```

Make a 2D histogram plot.

Parameters

-----

x, y : array-like, shape (n, )

Input values

bins : None or int or [int, int] or array-like or [array, array]

The bin specification:

- If int, the number of bins for the two dimensions (``nx = ny = bins``).
- If ```[int, int]```, the number of bins in each dimension (``nx, ny = bins``).
- If array-like, the bin edges for the two dimensions (``x_edges = y_edges = bins``).
- If ```[array, array]```, the bin edges in each dimension (``x_edges, y_edges = bins``).

The default value is 10.

range : array-like shape(2, 2), optional

The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): ``[[xmin, xmax], [ymin, ymax]]``. All values outside of this range will be considered outliers and not tallied in the histogram.

density : bool, default: False

Normalize histogram. See the documentation for the `*density*` parameter of `~.Axes.hist`` for more details.

weights : array-like, shape (n, ), optional

An array of values `w_i` weighing each sample `(x_i, y_i)`.

cmin, cmax : float, default: None

All bins that has count less than `*cmin*` or more than `*cmax*` will not be displayed (set to NaN before passing to `~.Axes.pcolormesh``) and these count values in the return value count histogram will also be set to nan upon return.

#### Returns

-----

h : 2D array

The bi-dimensional histogram of samples x and y. Values in x are histogrammed along the first dimension and values in y are histogrammed along the second dimension.

xedges : 1D array

The bin edges along the x-axis.

yedges : 1D array

The bin edges along the y-axis.

image : `~.matplotlib.collections.QuadMesh``

#### Other Parameters

-----

cmap : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

norm : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses

(see `:ref:`colormapnorms``).

- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``.

In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

vmin, vmax : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/*vmax*` when a `*norm*` instance is given (but using a ``str`` `*norm*`

name together with `*vmin*/vmax` is acceptable).

`colorizer` : `~matplotlib.colorbar.Colorizer`` or `None`, default: `None`  
The `Colorizer` object used to map color to data. If `None`, a `Colorizer` object is created from a `*norm*` and `*cmap*`.

`alpha` : ```0 <= scalar <= 1``` or ```None```, optional  
The alpha blending value.

`data` : indexable object, optional  
If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*y*`, `*weights*`

`**kwargs`  
Additional parameters are passed along to the  
`~.Axes.pcolormesh`` method and `~matplotlib.collections.QuadMesh`` constructor.

See Also

-----  
`hist` : 1D histogram plotting  
`hexbin` : 2D histogram with hexagonal bins

Notes

-----  
.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.hist2d``.

- Currently ```hist2d``` calculates its own axis limits, and any limits previously set are ignored.
- Rendering the histogram with a logarithmic color scale is accomplished by passing a `~.colors.LogNorm`` instance to the `*norm*` keyword argument. Likewise, power-law normalization (similar in effect to gamma correction) can be accomplished with `~.colors.PowerNorm``.

## matplotlib.pyplot.hlines

```
hlines(y: 'float | ArrayLike', xmin: 'float | ArrayLike', xmax: 'float | ArrayLike',
 colors: 'ColorType | Sequence[ColorType] | None' = None, linestyle: 'LineStyleType' =
 'solid', *, label: 'str' = '', data=None, **kwargs) -> 'LineCollection'
```

Plot horizontal lines at each `*y*` from `*xmin*` to `*xmax*`.

Parameters

-----  
`y` : float or array-like  
y-indexes where to plot the lines.

`xmin`, `xmax` : float or array-like  
Respective beginning and end of each line. If scalars are

provided, all lines will have the same length.

colors : :mpltype:`color` or list of color , default: :rc:`lines.color`

linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, default: 'solid'

label : str, default: ''

Returns

-----  
`~matplotlib.collections.LineCollection`

Other Parameters

-----  
data : indexable object, optional

If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

\*y\*, \*xmin\*, \*xmax\*, \*colors\*

\*\*kwargs : `~matplotlib.collections.LineCollection` properties.

See Also

-----  
vlines : vertical lines

axhline : horizontal line across the Axes

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.hlines``.

## matplotlib.pyplot.hot

```
hot() -> 'None'
```

Set the colormap to 'hot'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

## matplotlib.pyplot.hsv

```
hsv() -> 'None'
```

Set the colormap to 'hsv'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

## matplotlib.pyplot.imread

```
imread(fname: 'str | pathlib.Path | BinaryIO', format: 'str | None' = None) ->
'np.ndarray'
```

Read an image from a file into an array.

.. note::

This function exists for historical reasons. It is recommended to use `PIL.Image.open`` instead for loading images.

#### Parameters

-----

fname : str or file-like

The image file to read: a filename, a URL or a file-like object opened in read-binary mode.

Passing a URL is deprecated. Please open the URL for reading and pass the result to Pillow, e.g. with `np.array(PIL.Image.open(urllib.request.urlopen(url)))``.

format : str, optional

The image file format assumed for reading the data. The image is loaded as a PNG file if `*format*` is set to "png", if `*fname*` is a path or opened file with a ".png" extension, or if it is a URL. In all other cases, `*format*` is ignored and the format is auto-detected by `PIL.Image.open``.

#### Returns

-----

``numpy.array``

The image data. The returned array has shape

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

PNG images are returned as float arrays (0-1). All other formats are returned as int arrays, with a bit depth determined by the file's contents.

#### Notes

-----

.. note::

This is equivalent to ``matplotlib.image.imread``.

## matplotlib.pyplot.imshow

```
imshow(fname: 'str | os.PathLike | BinaryIO', arr: 'ArrayLike', **kwargs) -> 'None'
```

Colormap and save an array as an image file.

RGB(A) images are passed through. Single channel images will be colormapped according to `*cmap*` and `*norm*`.

.. note::

If you want to save a single channel image as gray scale please use an image I/O library (such as pillow, tiff file, or imageio) directly.

#### Parameters

-----

`fname` : str or path-like or file-like

A path or a file-like object to store the image in.

If `*format*` is not set, then the output format is inferred from the extension of `*fname*`, if any, and from `:rc:`savefig.format`` otherwise.

If `*format*` is set, it determines the output format.

`arr` : array-like

The image data. Accepts NumPy arrays or sequences

(e.g., lists or tuples). The shape can be one of

MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).

`vmin`, `vmax` : float, optional

`*vmin*` and `*vmax*` set the color scaling for the image by fixing the

values that map to the colormap color limits. If either `*vmin*`

or `*vmax*` is None, that limit is determined from the `*arr*`

min/max value.

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

A Colormap instance or registered colormap name. The colormap maps scalar data to colors. It is ignored for RGB(A) data.

`format` : str, optional

The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this

is unset is documented under `*fname*`.

`origin` : {'upper', 'lower'}, default: `:rc:`image.origin``

Indicates whether the ``(0, 0)`` index of the array is in the upper left or lower left corner of the Axes.

`dpi` : float

The DPI to store in the metadata of the file. This does not affect the

resolution of the output image. Depending on file format, this may be rounded to the nearest integer.

`metadata` : dict, optional

Metadata in the image file. The supported keys depend on the output format, see the documentation of the respective backends for more information.

Currently only supported for "png", "pdf", "ps", "eps", and "svg".

`pil_kwargs` : dict, optional

Keyword arguments passed to ``PIL.Image.Image.save``. If the 'pnginfo'

key is present, it completely overrides `*metadata*`, including the default 'Software' key.

#### Notes

-----

.. note::

This is equivalent to ``matplotlib.image.imsave``.

## matplotlib.pyplot.imshow

---

```
imshow(X: 'ArrayLike | PIL.Image.Image', cmap: 'str | Colormap | None' = None, norm:
'str | Normalize | None' = None, *, aspect: "Literal['equal', 'auto'] | float | None"
= None, interpolation: 'str | None' = None, alpha: 'float | ArrayLike | None' = None,
vmin: 'float | None' = None, vmax: 'float | None' = None, colorizer: 'Colorizer |
None' = None, origin: "Literal['upper', 'lower'] | None" = None, extent: 'tuple[float,
float, float, float] | None' = None, interpolation_stage: "Literal['data', 'rgba',
'auto'] | None" = None, filternorm: 'bool' = True, filterrad: 'float' = 4.0, resample:
'bool | None' = None, url: 'str | None' = None, data=None, **kwargs) -> 'AxesImage'
```

Display data as an image, i.e., on a 2D regular raster.

The input may either be actual RGB(A) data, or 2D scalar data, which will be rendered as a pseudocolor image. For displaying a grayscale image, set up the colormapping using the parameters

```cmap='gray', vmin=0, vmax=255```.

The number of pixels used to render an image is set by the Axes size and the figure `*dpi*`. This can lead to aliasing artifacts when the image is resampled, because the displayed image size will usually not match the size of `*X*` (see

:doc:`gallery/images_contours_and_fields/image_antialiasing`).

The resampling can be controlled via the `*interpolation*` parameter and/or `:rc:`image.interpolation``.

Parameters

X : array-like or PIL image

The image data. Supported array shapes are:

- (M, N): an image with scalar data. The values are mapped to colors using normalization and a colormap. See parameters `*norm*`, `*cmap*`, `*vmin*`, `*vmax*`.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the image.

Out-of-range RGB(A) values are clipped.

cmap : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*X*` is RGB(A).

norm : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a

list of available scales, call `matplotlib.scale.get_scale_names()`.
In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*X*` is RGB(A).

`vmin, vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

This parameter is ignored if `*X*` is RGB(A).

`colorizer` : `~matplotlib.colorbar.Colorizer`` or None, default: None

The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

This parameter is ignored if `*X*` is RGB(A).

`aspect` : {'equal', 'auto'} or float or None, default: None

The aspect ratio of the Axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling `.Axes.set_aspect``. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square (unless pixel sizes are explicitly made non-square in data coordinates using `*extent*`).

- 'auto': The Axes is kept fixed and the aspect is adjusted so that the data fit in the Axes. In general, this will result in non-square pixels.

Normally, None (the default) means to use `:rc:`image.aspect``. However, if the image uses a transform that does not contain the axes data transform, then None means to not modify the axes aspect at all (in that case, directly call `.Axes.set_aspect`` if desired).

`interpolation` : str, default: `:rc:`image.interpolation``

The interpolation method used.

Supported values are 'none', 'auto', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'.

The data `*X*` is resampled to the pixel size of the image on the figure canvas, using the interpolation method to either up- or downsample the data.

If `*interpolation*` is 'none', then for the ps, pdf, and svg backends no down- or upsampling occurs, and the image data is

passed to the backend as a native image. Note that different ps, pdf, and svg viewers may display these raw pixels differently. On other backends, 'none' is the same as 'nearest'.

If **interpolation** is the default 'auto', then 'nearest' interpolation is used if the image is upsampled by more than a factor of three (i.e. the number of display pixels is at least three times the size of the data array). If the upsampling rate is smaller than 3, or the image is downsampled, then 'hanning' interpolation is used to act as an anti-aliasing filter, unless the image happens to be upsampled by exactly a factor of two or one.

See

`:doc:`gallery/images_contours_and_fields/interpolation_methods`` for an overview of the supported interpolation methods, and `:doc:`gallery/images_contours_and_fields/image_antialiasing`` for a discussion of image antialiasing.

Some interpolation methods require an additional radius parameter, which can be set by **filterrad**. Additionally, the antigrain image resize filter is controlled by the parameter **filternorm**.

`interpolation_stage` : {'auto', 'data', 'rgba'}, default: 'auto'

Supported values:

- 'data': Interpolation is carried out on the data provided by the user. This is useful if interpolating between pixels during upsampling.
- 'rgba': The interpolation is carried out in RGBA-space after the color-mapping has been applied. This is useful if downsampling and combining pixels visually.
- 'auto': Select a suitable interpolation stage automatically. This uses 'rgba' when downsampling, or upsampling at a rate less than 3, and 'data' when upsampling at a higher rate.

See `:doc:`gallery/images_contours_and_fields/image_antialiasing`` for a discussion of image antialiasing.

`alpha` : float or array-like, optional

The alpha blending value, between 0 (transparent) and 1 (opaque).

If **alpha** is an array, the alpha blending values are applied pixel by pixel, and **alpha** must have the same shape as **X**.

`origin` : {'upper', 'lower'}, default: `:rc:`image.origin``

Place the [0, 0] index of the array in the upper left or lower left corner of the Axes. The convention (the default) 'upper' is typically used for matrices and images.

Note that the vertical axis points upward for 'lower' but downward for 'upper'.

See the `:ref:`imshow_extent`` tutorial for examples and a more detailed description.

`extent` : floats (left, right, bottom, top), optional

The bounding box in data coordinates that the image will fill.

These values may be unitful and match the units of the Axes.
The image is stretched individually along x and y to fill the box.

The default extent is determined by the following conditions.
Pixels have unit size in data coordinates. Their centers are on integer coordinates, and their center coordinates range from 0 to columns-1 horizontally and from 0 to rows-1 vertically.

Note that the direction of the vertical axis and thus the default values for top and bottom depend on `*origin*`:

- For ```origin == 'upper'``` the default is ```(-0.5, numcols-0.5, numrows-0.5, -0.5)```.
- For ```origin == 'lower'``` the default is ```(-0.5, numcols-0.5, -0.5, numrows-0.5)```.

See the :ref:`imshow_extent` tutorial for examples and a more detailed description.

`filternorm` : bool, default: True

A parameter for the antigrain image resize filter (see the antigrain documentation). If `*filternorm*` is set, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

`filterrad` : float > 0, default: 4.0

The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

`resample` : bool, default: :rc:`image.resample`

When `*True*`, use a full resampling method. When `*False*`, only resample when the output image is larger than the input image.

`url` : str, optional

Set the url of the created ``.AxesImage``. See ``.Artist.set_url``.

Returns

`~matplotlib.image.AxesImage``

Other Parameters

`data` : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

`**kwargs` : `~matplotlib.artist.Artist`` properties

These parameters are passed on to the constructor of the ``.AxesImage`` artist.

See Also

imshow : Plot a matrix or an array as an image.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.imshow``.

Unless `*extent*` is used, pixel centers will be located at integer coordinates. In other words: the origin will coincide with the center of pixel (0, 0).

There are two common representations for RGB images with an alpha channel:

- Straight (unassociated) alpha: R, G, and B channels represent the color of the pixel, disregarding its opacity.
- Premultiplied (associated) alpha: R, G, and B channels represent the color of the pixel, adjusted for its opacity by multiplication.

`~matplotlib.pyplot.imshow`` expects RGB images adopting the straight (unassociated) alpha representation.

matplotlib.pyplot.inferno

```
inferno() -> 'None'
```

Set the colormap to 'inferno'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

matplotlib.pyplot.install_repl_displayhook

```
install_repl_displayhook() -> 'None'
```

Connect to the display hook of the current shell.

The display hook gets called when the read-evaluate-print-loop (REPL) of the shell has finished the execution of a command. We use this callback to be able to automatically update a figure in interactive mode.

This works both with IPython and with vanilla python shells.

matplotlib.pyplot.interactive

```
interactive(b)
```

Set whether to redraw after every plotting command (e.g. `.pyplot.xlabel``).

matplotlib.pyplot.ioff

```
ioff() -> 'AbstractContextManager'
```

Disable interactive mode.

See ``pyplot.isinteractive`` for more details.

See Also

`ion` : Enable interactive mode.

`isinteractive` : Whether interactive mode is enabled.

`show` : Show all figures (and maybe block).

`pause` : Show all figures, and block for a time.

Notes

For a temporary change, this can be used as a context manager::

```
# if interactive mode is on
# then figures will be shown on creation
plt.ion()
# This figure will be shown immediately
fig = plt.figure()
```

```
with plt.ioff():
# interactive mode will be off
# figures will not automatically be shown
fig2 = plt.figure()
# ...
```

To enable optional usage as a context manager, this function returns a context manager object, which is not intended to be stored or accessed by the user.

matplotlib.pyplot.ion

```
ion() -> 'AbstractContextManager'
```

Enable interactive mode.

See ``pyplot.isinteractive`` for more details.

See Also

`ioff` : Disable interactive mode.

`isinteractive` : Whether interactive mode is enabled.

`show` : Show all figures (and maybe block).

`pause` : Show all figures, and block for a time.

Notes

For a temporary change, this can be used as a context manager::

```
# if interactive mode is off
# then figures will not be shown on creation
plt.ioff()
# This figure will not be shown immediately
fig = plt.figure()
```

```
with plt.ion():
# interactive mode will be on
# figures will automatically be shown
fig2 = plt.figure()
# ...
```

To enable optional usage as a context manager, this function returns a context manager object, which is not intended to be stored or accessed by the user.

matplotlib.pyplot.isinteractive

```
isinteractive() -> 'bool'
```

Return whether plots are updated after every plotting command.

The interactive mode is mainly useful if you build plots from the command line and want to see the effect of each command while you are building the figure.

In interactive mode:

- newly created figures will be shown immediately;
- figures will automatically redraw on change;
- `plt.show` will not block by default.

In non-interactive mode:

- newly created figures and changes to figures will not be reflected until explicitly asked to be;
- `plt.show` will block by default.

See Also

`ion` : Enable interactive mode.

`ioff` : Disable interactive mode.

`show` : Show all figures (and maybe block).

`pause` : Show all figures, and block for a time.

matplotlib.pyplot.jet

```
jet() -> 'None'
```

Set the colormap to 'jet'.

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

matplotlib.pyplot.legend

```
legend(*args, **kwargs) -> 'Legend'
```

Place a legend on the Axes.

Call signatures::

```
legend()  
legend(handles, labels)  
legend(handles=handles)  
legend(labels)
```

The call signatures correspond to the following different ways to use this method:

****1. Automatic detection of elements to be shown in the legend****

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the :meth:`~.Artist.set_label` method on the artist::

```
ax.plot([1, 2, 3], label='Inline label')  
ax.legend()
```

or::

```
line, = ax.plot([1, 2, 3])  
line.set_label('Label via method')  
ax.legend()
```

.. note::

Specific artists can be excluded from the automatic legend element selection by using a label starting with an underscore, "". A string starting with an underscore is the default label for all artists, so calling `.Axes.legend` without any arguments and without setting the labels manually will result in a `UserWarning` and an empty legend being drawn.

****2. Explicitly listing the artists and labels in the legend****

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively::

```
ax.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

****3. Explicitly listing the artists in the legend****

This is similar to 2, but the labels are taken from the artists' label properties. Example::

```
line1, = ax.plot([1, 2, 3], label='label1')  
line2, = ax.plot([1, 2, 3], label='label2')  
ax.legend(handles=[line1, line2])
```

****4. Labeling existing plot elements****

.. admonition:: Discouraged

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

To make a legend for all artists on an Axes, call this function with an iterable of strings, one for each legend item. For example::

```
ax.plot([1, 2, 3])
ax.plot([5, 6, 7])
ax.legend(['First line', 'Second line'])
```

Parameters

handles : list of (`.Artist`` or tuple of `.Artist``), optional
A list of Artists (lines, patches) to be added to the legend.
Use this together with `*labels*`, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

If an entry contains a tuple, then the legend handler for all Artists in the tuple will be placed alongside a single label.

labels : list of str, optional
A list of labels to show next to the artists.
Use this together with `*handles*`, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

Returns

~~~~~  
`~matplotlib.legend.Legend``

#### Other Parameters

~~~~~  
loc : str or pair of floats, default: `:rc:`legend.loc``
The location of the legend.

The strings ```'upper left```, ```'upper right```, ```'lower left```, ```'lower right``` place the legend at the corresponding corner of the axes.

The strings ```'upper center```, ```'lower center```, ```'center left```, ```'center right``` place the legend at the center of the corresponding edge of the axes.

The string `''center''` places the legend at the center of the axes.

The string `''best''` places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes coordinates (in which case `*bbox_to_anchor*` will be ignored).

For back-compatibility, `''center right''` (but no other location) can also be spelled `''right''`, and each "string" location can also be given as a numeric value:

```
=====
Location String Location Code
=====
'best' (Axes only) 0
'upper right' 1
'upper left' 2
'lower left' 3
'lower right' 4
'right' 5
'center left' 6
'center right' 7
'lower center' 8
'upper center' 9
'center' 10
=====
```

`bbox_to_anchor`: ``BboxBase``, 2-tuple, or 4-tuple of floats
Box that is used to position the legend in conjunction with `*loc*`.
Defaults to ```axes.bbox``` (if called as a method to ``Axes.legend``) or ```figure.bbox``` (if ```figure.legend```). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by `*bbox_transform*`, with the default transform Axes or Figure coordinates, depending on which ```legend``` is called.

If a 4-tuple or ``BboxBase`` is given, then it specifies the bbox ```(x, y, width, height)``` that the legend is placed in.
To put the legend in the best location in the bottom right quadrant of the Axes (or figure)::

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple ```(x, y)``` places the corner of the legend specified by `*loc*` at `x, y`. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used::

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

`ncols` : int, default: 1

The number of columns that the legend has.

For backward compatibility, the spelling `*ncol*` is also supported but it is discouraged. If both are given, `*ncols*` takes precedence.

`prop` : None or `~matplotlib.font_manager.FontProperties`` or dict

The font properties of the legend. If None (default), the current

`:data:`matplotlib.rcParams`` will be used.

`fontsize` : int or `{'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}`

The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if `*prop*` is not specified.

`labelcolor` : str or list, default: `:rc:`legend.labelcolor``

The color of the text in the legend. Either a valid color string

(for example, 'red'), or a list of color strings. The labelcolor can

also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using `:rc:`legend.labelcolor``. If None, use `:rc:`text.color``.

`numpoints` : int, default: `:rc:`legend.numpoints``

The number of marker points in the legend when creating a legend entry for a `.Line2D`` (line).

`scatterpoints` : int, default: `:rc:`legend.scatterpoints``

The number of marker points in the legend when creating

a legend entry for a `.PathCollection`` (scatter plot).

`scatteryoffsets` : iterable of floats, default: ```[0.375, 0.5, 0.3125]```

The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to ```[0.5]```.

`markerscale` : float, default: `:rc:`legend.markerscale``

The relative size of legend markers compared to the originally drawn ones.

`markerfirst` : bool, default: True

If `*True*`, legend marker is placed to the left of the legend label.

If `*False*`, legend marker is placed to the right of the legend label.

`reverse` : bool, default: False

If `*True*`, the legend labels are displayed in reverse order from the input.

If `*False*`, the legend labels are displayed in the same order as the input.

.. versionadded:: 3.7

`frameon` : bool, default: `:rc:`legend.frameon``

Whether the legend should be drawn on a patch (frame).

`fancybox` : bool, default: `:rc:`legend.fancybox``

Whether round edges should be enabled around the ``FancyBboxPatch`` which makes up the legend's background.

`shadow` : None, bool or dict, default: `:rc:`legend.shadow``

Whether to draw a shadow behind the legend.

The shadow can be configured using ``Patch`` keywords.

Customization via `:rc:`legend.shadow`` is currently not supported.

`framealpha` : float, default: `:rc:`legend.framealpha``

The alpha transparency of the legend's background.

If `*shadow*` is activated and `*framealpha*` is ```None```, the default value is ignored.

`facecolor` : "inherit" or color, default: `:rc:`legend.facecolor``

The legend's background color.

If ```"inherit"```, use `:rc:`axes.facecolor``.

`edgecolor` : "inherit" or color, default: `:rc:`legend.edgecolor``

The legend's background patch edge color.

If ```"inherit"```, use `:rc:`axes.edgecolor``.

`mode` : {"expand", None}

If `*mode*` is set to ```"expand"``` the legend will be horizontally expanded to fill the Axes area (or `*bbox_to_anchor*` if defines the legend's size).

`bbox_transform` : None or `~matplotlib.transforms.Transform``

The transform for the bounding box (`*bbox_to_anchor*`). For a value of ```None``` (default) the Axes'

`:data:~matplotlib.axes.Axes.transAxes`` transform will be used.

`title` : str or None

The legend's title. Default is no title (```None```).

`title_fontproperties` : None or `~matplotlib.font_manager.FontProperties`` or dict

The font properties of the legend's title. If None (default), the

`*title_fontsize*` argument will be used if present; if `*title_fontsize*` is also None, the current `:rc:`legend.title_fontsize`` will be used.

`title_fontsize` : int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default:

`:rc:`legend.title_fontsize``

The font size of the legend's title.

Note: This cannot be combined with `*title_fontproperties*`. If you want to set the fontsize alongside other font properties, use the `*size*` parameter in `*title_fontproperties*`.

`alignment` : {'center', 'left', 'right'}, default: 'center'

The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

`borderpad` : float, default: `:rc:`legend.borderpad``

The fractional whitespace inside the legend border, in font-size units.

`labelspacing` : float, default: `:rc:`legend.labelspacing``

The vertical space between the legend entries, in font-size units.

`handlelength` : float, default: `:rc:`legend.handlelength``
The length of the legend handles, in font-size units.

`handleheight` : float, default: `:rc:`legend.handleheight``
The height of the legend handles, in font-size units.

`handletextpad` : float, default: `:rc:`legend.handletextpad``
The pad between the legend handle and text, in font-size units.

`borderaxespad` : float, default: `:rc:`legend.borderaxespad``
The pad between the Axes and legend border, in font-size units.

`columnspacing` : float, default: `:rc:`legend.columnspacing``
The spacing between columns, in font-size units.

`handler_map` : dict or None
The custom dictionary mapping instances or types to a legend handler. This `*handler_map*` updates the default handler map found at ``matplotlib.legend.Legend.get_legend_handler_map``.

`draggable` : bool, default: False
Whether the legend can be dragged with the mouse.

See Also

`.Figure.legend`

Notes

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``axes.Axes.legend``.

Some artists are not supported by this function. See `:ref:`legend_guide`` for details.

Examples

.. plot:: gallery/text_labels_and_annotations/legend.py

matplotlib.pyplot.locator_params

```
locator_params(axis: "Literal['both', 'x', 'y']" = 'both', tight: 'bool | None' =
None, **kwargs) -> 'None'
```

Control behavior of major tick locators.

Because the locator is involved in autoscaling, ``~.Axes.autoscale_view`` is called automatically after the parameters are changed.

Parameters

axis : {'both', 'x', 'y'}, default: 'both'
The axis on which to operate. (For 3D Axes, *axis* can also be set to 'z', and 'both' refers to all three axes.)
tight : bool or None, optional
Parameter passed to `~.Axes.autoscale_view`.
Default is None, for no change.

Other Parameters

****kwargs**

Remaining keyword arguments are passed directly to the `set_params()` method of the locator. Supported keywords depend on the type of the locator. See for example `~.ticker.MaxNLocator.set_params` for the `~.ticker.MaxNLocator` used by default for linear.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.locator_params`.

Examples

When plotting small subplots, one might want to reduce the maximum number of ticks and use tight bounds, for example::

```
ax.locator_params(tight=True, nbins=4)
```

matplotlib.pyplot.loglog

```
loglog(*args, **kwargs) -> 'list[Line2D]'
```

Make a plot with log scaling on both the x- and y-axis.

Call signatures::

```
loglog([x], y, [fmt], data=None, **kwargs)  
loglog([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around `~.plot` which additionally changes both the x-axis and the y-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters `*base*`, `*subs*` and `*nonpositive*` control the x/y-axis properties. They are just forwarded to `~.Axes.set_xscale` and `~.Axes.set_yscale`. To use different properties on the x-axis and the y-axis, use e.g.
`ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`

Parameters

base : float, default: 10

Base of the logarithm.

subs : sequence, optional

The location of the minor ticks. If `*None*`, reasonable locations are automatically chosen depending on the number of decades in the plot. See ``Axes.set_xscale`/`.Axes.set_yscale`` for details.

nonpositive : {'mask', 'clip'}, default: 'clip'

Non-positive values can be masked as invalid, or clipped to a very small positive number.

****kwargs**

All parameters supported by ``plot``.

Returns

list of ``Line2D``

Objects representing the plotted data.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.loglog``.

matplotlib.pyplot.magma

```
magma() -> 'None'
```

Set the colormap to 'magma'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

matplotlib.pyplot.magnitude_spectrum

```
magnitude_spectrum(x: 'ArrayLike', *, Fs: 'float | None' = None, Fc: 'int | None' = None, window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None, scale: "Literal['default', 'linear', 'dB'] | None" = None, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the magnitude spectrum.

Compute the magnitude spectrum of `*x*`. Data is padded to a length of `*pad_to*` and the windowing function `*window*` is applied to the signal.

Parameters

x : 1-D array or sequence

Array or sequence containing the data.

Fs : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

`window` : callable or ndarray, default: ``window_hanning``

A function or a vector of length `*NFFT*`. To create window vectors see ``window_hanning``, ``window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to ``~numpy.fft.fft``. The default is None, which sets `*pad_to*` equal to the length of the input signal (i.e. no padding).

`scale` : {'default', 'linear', 'dB'}

The scaling of the values in the `*spec*`. 'linear' is no scaling. 'dB' returns the values in dB scale, i.e., the dB amplitude ($20 * \log_{10}$). 'default' is 'linear'.

`Fc` : int, default: 0

The center frequency of `*x*`, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

`spectrum` : 1-D array

The values for the magnitude spectrum before scaling (real valued).

`freqs` : 1-D array

The frequencies corresponding to the elements in `*spectrum*`.

`line` : ``~matplotlib.lines.Line2D``

The line created by this function.

Other Parameters

`data` : indexable object, optional

If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

`*x*`

`**kwargs`

Keyword arguments control the ``Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: ``.CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``.JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

`markersize` or `ms`: float

`markevery`: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

`mouseover`: bool

`path_effects`: list of ``.AbstractPathEffect``

`picker`: float or callable[[Artist, Event], tuple[bool, dict]]

`pickradius`: float

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`solid_capstyle`: ``.CapStyle`` or {'butt', 'projecting', 'round'}

`solid_joinstyle`: ``.JoinStyle`` or {'miter', 'round', 'bevel'}

`transform`: unknown

`url`: str

`visible`: bool

`xdata`: 1D array

`ydata`: 1D array

`zorder`: float

See Also

`psd`

Plots the power spectral density.

`angle_spectrum`

Plots the angles of the corresponding frequencies.

`phase_spectrum`

Plots the phase (unwrapped angle) of the corresponding frequencies.

`spectrogram`

Can plot the magnitude spectrum of segments within the signal in a colormap.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.magnitude_spectrum``.

matplotlib.pyplot.margins

```
margins(*margins: 'float', x: 'float | None' = None, y: 'float | None' = None, tight: 'bool | None' = True) -> 'tuple[float, float] | None'
```

Set or retrieve margins around the data for autoscaling axis limits.

This allows to configure the padding around the data without having to set explicit limits using `~.Axes.set_xlim` / `~.Axes.set_ylim``.

Autoscaling determines the axis limits by adding `*margin*` times the data interval as padding around the data. See the following illustration:

.. plot:: _embedded_plots/axes_margins.py

All input parameters must be floats greater than -0.5. Passing both positional and keyword arguments is invalid and will raise a `TypeError`. If no arguments (positional or otherwise) are provided, the current margins will remain unchanged and simply be returned.

The default margins are :rc:`axes.xmargin` and :rc:`axes.ymargin`.

Parameters

`*margins` : float, optional

If a single positional argument is provided, it specifies both margins of the x-axis and y-axis limits. If two positional arguments are provided, they will be interpreted as `*xmargin*`, `*ymargin*`. If setting the margin on a single axis is desired, use the keyword arguments described below.

`x, y` : float, optional

Specific margin values for the x-axis and y-axis, respectively. These cannot be used with positional arguments, but can be used individually to alter on e.g., only the y-axis.

`tight` : bool or None, default: True

The `*tight*` parameter is passed to `~.axes.Axes.autoscale_view``, which is executed after a margin is changed; the default here is `*True*`, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting `*tight*` to `*None*` preserves the previous setting.

Returns

xmargin, ymargin : float

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.margins`.

If a previously used Axes method such as :meth:`pcolor` has set `~.Axes.use_sticky_edges` to `True`, only the limits not set by the "sticky artists" will be modified. To force all margins to be set, set `~.Axes.use_sticky_edges` to `False` before calling :meth:`margins`.

See Also

`.Axes.set_xmargin`, `.Axes.set_ymargin`

matplotlib.pyplot.matshow

```
matshow(A: 'ArrayLike', fignum: 'None | int' = None, **kwargs) -> 'AxesImage'
```

Display a 2D array as a matrix in a new figure window.

The origin is set at the upper left hand corner.

The indexing is `((row, column))` so that the first index runs vertically and the second index runs horizontally in the figure:

.. code-block:: none

`A[0, 0]` ■ `A[0, M-1]`

■ ■

`A[N-1, 0]` ■ `A[N-1, M-1]`

The aspect ratio of the figure window is that of the array, unless this would make an excessively short or narrow figure.

Tick labels for the xaxis are placed on top.

Parameters

`A` : 2D array-like

The matrix to be displayed.

`fignum` : None or int

If `*None`, create a new, appropriately sized figure window.

If 0, use the current Axes (creating one if there is none, without ever adjusting the figure size).

Otherwise, create a new Axes on the figure with the given number

(creating it at the appropriate size if it does not exist, but not adjusting the figure size otherwise). Note that this will be drawn on top of any preexisting Axes on the figure.

Returns

`~matplotlib.image.AxesImage`

Other Parameters

**kwargs : `~matplotlib.axes.Axes.imshow` arguments

matplotlib.pyplot.minorticks_off

```
minorticks_off() -> 'None'
```

Remove minor ticks from the Axes.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.minorticks_off``.

matplotlib.pyplot.minorticks_on

```
minorticks_on() -> 'None'
```

Display minor ticks on the Axes.

Displaying minor ticks may reduce performance; you may turn them off using ``minorticks_off()`` if drawing speed is a problem.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.minorticks_on``.

matplotlib.pyplot.new_figure_manager

```
new_figure_manager(num, *args, FigureClass=, **kwargs)
```

Create a new figure manager instance.

matplotlib.pyplot.nipy_spectral

```
nipy_spectral() -> 'None'
```

Set the colormap to 'nipy_spectral'.

This changes the default colormap as well as the colormap of the current

image if there is one. See ``help(colormaps)`` for more information.

matplotlib.pyplot.num2date

```
num2date(x, tz=None)
```

Convert Matplotlib dates to ``~datetime.datetime`` objects.

Parameters

x : float or sequence of floats
Number of days (fraction part represents hours, minutes, seconds)
since the epoch. See ``.get_epoch`` for the
epoch, which can be changed by ``:rc:`date.epoch` or ``.set_epoch``.
tz : str or ``~datetime.tzinfo``, default: ``:rc:`timezone`
Timezone of *x*. If a string, *tz* is passed to ``dateutil.tz``.

Returns

``~datetime.datetime`` or sequence of ``~datetime.datetime``
Dates are returned in timezone *tz*.

If *x* is a sequence, a sequence of ``~datetime.datetime`` objects will
be returned.

Notes

The Gregorian calendar is assumed; this is not universal practice.
For details, see the module docstring.

matplotlib.pyplot.pause

```
pause(interval: 'float') -> 'None'
```

Run the GUI event loop for *interval* seconds.

If there is an active figure, it will be updated and displayed before the
pause, and the GUI event loop (if any) will run during the pause.

This can be used for crude animation. For more complex animation use
:mod:`matplotlib.animation`.

If there is no active figure, sleep for *interval* seconds instead.

See Also

matplotlib.animation : Proper animations
show : Show all figures and optional block until all figures are closed.

matplotlib.pyplot.pcolor

```
pcolor(*args: 'ArrayLike', shading: "Literal['flat', 'nearest', 'auto'] | None" =
None, alpha: 'float | None' = None, norm: 'str | Normalize | None' = None, cmap: 'str
| Colormap | None' = None, vmin: 'float | None' = None, vmax: 'float | None' = None,
colorizer: 'Colorizer | None' = None, data=None, **kwargs) -> 'Collection'
```

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature::

```
pcolor([X, Y,] C, /, **kwargs)
```

X and **Y** can be used to specify the corners of the quadrilaterals.

The arguments **X**, **Y**, **C** are positional-only.

.. hint::

`pcolor()` can be very slow for large arrays. In most cases you should use the similar but much faster `~.Axes.pcolormesh` instead. See :ref:`Differences between pcolor() and pcolormesh() <differences-pcolor-pcolormesh>` for a discussion of the differences.

Parameters

C : 2D array-like

The color-mapped values. Color-mapping is controlled by **cmap**, **norm**, **vmin**, and **vmax**.

X, *Y* : array-like, optional

The coordinates of the corners of quadrilaterals of a `pcolormesh`::

```
(X[i+1, j], Y[i+1, j]) (X[i+1, j+1], Y[i+1, j+1])
```

```
●●●●●●●●
```

```
■ ■
```

```
●●●●●●●●
```

```
(X[i, j], Y[i, j]) (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the :ref:`Notes <axes-pcolormesh-grid-orientation>` section below.

If `shading='flat'` the dimensions of **X** and **Y** should be one greater than those of **C**, and the quadrilateral is colored due to the value at `C[i, j]`. If **X**, **Y** and **C** have equal dimensions, a warning will be raised and the last row and column of **C** will be ignored.

If `shading='nearest'`, the dimensions of **X** and **Y** should be the same as those of **C** (if not, a `ValueError` will be raised). The color `C[i, j]` will be centered on `(X[i, j], Y[i, j])`.

If **X** and/or **Y** are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2D arrays, making a rectangular grid.

shading : {'flat', 'nearest', 'auto'}, default: :rc:`pcolor.shading`
The fill style for the quadrilateral. Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of `X` and `Y` should be one greater than those of `C`; if they are the same as `C`, then a deprecation warning is raised, and the last row and column of `C` are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of `X` and `Y` must be the same as `C`.
- 'auto': Choose 'flat' if dimensions of `X` and `Y` are one larger than `C`. Choose 'nearest' if dimensions are the same.

See :doc:`gallery/images_contours_and_fields/pcolormesh_grids` for more description.

cmap : str or `~matplotlib.colors.Colormap`, default: :rc:`image.cmap`
The Colormap instance or registered colormap name used to map scalar data to colors.

norm : str or `~matplotlib.colors.Normalize`, optional
The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `cmap`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize` or one of its subclasses (see :ref:`colormapnorms`).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()`. In that case, a suitable `.Normalize` subclass is dynamically generated and instantiated.

vmin, vmax : float, optional
When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a `str *norm*` name together with `*vmin*/vmax*` is acceptable).

colorizer : `~matplotlib.colorbar.Colorizer` or None, default: None
The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

edgecolors : {'none', None, 'face', color, color sequence}, optional
The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.
- `*None*`: :rc:`patch.edgecolor` will be used. Note that currently :rc:`patch.force_edgecolor` has to be True for this to work.
- 'face': Use the adjacent face color.

- A color or sequence of colors will set the edge color.

The singular form `*edgecolor*` works as an alias.

`alpha` : float, default: None

The alpha blending value of the face color, between 0 (transparent) and 1 (opaque). Note: The `edgecolor` is currently not affected by this.

`snap` : bool, default: False

Whether to snap the mesh to pixel boundaries.

Returns

``matplotlib.collections.PolyQuadMesh``

Other Parameters

`antialiaseds` : bool, default: False

The default `*antialiaseds*` is False if the default `*edgecolors*` `\="none"` is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If `*edgecolors*` is not "none", then the default `*antialiaseds*` is taken from `:rc:`patch.antialiased``. Stroking the edges may be preferred if `*alpha*` is 1, but will cause artifacts otherwise.

`data` : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

****kwargs**

Additionally, the following arguments are allowed. They are passed along to the `~matplotlib.collections.PolyQuadMesh`` constructor:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: array-like or float or None

`animated`: bool

`antialiased` or `aa` or `antialiaseds`: bool or list of bools

`array`: array-like or None

`capstyle`: ``.CapStyle`` or `{'butt', 'projecting', 'round'}`

`clim`: (vmin: float, vmax: float)

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`cmap`: ``.Colormap`` or str or None

`color`: `:mpltype:`color`` or list of RGBA tuples

`edgecolor` or `ec` or `edgecolors`: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'

`facecolor` or `facecolors` or `fc`: `:mpltype:`color`` or list of `:mpltype:`color``

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`gid`: str

`hatch`: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`

`hatch_linewidth`: unknown

in_layout: bool
 joinstyle: ``JoinStyle`` or `{'miter', 'round', 'bevel'}`
 label: object
 linestyle or dashes or linestyles or ls: str or tuple or list thereof
 linewidth or linewidths or lw: float or list of floats
 mouseover: bool
 norm: ``Normalize`` or str or None
 offset_transform or transOffset: ``Transform``
 offsets: (N, 2) or (2,) array-like
 path_effects: list of ``AbstractPathEffect``
 paths: list of array-like
 picker: None or bool or float or callable
 pickradius: float
 rasterized: bool
 sizes: ``numpy.ndarray`` or None
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 transform: ``~matplotlib.transforms.Transform``
 url: str
 urls: list of str or None
 verts: list of array-like
 verts_and_codes: unknown
 visible: bool
 zorder: float

See Also

 pcolormesh : for an explanation of the differences between
 pcolor and pcolormesh.
 imshow : If `*X*` and `*Y*` are each equidistant, ``~.Axes.imshow`` can be a
 faster alternative.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``~.axes.Axes.pcolor``.

****Masked arrays****

`*X*`, `*Y*` and `*C*` may be masked arrays. If either ``C[i, j]``, or one
 of the vertices surrounding ``C[i, j]`` (`*X*` or `*Y*` at
``[i, j], [i+1, j], [i, j+1], [i+1, j+1]``) is masked, nothing is
 plotted.

.. _axes-pcolor-grid-orientation:

****Grid orientation****

The grid orientation follows the standard matrix convention: An array
`*C*` with shape (nrows, ncolumns) is plotted with the column number as
`*X*` and the row number as `*Y*`.

```
pcolormesh(*args: 'ArrayLike', alpha: 'float | None' = None, norm: 'str | Normalize | None' = None, cmap: 'str | Colormap | None' = None, vmin: 'float | None' = None, vmax: 'float | None' = None, colorizer: 'Colorizer | None' = None, shading: "Literal['flat', 'nearest', 'gouraud', 'auto'] | None" = None, antialiased: 'bool' = False, data=None, **kwargs) -> 'QuadMesh'
```

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature::

`pcolormesh([X, Y,] C, /, **kwargs)`

`*X*` and `*Y*` can be used to specify the corners of the quadrilaterals.

The arguments `*X*`, `*Y*`, `*C*` are positional-only.

.. hint::

`~.Axes.pcolormesh` is similar to ~.Axes.pcolor`. It is much faster and preferred in most cases. For a detailed discussion on the differences see :ref:`Differences between pcolor() and pcolormesh() <differences-pcolor-pcolormesh>`.`

Parameters

`C` : array-like

The mesh data. Supported array shapes are:

- (M, N) or M*N: a mesh with scalar data. The values are mapped to colors using normalization and a colormap. See parameters `*norm*`, `*cmap*`, `*vmin*`, `*vmax*`.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the mesh data.

`X`, `Y` : array-like, optional

The coordinates of the corners of quadrilaterals of a `pcolormesh`::

```
(X[i+1, j], Y[i+1, j]) (X[i+1, j+1], Y[i+1, j+1])
●■■■■■●
■ ■
●■■■■■●
(X[i, j], Y[i, j]) (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the :ref:`Notes <axes-pcolormesh-grid-orientation>` section below.

If ``shading='flat'`` the dimensions of `*X*` and `*Y*` should be one greater than those of `*C*`, and the quadrilateral is colored due to the value at ``C[i, j]``. If `*X*`, `*Y*` and `*C*` have equal dimensions, a warning will be raised and the last row and column

of **C** will be ignored.

If `shading='nearest'` or `'gouraud'`, the dimensions of **X** and **Y** should be the same as those of **C** (if not, a `ValueError` will be raised). For `'nearest'` the color `C[i, j]` is centered on `(X[i, j], Y[i, j])`. For `'gouraud'`, a smooth interpolation is carried out between the quadrilateral corners.

If **X** and/or **Y** are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2D arrays, making a rectangular grid.

`cmap` : str or `~matplotlib.colors.Colormap`, default: `:rc:image.cmap`
The `Colormap` instance or registered colormap name used to map scalar data to colors.

`norm` : str or `~matplotlib.colors.Normalize`, optional
The normalization method used to scale scalar data to the `[0, 1]` range before mapping to colors using **cmap**. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize` or one of its subclasses (see `:ref:colormapnorms`).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()`. In that case, a suitable `.Normalize` subclass is dynamically generated and instantiated.

`vmin, vmax` : float, optional
When using scalar data and no explicit **norm**, **vmin** and **vmax** define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use **vmin*/vmax** when a **norm** instance is given (but using a `str` **norm** name together with **vmin*/vmax** is acceptable).

`colorizer` : `~matplotlib.colorbar.Colorizer` or `None`, default: `None`
The `Colorizer` object used to map color to data. If `None`, a `Colorizer` object is created from a **norm** and **cmap**.

`edgecolors` : {'none', `None`, 'face', color, color sequence}, optional
The color of the edges. Defaults to 'none'. Possible values:

- 'none' or `''`: No edge.
- `*None*`: `:rc:patch.edgecolor` will be used. Note that currently `:rc:patch.force_edgecolor` has to be `True` for this to work.
- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form **edgecolor** works as an alias.

`alpha` : float, default: `None`
The alpha blending value, between 0 (transparent) and 1 (opaque).

shading : {'flat', 'nearest', 'gouraud', 'auto'}, optional
The fill style for the quadrilateral; defaults to
:rc:`pcolor.shading`. Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of `X` and `Y` should be one greater than those of `C`; if they are the same as `C`, then a deprecation warning is raised, and the last row and column of `C` are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of `X` and `Y` must be the same as `C`.
- 'gouraud': Each quad will be Gouraud shaded: The color of the corners (i, j) are given by `C[i, j]`. The color values of the area in between is interpolated from the corner values. The dimensions of `X` and `Y` must be the same as `C`. When Gouraud shading is used, `edgecolors` is ignored.
- 'auto': Choose 'flat' if dimensions of `X` and `Y` are one larger than `C`. Choose 'nearest' if dimensions are the same.

See :doc:`/gallery/images_contours_and_fields/pcolormesh_grids` for more description.

snap : bool, default: False
Whether to snap the mesh to pixel boundaries.

rasterized : bool, optional
Rasterize the pcolormesh when drawing vector graphics. This can speed up rendering and produce smaller files for large data sets. See also :doc:`/gallery/misc/rasterization_demo`.

Returns

`matplotlib.collections.QuadMesh`

Other Parameters

data : indexable object, optional
If given, all parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`.

**kwargs

Additionally, the following arguments are allowed. They are passed along to the `~matplotlib.collections.QuadMesh` constructor:

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
alpha: array-like or float or None
animated: bool
antialiased or aa or antialiaseds: bool or list of bools
array: array-like
capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
clim: (vmin: float, vmax: float)

`clip_box`: `~matplotlib.transforms.BboxBase` or `None`
`clip_on`: `bool`
`clip_path`: `Patch` or `(Path, Transform)` or `None`
`cmap`: `~.Colormap` or `str` or `None`
`color`: `:mpltype:color` or list of `RGBA` tuples
`edgecolor` or `ec` or `edgecolors`: `:mpltype:color` or list of `:mpltype:color` or `'face'`
`facecolor` or `facecolors` or `fc`: `:mpltype:color` or list of `:mpltype:color`
`figure`: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`
`gid`: `str`
`hatch`: `{ '/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*' }`
`hatch_linewidth`: `unknown`
`in_layout`: `bool`
`joinstyle`: `~.JoinStyle` or `{ 'miter', 'round', 'bevel' }`
`label`: `object`
`linestyle` or `dashes` or `linestyles` or `ls`: `str` or `tuple` or list thereof
`linewidth` or `linewidths` or `lw`: `float` or list of floats
`mouseover`: `bool`
`norm`: `~.Normalize` or `str` or `None`
`offset_transform` or `transOffset`: `~.Transform`
`offsets`: `(N, 2)` or `(2,)` array-like
`path_effects`: list of `~.AbstractPathEffect`
`picker`: `None` or `bool` or `float` or callable
`pickradius`: `float`
`rasterized`: `bool`
`sketch_params`: (`scale`: `float`, `length`: `float`, `randomness`: `float`)
`snap`: `bool` or `None`
`transform`: `~matplotlib.transforms.Transform`
`url`: `str`
`urls`: list of `str` or `None`
`visible`: `bool`
`zorder`: `float`

See Also

`pcolor` : An alternative implementation with slightly different features. For a detailed discussion on the differences see [:ref: Differences between pcolor\(\) and pcolormesh\(\)](#)
[<ifferences-pcolor-pcolormesh>](#)
`imshow` : If `*X*` and `*Y*` are each equidistant, `~.Axes.imshow` can be a faster alternative.

Notes

.. note::

This is the [:ref: pyplot wrapper <pyplot_interface>](#) for `~.axes.Axes.pcolormesh`.

****Masked arrays****

`*C*` may be a masked array. If ```C[i, j]``` is masked, the corresponding quadrilateral will be transparent. Masking of `*X*` and `*Y*` is not supported. Use `~.Axes.pcolor` if you need this functionality.

.. `_axes-pcolormesh-grid-orientation`:

****Grid orientation****

The grid orientation follows the standard matrix convention: An array **C** with shape (nrows, ncolumns) is plotted with the column number as **X** and the row number as **Y**.

.. _differences-pcolor-pcolormesh:

****Differences between pcolor() and pcolormesh()****

Both methods are used to create a pseudocolor plot of a 2D array using quadrilaterals.

The main difference lies in the created object and internal data handling:

While `~.Axes.pcolor`` returns a `~.PolyQuadMesh``, `~.Axes.pcolormesh`` returns a `~.QuadMesh``. The latter is more specialized for the given purpose and thus is faster. It should almost always be preferred.

There is also a slight difference in the handling of masked arrays. Both `~.Axes.pcolor`` and `~.Axes.pcolormesh`` support masked arrays for **C**. However, only `~.Axes.pcolor`` supports masked arrays for **X** and **Y**. The reason lies in the internal handling of the masked values. `~.Axes.pcolor`` leaves out the respective polygons from the `PolyQuadMesh`. `~.Axes.pcolormesh`` sets the facecolor of the masked elements to transparent. You can see the difference when using edgecolors. While all edges are drawn irrespective of masking in a `QuadMesh`, the edge between two adjacent masked quadrilaterals in `~.Axes.pcolor`` is not drawn as the corresponding polygons do not exist in the `PolyQuadMesh`. Because `PolyQuadMesh` draws each individual polygon, it also supports applying hatches and linestyle to the collection.

Another difference is the support of Gouraud shading in `~.Axes.pcolormesh``, which is not available with `~.Axes.pcolor``.

matplotlib.pyplot.phase_spectrum

```
phase_spectrum(x: 'ArrayLike', *, Fs: 'float | None' = None, Fc: 'int | None' = None,
window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, pad_to: 'int |
None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None,
data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the phase spectrum.

Compute the phase spectrum (unwrapped angle spectrum) of **x**. Data is padded to a length of **pad_to** and the windowing function **window** is applied to the signal.

Parameters

x : 1-D array or sequence

Array or sequence containing the data

Fs : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

`window` : callable or ndarray, default: ``window_hanning``

A function or a vector of length `*NFFT*`. To create window vectors see ``window_hanning``, ``window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to ``~numpy.fft.fft``. The default is None, which sets `*pad_to*` equal to the length of the input signal (i.e. no padding).

`Fc` : int, default: 0

The center frequency of `*x*`, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

`spectrum` : 1-D array

The values for the phase spectrum in radians (real valued).

`freqs` : 1-D array

The frequencies corresponding to the elements in `*spectrum*`.

`line` : ``~matplotlib.lines.Line2D``

The line created by this function.

Other Parameters

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`

`**kwargs`

Keyword arguments control the ``Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

antialiased or aa: bool
 clip_box: `~matplotlib.transforms.BboxBase` or None
 clip_on: bool
 clip_path: Patch or (Path, Transform) or None
 color or c: `:mpltype:color`
 dash_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}
 dash_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}
 dashes: sequence of floats (on/off ink in points) or (None, None)
 data: (2, N) array or two 1D arrays
 drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
 figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`
 fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
 gapcolor: `:mpltype:color` or None
 gid: str
 in_layout: bool
 label: object
 linestyle or ls: {'-', '--', '-.', ':', ''}, (offset, on-off-seq), ...}
 linewidth or lw: float
 marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
 markeredgewidth or mec: `:mpltype:color`
 markeredgewidth or mew: float
 markerfacecolor or mfc: `:mpltype:color`
 markerfacecoloralt or mfcalt: `:mpltype:color`
 markersize or ms: float
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]
 mouseover: bool
 path_effects: list of `~.AbstractPathEffect`
 picker: float or callable[[Artist, Event], tuple[bool, dict]]
 pickradius: float
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 solid_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}
 solid_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}
 transform: unknown
 url: str
 visible: bool
 xdata: 1D array
 ydata: 1D array
 zorder: float

See Also

`magnitude_spectrum`

Plots the magnitudes of the corresponding frequencies.

`angle_spectrum`

Plots the wrapped version of this function.

`specgram`

Can plot the phase spectrum of segments within the signal in a colormap.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.phase_spectrum``.

matplotlib.pyplot.pie

```
pie(x: 'ArrayLike', *, explode: 'ArrayLike | None' = None, labels: 'Sequence[str] | None' = None, colors: 'ColorType | Sequence[ColorType] | None' = None, autopct: 'str | Callable[[float], str] | None' = None, pctdistance: 'float' = 0.6, shadow: 'bool' = False, labeldistance: 'float | None' = 1.1, startangle: 'float' = 0, radius: 'float' = 1, counterclock: 'bool' = True, wedgeprops: 'dict[str, Any] | None' = None, textprops: 'dict[str, Any] | None' = None, center: 'tuple[float, float]' = (0, 0), frame: 'bool' = False, rotatelabels: 'bool' = False, normalize: 'bool' = True, hatch: 'str | Sequence[str] | None' = None, data=None) -> 'tuple[list[Wedge], list[Text]] | tuple[list[Wedge], list[Text], list[Text]]'
```

Plot a pie chart.

Make a pie chart of array `*x*`. The fractional area of each wedge is given by ```x/sum(x)```.

The wedges are plotted counterclockwise, by default starting from the x-axis.

Parameters

`x` : 1D array-like

The wedge sizes.

`explode` : array-like, default: None

If not `*None*`, is a ```len(x)``` array which specifies the fraction of the radius with which to offset each wedge.

`labels` : list, default: None

A sequence of strings providing the labels for each wedge

`colors` : :mpltype:`color` or list of :mpltype:`color`, default: None

A sequence of colors through which the pie chart will cycle. If `*None*`, will use the colors in the currently active cycle.

`hatch` : str or list, default: None

Hatching pattern applied to all pie wedges or sequence of patterns through which the chart will cycle. For a list of valid patterns, see :doc:`/gallery/shapes_and_collections/hatch_style_reference`.

.. versionadded:: 3.7

`autopct` : None or str or callable, default: None

If not `*None*`, `*autopct*` is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If `*autopct*` is a format string, the label will be ```fmt % pct```. If `*autopct*` is a function, then it will be called.

`pctdistance` : float, default: 0.6

The relative distance along the radius at which the text generated by `*autopct*` is drawn. To draw the text outside the pie, set `*pctdistance* > 1`. This parameter is ignored if `*autopct*` is

```None```.

`labeldistance` : float or None, default: 1.1

The relative distance along the radius at which the labels are drawn. To draw the labels inside the pie, set `*labeldistance* < 1`. If set to ```None```, labels are not drawn but are still stored for use in ``.legend``.

`shadow` : bool or dict, default: False

If bool, whether to draw a shadow beneath the pie. If dict, draw a shadow passing the properties in the dict to ``.Shadow``.

.. versionadded:: 3.8

`*shadow*` can be a dict.

`startangle` : float, default: 0 degrees

The angle by which the start of the pie is rotated, counterclockwise from the x-axis.

`radius` : float, default: 1

The radius of the pie.

`counterclock` : bool, default: True

Specify fractions direction, clockwise or counterclockwise.

`wedgeprops` : dict, default: None

Dict of arguments passed to each ``.patches.Wedge`` of the pie. For example, ```wedgeprops = {'linewidth': 3}``` sets the width of the wedge border lines equal to 3. By default, ```clip_on=False```. When there is a conflict between these properties and other keywords, properties passed to `*wedgeprops*` take precedence.

`textprops` : dict, default: None

Dict of arguments to pass to the text objects.

`center` : (float, float), default: (0, 0)

The coordinates of the center of the chart.

`frame` : bool, default: False

Plot Axes frame with the chart if true.

`rotatelabels` : bool, default: False

Rotate each label to the angle of the corresponding slice if true.

`normalize` : bool, default: True

When `*True*`, always make a full pie by normalizing `x` so that ```sum(x) == 1```. `*False*` makes a partial pie if ```sum(x) <= 1``` and raises a ``ValueError`` for ```sum(x) > 1```.

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*explode*`, `*labels*`, `*colors*`

## Returns

-----

patches : list

A sequence of ``matplotlib.patches.Wedge`` instances

texts : list

A list of the label ``Text`` instances.

autotexts : list

A list of ``Text`` instances for the numeric labels. This will only be returned if the parameter `*autopct*` is not `*None*`.

## Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``axes.Axes.pie``.

The pie chart will probably look best if the figure and Axes are square, or the Axes aspect is equal.

This method sets the aspect ratio of the axis to "equal".

The Axes aspect ratio can be controlled with ``Axes.set_aspect``.

## matplotlib.pyplot.pink

```
pink() -> 'None'
```

Set the colormap to 'pink'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.plasma

```
plasma() -> 'None'
```

Set the colormap to 'plasma'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.plot

```
plot(*args: 'float | ArrayLike | str', scalex: 'bool' = True, scaley: 'bool' = True, data=None, **kwargs) -> 'list[Line2D]'
```

Plot y versus x as lines and/or markers.

Call signatures::

```
plot([x], y, [fmt], *, data=None, **kwargs)
```

```
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by `*x*`, `*y*`.

The optional parameter `*fmt*` is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the `*Notes*` section below.

```
>>> plot(x, y) # plot x and y using default line style and color
>>> plot(x, y, 'bo') # plot x and y using blue circle markers
>>> plot(y) # plot y using x as index array 0..N-1
>>> plot(y, 'r+') # ditto, but with red plusses
```

You can use `.Line2D`` properties as keyword arguments for more control on the appearance. Line properties and `*fmt*` can be mixed. The following two calls yield identical results:

```
>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
... linewidth=2, markersize=12)
```

When conflicting with `*fmt*`, keyword arguments take precedence.

#### **\*\*Plotting labelled data\*\***

There's a convenient way for plotting objects with labelled data (i.e. data that can be accessed by index `obj['y']`). Instead of giving the data in `*x*` and `*y*`, you can provide the object in the `*data*` parameter and just give the labels for `*x*` and `*y*`:

```
>>> plot('xlabel', 'ylabel', data=obj)
```

All indexable objects are supported. This could e.g. be a ``dict``, a ``pandas.DataFrame`` or a structured numpy array.

#### **\*\*Plotting multiple sets of data\*\***

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call ``plot`` multiple times.  
Example:

```
>>> plot(x1, y1, 'bo')
>>> plot(x2, y2, 'go')
```

- If `*x*` and/or `*y*` are 2D arrays, a separate data set will be drawn for every column. If both `*x*` and `*y*` are 2D, they must have the same shape. If only one of them is 2D with shape `(N, m)` the other must have length `N` and will be used for every data set `m`.

Example:

```
>>> x = [1, 2, 3]
>>> y = np.array([[1, 2], [3, 4], [5, 6]])
>>> plot(x, y)
```

is equivalent to:

```
>>> for col in range(y.shape[1]):
... plot(x, y[:, col])
```

- The third way is to specify multiple sets of `*[x]*`, `*y*`, `*[fmt]*` groups::

```
>>> plot(x1, y1, 'g^', x2, y2, 'g-')
```

In this case, any additional keyword argument applies to all datasets. Also, this syntax cannot be combined with the `*data*` parameter.

By default, each line is assigned a different style specified by a 'style cycle'. The `*fmt*` and line property parameters are only necessary if you want explicit deviations from these defaults. Alternatively, you can also change the style cycle using `:rc:`axes.prop_cycle``.

#### Parameters

-----  
`x, y` : array-like or float

The horizontal / vertical coordinates of the data points.

`*x*` values are optional and default to ```range(len(y))```.

Commonly, these parameters are 1D arrays.

They can also be scalars, or two-dimensional (in that case, the columns represent separate data sets).

These arguments cannot be passed as keywords.

`fmt` : str, optional

A format string, e.g. 'ro' for red circles. See the `*Notes*` section for a full description of the format strings.

Format strings are just an abbreviation for quickly setting basic line properties. All of these and more can also be controlled by keyword arguments.

This argument cannot be passed as keyword.

`data` : indexable object, optional

An object with labelled data. If given, provide the label names to plot in `*x*` and `*y*`.

.. note::

Technically there's a slight ambiguity in calls where the second label is a valid `*fmt*`. ```plot('n', 'o', data=obj)``` could be ```plt(x, y)``` or ```plt(y, fmt)```. In such cases, the former interpretation is chosen, but a warning is issued. You may suppress the warning by adding an empty format string ```plot('n', 'o', "", data=obj)```.

## Returns

-----

list of ``Line2D``

A list of lines representing the plotted data.

## Other Parameters

-----

`scalex, scaley` : bool, default: True

These parameters determine if the view limits are adapted to the data limits. The values are passed on to

``~.axes.Axes.autoscale_view``.

**\*\*kwargs** : ``~matplotlib.lines.Line2D`` properties, optional

**\*kwargs\*** are used to specify properties like a line label (for auto legends), linewidth, antialiasing, marker face color.

Example::

```
>>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
```

```
>>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')
```

If you specify multiple lines with one plot call, the kwargs apply to all those lines. In case the label object is iterable, each element is used as labels for each set of data.

Here is a list of available ``Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: ``~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: ``~.CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``~.JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, ``~.path.Path`` or ``~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

markersize or ms: float  
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]  
 mouseover: bool  
 path\_effects: list of ``AbstractPathEffect``  
 picker: float or callable[[Artist, Event], tuple[bool, dict]]  
 pickradius: float  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 solid\_capstyle: ``CapStyle`` or {'butt', 'projecting', 'round'}  
 solid\_joinstyle: ``JoinStyle`` or {'miter', 'round', 'bevel'}  
 transform: unknown  
 url: str  
 visible: bool  
 xdata: 1D array  
 ydata: 1D array  
 zorder: float

#### See Also

-----

scatter : XY scatter plot with markers of varying size and/or color (sometimes also called bubble chart).

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``axes.Axes.plot``.

#### **\*\*Format Strings\*\***

A format string consists of a part for color, marker and line::

```
fmt = '[marker][line][color]'
```

Each of them is optional. If not provided, the value from the style cycle is used. Exception: If ```line``` is given, but no ```marker```, the data will be a line without markers.

Other combinations such as ```[color][marker][line]``` are also supported, but note that their parsing may be ambiguous.

#### **\*\*Markers\*\***

```
=====
character description
=====
```

```
``.``` point marker
``,`` pixel marker
``o`` circle marker
``v`` triangle_down marker
``^`` triangle_up marker
``<`` triangle_left marker
``>`` triangle_right marker
```

```

'''1''' tri_down marker
'''2''' tri_up marker
'''3''' tri_left marker
'''4''' tri_right marker
'''8''' octagon marker
'''s''' square marker
'''p''' pentagon marker
'''P''' plus (filled) marker
'''*''' star marker
'''h''' hexagon1 marker
'''H''' hexagon2 marker
'''+'''' plus marker
'''x''' x marker
'''X''' x (filled) marker
'''D''' diamond marker
'''d''' thin_diamond marker
'''|''' vline marker
'''_''' hline marker
=====

```

## **\*\*Line Styles\*\***

```

=====
character description
=====
'''-''' solid line style
'''--''' dashed line style
'''-.'''' dash-dot line style
'''.'''' dotted line style
=====

```

Example format strings::

```

'b' # blue markers with default shape
'or' # red circles
'g' # green solid line
'--' # dashed line with default color
'^k:' # black triangle_up markers connected by a dotted line

```

## **\*\*Colors\*\***

The supported color abbreviations are the single letter codes

```

=====
character color
=====
'''b''' blue
'''g''' green
'''r''' red
'''c''' cyan
'''m''' magenta
'''y''' yellow
'''k''' black
'''w''' white
=====

```

and the ``CN`` colors that index into the default property cycle.

If the color is the only part of the format string, you can additionally use any `matplotlib.colors` spec, e.g. full names (`'green'`) or hex strings (`'#008000'`).

## matplotlib.pyplot.plot\_date

```
plot_date(x: 'ArrayLike', y: 'ArrayLike', fmt: 'str' = 'o', tz: 'str | datetime.tzinfo
| None' = None, xdate: 'bool' = True, ydate: 'bool' = False, *, data=None, **kwargs)
-> 'list[Line2D]'
```

[\*Deprecated\*] Plot coercing the axis to treat floats as dates.

.. deprecated:: 3.9

This method exists for historic reasons and will be removed in version 3.11.

- ``datetime``-like data should directly be plotted using  
`~.Axes.plot``.  
- If you need to plot plain numeric data as `:ref:`date-format`` or  
need to set a timezone, call `~ax.xaxis.axis_date`` /  
`~ax.yaxis.axis_date`` before `~.Axes.plot``. See  
`~.Axis.axis_date``.

Similar to `~.plot``, this plots `*y*` vs. `*x*` as lines or markers.  
However, the axis labels are formatted as dates depending on `*xdate*`  
and `*ydate*`. Note that `~.plot`` will work with `datetime`` and  
`numpy.datetime64`` objects without resorting to this method.

### Parameters

-----

`x, y` : array-like

The coordinates of the data points. If `*xdate*` or `*ydate*` is  
`*True*`, the respective values `*x*` or `*y*` are interpreted as  
`:ref:`Matplotlib dates <date-format>``.

`fmt` : str, optional

The plot format string. For details, see the corresponding  
parameter in `~.plot``.

`tz` : timezone string or `datetime.tzinfo``, default: `:rc:`timezone``

The time zone to use in labeling dates.

`xdate` : bool, default: True

If `*True*`, the `*x*`-axis will be interpreted as Matplotlib dates.

`ydate` : bool, default: False

If `*True*`, the `*y*`-axis will be interpreted as Matplotlib dates.

### Returns

-----

list of `~.Line2D``

Objects representing the plotted data.

## Other Parameters

-----  
data : indexable object, optional

If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

\*x\*, \*y\*

\*\*kwargs

Keyword arguments control the `.Line2D`` properties:

Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None

animated: bool

antialiased or aa: bool

clip\_box: `~matplotlib.transforms.BboxBase`` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

color or c: `:mpltype:`color``

dash\_capstyle: `.CapStyle`` or {'butt', 'projecting', 'round'}

dash\_joinstyle: `.JoinStyle`` or {'miter', 'round', 'bevel'}

dashes: sequence of floats (on/off ink in points) or (None, None)

data: (2, N) array or two 1D arrays

drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}

gapcolor: `:mpltype:`color`` or None

gid: str

in\_layout: bool

label: object

linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

linewidth or lw: float

marker: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``

markeredgecolor or mec: `:mpltype:`color``

markeredgewidth or mew: float

markerfacecolor or mfc: `:mpltype:`color``

markerfacecoloralt or mfcalt: `:mpltype:`color``

markersize or ms: float

markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

mouseover: bool

path\_effects: list of `.AbstractPathEffect``

picker: float or callable[[Artist, Event], tuple[bool, dict]]

pickradius: float

rasterized: bool

sketch\_params: (scale: float, length: float, randomness: float)

snap: bool or None

solid\_capstyle: `.CapStyle`` or {'butt', 'projecting', 'round'}

solid\_joinstyle: `.JoinStyle`` or {'miter', 'round', 'bevel'}

transform: unknown

url: str

visible: bool

xdata: 1D array

ydata: 1D array

zorder: float

See Also

-----

matplotlib.dates : Helper functions on dates.

matplotlib.dates.date2num : Convert dates to num.

matplotlib.dates.num2date : Convert num to dates.

matplotlib.dates.drange : Create an equally spaced sequence of dates.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.plot_date``.

If you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to `.plot_date``. `.plot_date`` will set the default tick locator to `.AutoDateLocator`` (if the tick locator is not already set to a `.DateLocator`` instance) and the default tick formatter to `.AutoDateFormatter`` (if the tick formatter is not already set to a `.DateFormatter`` instance).

.. deprecated:: 3.9

Use `plot` instead.

## matplotlib.pyplot.polar

```
polar(*args, **kwargs) -> 'list[Line2D]'
```

Make a polar plot.

call signature::

`polar(theta, r, [fmt], **kwargs)`

This is a convenience wrapper around `.pyplot.plot``. It ensures that the current Axes is polar (or creates one if needed) and then passes all parameters to ``.pyplot.plot``.

.. note::

When making polar plots using the :ref:`pyplot API <pyplot\_interface>`, ```polar()``` should typically be the first command because that makes sure a polar Axes is created. Using other commands such as ```plt.title()``` before this can lead to the implicit creation of a rectangular Axes, in which case a subsequent ```polar()``` call will fail.

## matplotlib.pyplot.prism

```
prism() -> 'None'
```

Set the colormap to 'prism'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

## matplotlib.pyplot.psd

```
psd(x: 'ArrayLike', *, NFFT: 'int | None' = None, Fs: 'float | None' = None, Fc: 'int | None' = None, detrend: "Literal['none', 'mean', 'linear'] | Callable[[ArrayLike], ArrayLike] | None" = None, window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, noverlap: 'int | None' = None, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None, scale_by_freq: 'bool | None' = None, return_line: 'bool | None' = None, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray] | tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the power spectral density.

The power spectral density :math:`P\_{xx}` by Welch's average periodogram method. The vector  $x$  is divided into  $NFFT$  length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The :math:`|\mathrm{fft}(i)|^2` of each segment :math:`i` are averaged to compute :math:`P\_{xx}`, with a scaling to correct for power loss due to windowing.

If  $\text{len}(x) < NFFT$ , it will be zero padded to  $NFFT$ .

### Parameters

-----  
 $x$  : 1-D array or sequence

Array or sequence containing the data

$F_s$  : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies,  $f_{\text{reqs}}$ , in cycles per time unit.

`window` : callable or ndarray, default: ``.window_hanning``

A function or a vector of length  $NFFT$ . To create window vectors see ``.window_hanning``, ``.window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. This can be different from  $NFFT$ , which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the  $n$  parameter in the call to `~numpy.fft.fft`. The default is None, which sets `pad_to` equal to  $NFFT$

NFFT : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad\_to* for this instead.

detrend : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The `~matplotlib.mlab` module defines `.detrend_none`, `.detrend_mean`, and `.detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `.detrend_none`. 'mean' calls `.detrend_mean`. 'linear' calls `.detrend_linear`.

scale\_by\_freq : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap : int, default: 0 (no overlap)

The number of points of overlap between segments.

Fc : int, default: 0

The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

return\_line : bool, default: False

Whether to include the line object plotted in the returned values.

## Returns

-----

Pxx : 1-D array

The values for the power spectrum  $P_{xx}$  before scaling (real valued).

freqs : 1-D array

The frequencies corresponding to the elements in *Pxx*.

line : `~matplotlib.lines.Line2D`

The line created by this function.

Only returned if *return\_line* is True.

## Other Parameters

-----

data : indexable object, optional

If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`:

*x*

**kwargs**

Keyword arguments control the `.Line2D` properties:

#### Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: ``.CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``.JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

`markersize` or `ms`: float

`markevery`: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

`mouseover`: bool

`path_effects`: list of ``.AbstractPathEffect``

`picker`: float or callable[[Artist, Event], tuple[bool, dict]]

`pickradius`: float

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`solid_capstyle`: ``.CapStyle`` or {'butt', 'projecting', 'round'}

`solid_joinstyle`: ``.JoinStyle`` or {'miter', 'round', 'bevel'}

`transform`: unknown

`url`: str

`visible`: bool

`xdata`: 1D array

`ydata`: 1D array

`zorder`: float

#### See Also

-----

#### specgram

Differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

#### magnitude\_spectrum

Plots the magnitude spectrum.

#### csd

Plots the spectral density between two signals.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.psd``.

For plotting, the power is plotted as  $10\log_{10}(P_{xx})$  for decibels, though `*Pxx*` itself is returned.

#### References

-----

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

## matplotlib.pyplot.quiver

```
quiver(*args, data=None, **kwargs) -> 'Quiver'
```

Plot a 2D field of arrows.

Call signature::

`quiver([X, Y], U, V, [C], /, **kwargs)`

`*X*`, `*Y*` define the arrow locations, `*U*`, `*V*` define the arrow directions, and `*C*` optionally sets the color. The arguments `*X*`, `*Y*`, `*U*`, `*V*`, `*C*` are positional-only.

**\*\*Arrow length\*\***

The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the `*scale*` and `*scale_units*` parameters.

**\*\*Arrow shape\*\***

The arrow shape is determined by `*width*`, `*headwidth*`, `*headlength*` and `*headaxislength*`. See the notes below.

**\*\*Arrow styling\*\***

Each arrow is internally represented by a filled polygon with a default edge linewidth of 0. As a result, an arrow is rather a filled area, not a line with a head, and `.PolyCollection`` properties like `*linewidth*`, `*edgecolor*`, `*facecolor*`, etc. act accordingly.

#### Parameters

-----

X, Y : 1D or 2D array-like, optional

The x and y coordinates of the arrow locations.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of `*U*` and `*V*`.

If `*X*` and `*Y*` are 1D but `*U*`, `*V*` are 2D, `*X*`, `*Y*` are expanded to 2D using `np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of `*U*` and `*V*`.

`U, V` : 1D or 2D array-like

The x and y direction components of the arrow vectors. The interpretation of these components (in data or in screen space) depends on `*angles*`.

`*U*` and `*V*` must have the same number of elements, matching the number of arrow locations in `*X*`, `*Y*`. `*U*` and `*V*` may be masked. Locations masked in any of `*U*`, `*V*`, and `*C*` will not be drawn.

`C` : 1D or 2D array-like, optional

Numeric data that defines the arrow colors by colormapping via `*norm*` and `*cmap*`.

This does not support explicit colors. If you want to set colors directly, use `*color*` instead. The size of `*C*` must match the number of arrow locations.

`angles` : {'uv', 'xy'} or array-like, default: 'uv'

Method for determining the angle of the arrows.

- 'uv': Arrow directions are based on

`:ref:display coordinates <coordinate-systems>`; i.e. a 45° angle will always show up as diagonal on the screen, irrespective of figure or Axes aspect ratio or Axes data ranges. This is useful when the arrows represent a quantity whose direction is not tied to the x and y data coordinates.

If `*U* == *V*` the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).

- 'xy': Arrow direction in data coordinates, i.e. the arrows point from

(x, y) to (x+u, y+v). This is ideal for vector fields or gradient plots where the arrows should directly represent movements or gradients in the x and y directions.

- Arbitrary angles may be specified explicitly as an array of values in degrees, counter-clockwise from the horizontal axis.

In this case `*U*`, `*V*` is only used to determine the length of the arrows.

For example, `angles=[30, 60, 90]` will orient the arrows at 30, 60, and 90 degrees respectively, regardless of the `*U*` and `*V*` components.

Note: inverting a data axis will correspondingly invert the arrows only with `angles='xy'`.

`pivot` : {'tail', 'mid', 'middle', 'tip'}, default: 'tail'

The part of the arrow that is anchored to the `*X*`, `*Y*` grid. The arrow

rotates about this point.

'mid' is a synonym for 'middle'.

scale : float, optional

Scales the length of the arrow inversely.

Number of data values represented by one unit of arrow length on the plot.

For example, if the data represents velocity in meters per second (m/s), the scale parameter determines how many meters per second correspond to one unit of arrow length relative to the width of the plot.

Smaller scale parameter makes the arrow longer.

By default, an autoscaling algorithm is used to scale the arrow length to a reasonable size, which is based on the average vector length and the number of vectors.

The arrow length unit is given by the `scale_units` parameter.

`scale_units` : {'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, default: 'width'

The physical image unit, which is used for rendering the scaled arrow data  $U^*$ ,  $V^*$ .

The rendered arrow length is given by

length in x direction =  $\frac{u}{\mathrm{scale}} \mathrm{scale\_unit}$

length in y direction =  $\frac{v}{\mathrm{scale}} \mathrm{scale\_unit}$

For example, `((u, v) = (0.5, 0))` with `scale=10`, `scale_units="width"` results in a horizontal arrow with a length of  $0.5 / 10 * \text{"width"}$ , i.e. 0.05 times the Axes width.

Supported values are:

- 'width' or 'height': The arrow length is scaled relative to the width or height of the Axes.

For example, `scale_units='width', scale=1.0`, will result in an arrow length of width of the Axes.

- 'dots': The arrow length of the arrows is measured in display dots (pixels).

- 'inches': Arrow lengths are scaled based on the DPI (dots per inch) of the figure. This ensures that the arrows have a consistent physical size on the figure, in inches, regardless of data values or plot scaling.

For example, `((u, v) = (1, 0))` with `scale_units='inches', scale=2` results in a 0.5 inch-long arrow.

- 'x' or 'y': The arrow length is scaled relative to the x or y axis units.

For example, `((u, v) = (0, 1))` with `scale_units='x', scale=1` results in a vertical arrow with the length of 1 x-axis unit.

- 'xy': Arrow length will be same as 'x' or 'y' units.

This is useful for creating vectors in the x-y plane where u and v have the same units as x and y. To plot vectors in the x-y plane with u and v having

the same units as x and y, use ``angles='xy', scale\_units='xy', scale=1``.

Note: Setting `*scale_units*` without setting `scale` does not have any effect because the scale units only differ by a constant factor and that is rescaled through autoscaling.

units : {'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, default: 'width'  
Affects the arrow size (except for the length). In particular, the shaft `*width*` is measured in multiples of this unit.

Supported values are:

- 'width', 'height': The width or height of the Axes.
- 'dots', 'inches': Pixels or inches based on the figure dpi.
- 'x', 'y', 'xy': `*X*`, `*Y*` or `:math:\sqrt{X^2 + Y^2}` in data units.

The following table summarizes how these values affect the visible arrow size under zooming and figure size changes:

=====	=====	=====
units	zoom	figure size change
=====	=====	=====
'x', 'y', 'xy'	arrow size scales	—
'width', 'height'	—	arrow size scales
'dots', 'inches'	—	—
=====	=====	=====

width : float, optional

Shaft width in arrow units. All head parameters are relative to `*width*`.

The default depends on choice of `*units*` above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

headwidth : float, default: 3

Head width as multiple of shaft `*width*`. See the notes below.

headlength : float, default: 5

Head length as multiple of shaft `*width*`. See the notes below.

headaxislength : float, default: 4.5

Head length at shaft intersection as multiple of shaft `*width*`.

See the notes below.

minshaft : float, default: 1

Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible!

minlength : float, default: 1

Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead.

color : `:mpltype:'color'` or list `:mpltype:'color'`, optional

Explicit color(s) for the arrows. If `*C*` has been set, `*color*` has no effect.

This is a synonym for the `~matplotlib.collections.PolyCollection` *facecolor`` parameter.

## Other Parameters

-----  
data : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

**\*\*kwargs :** `~matplotlib.collections.PolyCollection`` properties, optional

All other keyword arguments are passed on to `~matplotlib.collections.PolyCollection``:

### Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: array-like or float or None

animated: bool

antialiased or aa or antialiaseds: bool or list of bools

array: array-like or None

capstyle: `~matplotlib.collections.PolyCollection`.CapStyle`` or {'butt', 'projecting', 'round'}

clim: (vmin: float, vmax: float)

clip\_box: `~matplotlib.transforms.BboxBase`` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

cmap: `~matplotlib.colors.Colormap`` or str or None

color: :mpltype:`color` or list of RGBA tuples

edgecolor or ec or edgecolors: :mpltype:`color` or list of :mpltype:`color` or 'face'

facecolor or facecolors or fc: :mpltype:`color` or list of :mpltype:`color`

figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

gid: str

hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '\*'}

hatch\_linewidth: unknown

in\_layout: bool

joinstyle: `~matplotlib.collections.PolyCollection`.JoinStyle`` or {'miter', 'round', 'bevel'}

label: object

linestyle or dashes or linestyles or ls: str or tuple or list thereof

linewidth or linewidths or lw: float or list of floats

mouseover: bool

norm: `~matplotlib.colors.Normalize`` or str or None

offset\_transform or transOffset: `~matplotlib.transforms.Transform``

offsets: (N, 2) or (2,) array-like

path\_effects: list of `~matplotlib.path.AbstractPathEffect``

paths: list of array-like

picker: None or bool or float or callable

pickradius: float

rasterized: bool

sizes: `~numpy.ndarray`` or None

sketch\_params: (scale: float, length: float, randomness: float)

snap: bool or None

transform: `~matplotlib.transforms.Transform``

url: str

urls: list of str or None

verts: list of array-like

verts\_and\_codes: unknown

visible: bool

zorder: float

## Returns

-----  
``~matplotlib.quiver.Quiver``

## See Also

-----  
`.Axes.quiverkey` : Add a key to a quiver plot.

## Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``~.axes.Axes.quiver``.

## **\*\*Arrow shape\*\***

The arrow is drawn as a polygon using the nodes as shown below. The values `*headwidth*`, `*headlength*`, and `*headaxislength*` are in units of `*width*`.

.. image:: /\_static/quiver\_sizes.svg  
:width: 500px

The defaults give a slightly swept-back arrow. Here are some guidelines how to get other head shapes:

- To make the head a triangle, make `*headaxislength*` the same as `*headlength*`.
- To make the arrow more pointed, reduce `*headwidth*` or increase `*headlength*` and `*headaxislength*`.
- To make the head smaller relative to the shaft, scale down all the head parameters proportionally.
- To remove the head completely, set all `*head*` parameters to 0.
- To get a diamond-shaped head, make `*headaxislength*` larger than `*headlength*`.
- Warning: For `*headaxislength* < (*headlength* / *headwidth*)`, the "headaxis" nodes (i.e. the ones connecting the head with the shaft) will protrude out of the head in forward direction so that the arrow head looks broken.

## matplotlib.pyplot.quiverkey

```
quiverkey(Q: 'Quiver', X: 'float', Y: 'float', U: 'float', label: 'str', **kwargs) -> 'QuiverKey'
```

Add a key to a quiver plot.

The positioning of the key depends on `*X*`, `*Y*`, `*coordinates*`, and `*labelpos*`. If `*labelpos*` is 'N' or 'S', `*X*`, `*Y*` give the position of the middle of the key arrow. If `*labelpos*` is 'E', `*X*`, `*Y*` positions the head, and if `*labelpos*` is 'W', `*X*`, `*Y*` positions the tail; in either of these two cases, `*X*`, `*Y*` is somewhere in the middle of the arrow+label key object.

## Parameters

-----

Q : `~matplotlib.quiver.Quiver``  
A `~.Quiver`` object as returned by a call to `~.Axes.quiver()`.  
X, Y : float  
The location of the key.  
U : float  
The length of the key.  
label : str  
The key label (e.g., length and units of the key).  
angle : float, default: 0  
The angle of the key arrow, in degrees anti-clockwise from the horizontal axis.  
coordinates : {'axes', 'figure', 'data', 'inches'}, default: 'axes'  
Coordinate system and units for \*X\*, \*Y\*: 'axes' and 'figure' are normalized coordinate systems with (0, 0) in the lower left and (1, 1) in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with (0, 0) at the lower left corner.  
color : :mpltype:`color`  
Overrides face and edge colors from \*Q\*.  
labelpos : {'N', 'S', 'E', 'W'}  
Position the label above, below, to the right, to the left of the arrow, respectively.  
labelsep : float, default: 0.1  
Distance in inches between the arrow and the label.  
labelcolor : :mpltype:`color`, default: :rc:`text.color`  
Label color.  
fontproperties : dict, optional  
A dictionary with keyword arguments accepted by the `~matplotlib.font_manager.FontProperties`` initializer:  
\*family\*, \*style\*, \*variant\*, \*size\*, \*weight\*.  
zorder : float  
The zorder of the key. The default is 0.1 above \*Q\*.  
\*\*kwargs  
Any additional keyword arguments are used to override vector properties taken from \*Q\*.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.quiverkey``.

## matplotlib.pyplot.rc

```
rc(group: 'str', **kwargs) -> 'None'
```

Set the current `~.rcParams``. \*group\* is the grouping for the rc, e.g., for `~lines.linewidth`` the group is `~lines``, for `~axes.facecolor``, the group is `~axes``, and so on. Group may also be a list or tuple of group names, e.g., (\*xtick\*, \*ytick\*).  
\*kwargs\* is a dictionary attribute name/value pairs, e.g.,::

```
rc('lines', linewidth=2, color='r')
```

sets the current ``rcParams`` and is equivalent to::

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

```
=====
Alias Property
=====
'lw' 'linewidth'
'ls' 'linestyle'
'c' 'color'
'fc' 'facecolor'
'ec' 'edgecolor'
'mew' 'markeredgewidth'
'aa' 'antialiased'
=====
```

Thus you could abbreviate the above call as::

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows::

```
font = {'family' : 'monospace',
'weight' : 'bold',
'size' : 'larger'}
rc('font', **font) # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use ``matplotlib.style.use('default')`` or `:func:`~matplotlib.rcdefaults`` to restore the default ``rcParams`` after changes.

Notes

-----

.. note::

This is equivalent to ``matplotlib.rc``.

Similar functionality is available by using the normal dict interface, i.e. ``rcParams.update({"lines.linewidth": 2, ...})`` (but ``rcParams.update`` does not support abbreviations or grouping).

## matplotlib.pyplot.rc\_context

```
rc_context(rc: 'dict[str, Any] | None' = None, fname: 'str | pathlib.Path |
os.PathLike | None' = None) -> 'AbstractContextManager[None]'
```

Return a context manager for temporarily changing rcParams.

The `:rc:`backend`` will not be reset by the context manager.

rcParams changed both through the context manager invocation and in the body of the context will be reset on context exit.

#### Parameters

-----

rc : dict

The rcParams to temporarily set.

fname : str or path-like

A file with Matplotlib rc settings. If both `*fname*` and `*rc*` are given, settings from `*rc*` take precedence.

#### See Also

-----

`:ref:`customizing-with-matplotlibrc-files``

#### Notes

-----

.. note::

This is equivalent to ``matplotlib.rc_context``.

#### Examples

-----

Passing explicit values via a dict::

```
with mpl.rc_context({'interactive': False}):
 fig, ax = plt.subplots()
 ax.plot(range(3), range(3))
 fig.savefig('example.png')
 plt.close(fig)
```

Loading settings from a file::

```
with mpl.rc_context(fname='print.rc'):
 plt.plot(x, y) # uses 'print.rc'
```

Setting in the context body::

```
with mpl.rc_context():
 # will be reset
 mpl.rcParams['lines.linewidth'] = 5
 plt.plot(x, y)
```

## matplotlib.pylab.rcdefaults

```
rcdefaults() -> 'None'
```

Restore the ``rcParams`` from Matplotlib's internal default style.

Style-blacklisted ``rcParams`` (defined in

```matplotlib.style.core.STYLE_BLACKLIST```) are not updated.

See Also

`matplotlib.rc_file_defaults`

Restore the ``.rcParams`` from the rc file originally loaded by Matplotlib.

`matplotlib.style.use`

Use a specific style file. Call ```style.use('default')``` to restore the default style.

Notes

.. note::

This is equivalent to `matplotlib.rcdefaults``.

matplotlib.pyplot.rgrids

```
rgrids(radii: 'ArrayLike | None' = None, labels: 'Sequence[str | Text] | None' = None,
angle: 'float | None' = None, fmt: 'str | None' = None, **kwargs) ->
'tuple[list[Line2D], list[Text]]'
```

Get or set the radial gridlines on the current polar plot.

Call signatures::

`lines, labels = rgrids()`

`lines, labels = rgrids(radii, labels=None, angle=22.5, fmt=None, **kwargs)`

When called with no arguments, ``.rgrids`` simply returns the tuple `(*lines*, *labels*)`. When called with arguments, the labels will appear at the specified radial distances and angle.

Parameters

`radii` : tuple with floats

The radii for the radial gridlines

`labels` : tuple with strings or None

The labels to use at each radial gridline. The ``.matplotlib.ticker.ScalarFormatter`` will be used if None.

`angle` : float

The angular position of the radius labels in degrees.

`fmt` : str or None

Format string used in ``.matplotlib.ticker.FormatStrFormatter``.

For example ```%f```.

Returns

`lines` : list of ``.lines.Line2D``

The radial gridlines.

labels : list of ``text.Text``
The tick labels.

Other Parameters

****kwargs**
kwargs are optional ``Text`` properties for the labels.

See Also

`.pyplot.thetagrids`
`.projections.polar.PolarAxes.set_rgrids`
`.Axis.get_gridlines`
`.Axis.get_ticklabels`

Examples

::

set the locations of the radial gridlines
lines, labels = rgrids((0.25, 0.5, 1.0))

set the locations and labels of the radial gridlines
lines, labels = rgrids((0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry'))

matplotlib.pyplot.savefig

```
savefig(*args, **kwargs) -> 'None'
```

Save the current figure as an image or vector graphic to a file.

Call signature::

```
savefig(fname, *, transparent=None, dpi='figure', format=None,  
metadata=None, bbox_inches=None, pad_inches=0.1,  
facecolor='auto', edgecolor='auto', backend=None,  
**kwargs  
)
```

The available output formats depend on the backend being used.

Parameters

fname : str or path-like or binary file-like
A path, or a Python file-like object, or
possibly some backend-dependent object such as
``matplotlib.backends.backend_pdf.PdfPages``.

If ***format*** is set, it determines the output format, and the file
is saved as ***fname***. Note that ***fname*** is used verbatim, and there
is no attempt to make the extension, if any, of ***fname*** match
format, and no extension is appended.

If **format** is not set, then the format is inferred from the extension of **fname**, if there is one. If **format** is not set and **fname** has no extension, then the file is saved with `:rc:`savefig.format`` and the appropriate extension is appended to **fname**.

Other Parameters

`transparent : bool, default: :rc:`savefig.transparent``
If **True**, the Axes patches will all be transparent; the Figure patch will also be transparent unless **facecolor** and/or **edgecolor** are specified via kwargs.

If **False** has no effect and the color of the Axes and Figure patches are unchanged (unless the Figure patch is specified via the **facecolor** and/or **edgecolor** keyword arguments in which case those colors are used).

The transparency of these patches will be restored to their original values upon exit of this function.

This is useful, for example, for displaying a plot on top of a colored background on a web page.

`dpi : float or 'figure', default: :rc:`savefig.dpi``
The resolution in dots per inch. If 'figure', use the figure's dpi value.

`format : str`
The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under **fname**.

`metadata : dict, optional`
Key/value pairs to store in the image metadata. The supported keys and defaults depend on the image format and backend:

- 'png' with Agg backend: See the parameter ```metadata``` of `~.FigureCanvasAgg.print_png``.
- 'pdf' with pdf backend: See the parameter ```metadata``` of `~.backend_pdf.PdfPages``.
- 'svg' with svg backend: See the parameter ```metadata``` of `~.FigureCanvasSVG.print_svg``.
- 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

Not supported for 'pgf', 'raw', and 'rgba' as those formats do not support embedding metadata.

Does not currently support 'jpg', 'tiff', or 'webp', but may include embedding EXIF metadata in the future.

`bbox_inches : str or `~.Bbox`, default: :rc:`savefig.bbox``
Bounding box in inches: only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

`pad_inches : float or 'layout', default: :rc:`savefig.pad_inches``
Amount of padding in inches around the figure when `bbox_inches` is

'tight'. If 'layout' use the padding from the constrained or compressed layout engine; ignored if one of those engines is not in use.

facecolor : :mpltype:`color` or 'auto', default: :rc:`savefig.facecolor`
The facecolor of the figure. If 'auto', use the current figure facecolor.

edgecolor : :mpltype:`color` or 'auto', default: :rc:`savefig.edgecolor`
The edgecolor of the figure. If 'auto', use the current figure edgecolor.

backend : str, optional
Use a non-default backend to render the file, e.g. to render a png file with the "cairo" backend rather than the default "agg", or a pdf file with the "pgf" backend rather than the default "pdf". Note that the default backend is normally sufficient. See :ref:`the-builtin-backends` for a list of valid backends for each file format. Custom backends can be referenced as "module://...".

orientation : {'landscape', 'portrait'}
Currently only supported by the postscript backend.

papertype : str
One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

bbox_extra_artists : list of ~matplotlib.artist.Artist, optional
A list of extra artists that will be considered when the tight bbox is calculated.

pil_kwargs : dict, optional
Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.Figure.savefig``.

matplotlib.pyplot.sca

```
sca(ax: 'Axes') -> 'None'
```

Set the current Axes to *ax* and the current Figure to the parent of *ax*.

matplotlib.pyplot.scatter

```
scatter(x: 'float | ArrayLike', y: 'float | ArrayLike', s: 'float | ArrayLike | None'
= None, c: 'ArrayLike | Sequence[ColorType] | ColorType | None' = None, *, marker:
'MarkerType | None' = None, cmap: 'str | Colormap | None' = None, norm: 'str |
Normalize | None' = None, vmin: 'float | None' = None, vmax: 'float | None' = None,
alpha: 'float | None' = None, linewidths: 'float | Sequence[float] | None' = None,
edgecolors: "Literal['face', 'none'] | ColorType | Sequence[ColorType] | None" = None,
colorizer: 'Colorizer | None' = None, plotnonfinite: 'bool' = False, data=None,
**kwargs) -> 'PathCollection'
```

A scatter plot of *y* vs. *x* with varying marker size and/or color.

Parameters

x, y : float or array-like, shape (n,)

The data positions.

s : float or array-like, shape (n,), optional

The marker size in points**2 (typographic points are 1/72 in.).

Default is ``rcParams['lines.markersize'] ** 2``.

The linewidth and edgecolor can visually interact with the marker size, and can lead to artifacts if the marker size is smaller than the linewidth.

If the linewidth is greater than 0 and the edgecolor is anything but *'none'*, then the effective size of the marker will be increased by half the linewidth because the stroke will be centered on the edge of the shape.

To eliminate the marker edge either set *linewidth=0* or *edgecolor='none'*.

c : array-like or list of :mpltype:`color` or :mpltype:`color`, optional

The marker colors. Possible values:

- A scalar or sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2D array in which the rows are RGB or RGBA.
- A sequence of colors of length n.
- A single color format string.

Note that *c* should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. If you want to specify the same RGB or RGBA value for all points, use a 2D array with a single row. Otherwise, value-matching will have precedence in case of a size matching with *x* and *y*.

If you wish to specify a single color for all points prefer the *color* keyword argument.

Defaults to `None`. In that case the marker color is determined by the value of *color*, *facecolor* or *facecolors*. In case those are not specified or `None`, the marker color is determined by the next color of the ``Axes``' current "shape and fill" color cycle. This cycle defaults to :rc:`axes.prop_cycle`.

marker : `~.markers.MarkerStyle``, default: `:rc:`scatter.marker``

The marker style. `*marker*` can be either an instance of the class or the text shorthand for a particular marker.

See `:mod:`matplotlib.markers`` for more information about marker styles.

cmap : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*c*` is RGB(A).

norm : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call ``matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*c*` is RGB(A).

vmin, vmax : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

This parameter is ignored if `*c*` is RGB(A).

alpha : float, default: None

The alpha blending value, between 0 (transparent) and 1 (opaque).

linewidths : float or array-like, default: `:rc:`lines.linewidth``

The linewidth of the marker edges. Note: The default `*edgecolors*` is 'face'. You may want to change this as well.

edgecolors : {'face', 'none', `*None*`} or `:mpltype:`color`` or list of `:mpltype:`color``, default: `:rc:`scatter.edgecolors``

The edge color of the marker. Possible values:

- 'face': The edge color will always be the same as the face color.
- 'none': No patch boundary will be drawn.
- A color or sequence of colors.

For non-filled markers, `*edgecolors*` is ignored. Instead, the color is determined like with 'face', i.e. from `*c*`, `*colors*`, or `*facecolors*`.

colorizer : `~matplotlib.colorbar.Colorizer`` or None, default: None
The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

This parameter is ignored if `*c*` is RGB(A).

plotnonfinite : bool, default: False
Whether to plot points with nonfinite `*c*` (i.e. ```inf```, ```-inf``` or ```nan```). If ```True``` the points are drawn with the `*bad*` colormap color (see ``.Colormap.set_bad``).

Returns

`~matplotlib.collections.PathCollection``

Other Parameters

data : indexable object, optional
If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*y*`, `*s*`, `*linewidths*`, `*edgecolors*`, `*c*`, `*facecolor*`, `*facecolors*`, `*color*`
`**kwargs` : `~matplotlib.collections.PathCollection`` properties

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: array-like or float or None

animated: bool

antialiased or aa or antialiaseds: bool or list of bools

array: array-like or None

capstyle: ``.CapStyle`` or `{'butt', 'projecting', 'round'}`

clim: (vmin: float, vmax: float)

clip_box: `~matplotlib.transforms.BboxBase`` or None

clip_on: bool

clip_path: Patch or (Path, Transform) or None

cmap: ``.Colormap`` or str or None

color: :mpltype:```color``` or list of RGBA tuples

edgecolor or ec or edgecolors: :mpltype:```color``` or list of :mpltype:```color``` or 'face'

facecolor or facecolors or fc: :mpltype:```color``` or list of :mpltype:```color```

figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

gid: str

hatch: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`

hatch_linewidth: unknown

in_layout: bool

joinstyle: ``.JoinStyle`` or `{'miter', 'round', 'bevel'}`

label: object

linestyle or dashes or linestyles or ls: str or tuple or list thereof

linewidth or linewidths or lw: float or list of floats

mouseover: bool

norm: ``.Normalize`` or str or None

offset_transform or transOffset: ``.Transform``

offsets: (N, 2) or (2,) array-like

path_effects: list of ``.AbstractPathEffect``

paths: unknown

picker: None or bool or float or callable
pickradius: float
rasterized: bool
sizes: `numpy.ndarray` or None
sketch_params: (scale: float, length: float, randomness: float)
snap: bool or None
transform: `~matplotlib.transforms.Transform`
url: str
urls: list of str or None
visible: bool
zorder: float

See Also

plot : To plot scatter plots when markers are identical in size and color.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.scatter`.

* The `~.plot` function will be faster for scatterplots where markers don't vary in size or color.

* Any or all of `*x*`, `*y*`, `*s*`, and `*c*` may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.

* Fundamentally, scatter works with 1D arrays; `*x*`, `*y*`, `*s*`, and `*c*` may be input as N-D arrays, but within scatter they will be flattened. The exception is `*c*`, which will be flattened only if its size matches the size of `*x*` and `*y*`.

matplotlib.pyplot.sci

```
sci(im: 'ColorizingArtist') -> 'None'
```

Set the current image.

This image will be the target of colormap functions like `~pyplot.viridis`, and other functions such as `~.pyplot.clim`. The current image is an attribute of the current Axes.

matplotlib.pyplot.semilogx

```
semilogx(*args, **kwargs) -> 'list[Line2D]'
```

Make a plot with log scaling on the x-axis.

Call signatures::

```
semilogx([x], y, [fmt], data=None, **kwargs)
semilogx([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around `.plot` which additionally changes the x-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters `*base*`, `*subs*`, and `*nonpositive*` control the x-axis properties. They are just forwarded to `.Axes.set_xscale`.

Parameters

`base` : float, default: 10
Base of the x logarithm.

`subs` : array-like, optional
The location of the minor xticks. If `*None*`, reasonable locations are automatically chosen depending on the number of decades in the plot. See `.Axes.set_xscale` for details.

`nonpositive` : {'mask', 'clip'}, default: 'clip'
Non-positive values in x can be masked as invalid, or clipped to a very small positive number.

`**kwargs`
All parameters supported by `.plot`.

Returns

list of `.Line2D`
Objects representing the plotted data.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.semilogx`.

matplotlib.pyplot.semilogy

```
semilogy(*args, **kwargs) -> 'list[Line2D]'
```

Make a plot with log scaling on the y-axis.

Call signatures::

```
semilogy([x], y, [fmt], data=None, **kwargs)
semilogy([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around `.plot` which additionally changes the y-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters `*base*`, `*subs*`, and `*nonpositive*` control the y-axis properties. They are just forwarded to `.Axes.set_yscale``.

Parameters

`base` : float, default: 10

Base of the y logarithm.

`subs` : array-like, optional

The location of the minor yticks. If `*None*`, reasonable locations are automatically chosen depending on the number of decades in the plot. See `.Axes.set_yscale`` for details.

`nonpositive` : {'mask', 'clip'}, default: 'clip'

Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

`**kwargs`

All parameters supported by `.plot``.

Returns

list of `.Line2D``

Objects representing the plotted data.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.semilogy``.

matplotlib.pyplot.set_cmap

```
set_cmap(cmap: 'Colormap | str') -> 'None'
```

Set the default colormap, and applies it to the current image if any.

Parameters

`cmap` : `~matplotlib.colors.Colormap`` or str

A colormap instance or the name of a registered colormap.

See Also

`colormaps`

`get_cmap`

matplotlib.pyplot.set_loglevel

```
set_loglevel(*args, **kwargs) -> 'None'
```

Configure Matplotlib's logging levels.

Matplotlib uses the standard library `logging` framework under the root logger 'matplotlib'. This is a helper function to:

- set Matplotlib's root logger level
- set the root logger handler's level, creating the handler if it does not exist yet

Typically, one should call `set_loglevel("info")` or `set_loglevel("debug")` to get additional debugging information.

Users or applications that are installing their own logging handlers may want to directly manipulate `logging.getLogger('matplotlib')` rather than use this function.

Parameters

level : {"notset", "debug", "info", "warning", "error", "critical"}
The log level of the handler.

Notes

.. note::

This is equivalent to `matplotlib.set_loglevel`.

The first time this function is called, an additional handler is attached to Matplotlib's root handler; this handler is reused every time and this function simply manipulates the logger and handler's level.

matplotlib.pyplot.setp

```
setp(obj, *args, **kwargs)
```

Set one or more properties on an `Artist`, or list allowed values.

Parameters

obj : `matplotlib.artist.Artist` or list of `Artist`

The artist(s) whose properties are being set or queried. When setting properties, all artists are affected; when querying the allowed values, only the first instance in the sequence is queried.

For example, two lines can be made thicker and red with a single call:

```
>>> x = arange(0, 1, 0.01)
>>> lines = plot(x, sin(2*pi*x), x, sin(4*pi*x))
>>> setp(lines, linewidth=2, color='r')
```

file : file-like, default: `sys.stdout`

Where `setp` writes its output when asked to list allowed values.

```
>>> with open('output.log') as file:
...     setp(line, file=file)
```

The default, ``None``, means `sys.stdout`.

`*args, **kwargs`

The properties to set. The following combinations are supported:

- Set the linestyle of a line to be dashed:

```
>>> line, = plot([1, 2, 3])
>>> setp(line, linestyle='--')
```

- Set multiple properties at once:

```
>>> setp(line, linewidth=2, color='r')
```

- List allowed values for a line's linestyle:

```
>>> setp(line, 'linestyle')
linestyle: {'-', '--', '-.', ':', '|', (offset, on-off-seq), ...}
```

- List all properties that can be set, and their allowed values:

```
>>> setp(line)
agg_filter: a filter function, ...
[long output listing omitted]
```

`setp` also supports MATLAB style string/value pairs. For example, the following are equivalent:

```
>>> setp(lines, 'linewidth', 2, 'color', 'r') # MATLAB style
>>> setp(lines, linewidth=2, color='r') # Python style
```

See Also

`getp`

Notes

.. note::

This is equivalent to `matplotlib.artist.setp`.

matplotlib.pyplot.show

```
show(close=None, block=None)
```

Display all open figures.

Parameters

`block` : bool, optional

Whether to wait for all figures to be closed before returning.

If `True` block and run the GUI main loop until all figure windows are closed.

If `False` ensure that all figure windows are displayed and return immediately. In this case, you are responsible for ensuring that the event loop is running to have responsive figures.

Defaults to `True` in non-interactive mode and to `False` in interactive mode (see `plt.isinteractive`).

See Also

`ion` : Enable interactive mode, which shows / updates the figure after every plotting command, so that calling `show()` is not necessary.
`ioff` : Disable interactive mode.
`savefig` : Save the figure to an image file instead of showing it on screen.

Notes

****Saving figures to file and showing a window at the same time****

If you want an image file as well as a user interface window, use `plt.savefig` before `plt.show`. At the end of (a blocking) `show()` the figure is closed and thus unregistered from pyplot. Calling `plt.savefig` afterwards would save a new and thus empty figure. This limitation of command order does not apply if the show is non-blocking or if you keep a reference to the figure and use `Figure.savefig`.

****Auto-show in jupyter notebooks****

The jupyter backends (activated via `%matplotlib inline`, `%matplotlib notebook`, or `%matplotlib widget`), call `show()` at the end of every cell by default. Thus, you usually don't have to call it explicitly there.

matplotlib.pyplot.silent_list

```
silent_list(type, seq=None)
```

A list with a short `repr()`.

This is meant to be used for a homogeneous list of artists, so that they don't cause long, meaningless output.

Instead of ::

```
[<matplotlib.lines.Line2D object at 0x7f5749fed3c8>,  
<matplotlib.lines.Line2D object at 0x7f5749fed4e0>,  
<matplotlib.lines.Line2D object at 0x7f5758016550>]
```

one will get ::

```
<a list of 3 Line2D objects>
```

If ``self.type`` is None, the type name is obtained from the first item in the list (if any).

matplotlib.pyplot.specgram

```
specgram(x: 'ArrayLike', *, NFFT: 'int | None' = None, Fs: 'float | None' = None, Fc:
'int | None' = None, detrend: "Literal['none', 'mean', 'linear'] |
Callable[[ArrayLike], ArrayLike] | None" = None, window: 'Callable[[ArrayLike],
ArrayLike] | ArrayLike | None' = None, noverlap: 'int | None' = None, cmap: 'str |
Colormap | None' = None, xextent: 'tuple[float, float] | None' = None, pad_to: 'int |
None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None,
scale_by_freq: 'bool | None' = None, mode: "Literal['default', 'psd', 'magnitude',
'angle', 'phase'] | None" = None, scale: "Literal['default', 'linear', 'dB'] | None" =
None, vmin: 'float | None' = None, vmax: 'float | None' = None, data=None, **kwargs)
-> 'tuple[np.ndarray, np.ndarray, np.ndarray, AxesImage]'
```

Plot a spectrogram.

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*. The spectrogram is plotted as a colormap (using imshow).

Parameters

x : 1-D array or sequence

Array or sequence containing the data.

Fs : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

window : callable or ndarray, default: `~.window_hanning`

A function or a vector of length *NFFT*. To create window vectors see `~.window_hanning`, `~.window_none`, `~numpy.blackman`, `~numpy.hamming`, `~numpy.bartlett`, `~scipy.signal`, `~scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to : int, optional

The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `~numpy.fft.fft`. The default is None, which sets *pad_to* equal to *NFFT*

NFFT : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should **NOT** be used to get zero padding, or the scaling of the result will be incorrect; use **pad_to** for this instead.

`detrend` : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the **detrend** parameter is a vector, in Matplotlib it is a function. The `~matplotlib.mlab` module defines `.detrend_none`, `.detrend_mean`, and `.detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `.detrend_none`. 'mean' calls `.detrend_mean`. 'linear' calls `.detrend_linear`.

`scale_by_freq` : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

`mode` : {'default', 'psd', 'magnitude', 'angle', 'phase'}

What sort of spectrum to use. Default is 'psd', which takes the power spectral density. 'magnitude' returns the magnitude spectrum. 'angle' returns the phase spectrum without unwrapping. 'phase' returns the phase spectrum with unwrapping.

`noverlap` : int, default: 128

The number of points of overlap between blocks.

`scale` : {'default', 'linear', 'dB'}

The scaling of the values in the **spec**. 'linear' is no scaling. 'dB' returns the values in dB scale. When **mode** is 'psd', this is dB power ($10 * \log_{10}$). Otherwise, this is dB amplitude ($20 * \log_{10}$). 'default' is 'dB' if **mode** is 'psd' or 'magnitude' and 'linear' otherwise. This must be 'linear' if **mode** is 'angle' or 'phase'.

`Fc` : int, default: 0

The center frequency of **x**, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

`cmap` : `.Colormap`, default: `:rc:~image.cmap`

`xextent` : **None** or (xmin, xmax)

The image extent along the x-axis. The default sets **xmin** to the left border of the first bin (**spectrum** column) and **xmax** to the right border of the last bin. Note that for **noverlap>0** the width of the bins is smaller than those of the segments.

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

x

`vmin, vmax` : float, optional
`vmin` and `vmax` define the data range that the colormap covers.
By default, the colormap covers the complete value range of the data.

****kwargs**

Additional keyword arguments are passed on to `~.axes.Axes.imshow`` which makes the spectrogram image. The origin keyword argument is not supported.

Returns

`spectrum` : 2D array
Columns are the periodograms of successive segments.

`freqs` : 1-D array
The frequencies corresponding to the rows in `*spectrum*`.

`t` : 1-D array
The times corresponding to midpoints of segments (i.e., the columns in `*spectrum*`).

`im` : `.AxesImage``
The image created by `imshow` containing the spectrogram.

See Also

`psd`

Differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

`magnitude_spectrum`

A single spectrum, similar to having a single segment when `*mode*` is 'magnitude'. Plots a line instead of a colormap.

`angle_spectrum`

A single spectrum, similar to having a single segment when `*mode*` is 'angle'. Plots a line instead of a colormap.

`phase_spectrum`

A single spectrum, similar to having a single segment when `*mode*` is 'phase'. Plots a line instead of a colormap.

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for `.axes.Axes.spectrogram``.

The parameters `*detrend*` and `*scale_by_freq*` do only apply when `*mode*` is set to 'psd'.

matplotlib.pyplot.spring

```
spring() -> 'None'
```

Set the colormap to 'spring'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

matplotlib.pyplot.spy

```
spy(Z: 'ArrayLike', *, precision: "float | Literal['present']" = 0, marker: 'str | None' = None, markersize: 'float | None' = None, aspect: "Literal['equal', 'auto'] | float | None" = 'equal', origin: "Literal['upper', 'lower']" = 'upper', **kwargs) -> 'AxesImage'
```

Plot the sparsity pattern of a 2D array.

This visualizes the non-zero values of the array.

Two plotting styles are available: image and marker. Both are available for full arrays, but only the marker style works for `scipy.sparse.spmatrix` instances.

****Image style****

If `*marker*` and `*markersize*` are `*None*`, `~.Axes.imshow` is used. Any extra remaining keyword arguments are passed to this method.

****Marker style****

If `*Z*` is a `scipy.sparse.spmatrix` or `*marker*` or `*markersize*` are `*None*`, a `~.Line2D` object will be returned with the value of marker determining the marker type, and any remaining keyword arguments passed to `~.Axes.plot`.

Parameters

Z : (M, N) array-like

The array to be plotted.

precision : float or 'present', default: 0

If `*precision*` is 0, any non-zero value will be plotted. Otherwise, values of `:math:`|Z| > precision`` will be plotted.

For `scipy.sparse.spmatrix` instances, you can also pass 'present'. In this case any value present in the array will be plotted, even if it is identically zero.

aspect : {'equal', 'auto', None} or float, default: 'equal'

The aspect ratio of the Axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling `~.Axes.set_aspect`. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square.
- 'auto': The Axes is kept fixed and the aspect is adjusted so that the data fit in the Axes. In general, this will result in

non-square pixels.

- `*None*`: Use `:rc:`image.aspect``.

`origin` : `{'upper', 'lower'}`, default: `:rc:`image.origin``

Place the `[0, 0]` index of the array in the upper left or lower left corner of the Axes. The convention 'upper' is typically used for matrices and images.

Returns

`~matplotlib.image.AxesImage`` or `~.Line2D``

The return type depends on the plotting style (see above).

Other Parameters

****kwargs**

The supported additional parameters depend on the plotting style.

For the image style, you can pass the following additional parameters of `~.Axes.imshow``:

- `*cmap*`
- `*alpha*`
- `*url*`
- any `~.Artist`` properties (passed on to the `~.AxesImage``)

For the marker style, you can pass any `~.Line2D`` property except for `*linestyle*`:

Properties:

`agg_filter`: a filter function, which takes a `(m, n, 3)` float array and a dpi value, and returns a `(m, n, 3)` array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or `(Path, Transform)` or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: `~.CapStyle`` or `{'butt', 'projecting', 'round'}`

`dash_joinstyle`: `~.JoinStyle`` or `{'miter', 'round', 'bevel'}`

`dashes`: sequence of floats (on/off ink in points) or `(None, None)`

`data`: `(2, N)` array or two 1D arrays

`drawstyle` or `ds`: `{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}`, default: 'default'

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fillstyle`: `{'full', 'left', 'right', 'bottom', 'top', 'none'}`

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: `{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}`

`linewidth` or `lw`: float

`marker`: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

markerfacecolor or mfc: :mpltype:`color`
 markerfacecoloralt or mfcalt: :mpltype:`color`
 markersize or ms: float
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]
 mouseover: bool
 path_effects: list of `.AbstractPathEffect``
 picker: float or callable[[Artist, Event], tuple[bool, dict]]
 pickradius: float
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 solid_capstyle: `.CapStyle`` or {'butt', 'projecting', 'round'}
 solid_joinstyle: `.JoinStyle`` or {'miter', 'round', 'bevel'}
 transform: unknown
 url: str
 visible: bool
 xdata: 1D array
 ydata: 1D array
 zorder: float

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.spy``.

matplotlib.pyplot.stackplot

```
stackplot(x, *args, labels=(), colors=None, hatch=None, baseline='zero', data=None,
**kwargs)
```

Draw a stacked area plot or a streamgraph.

Parameters

x : (N,) array-like

y : (M, N) array-like

The data can be either stacked or unstacked. Each of the following calls is legal::

`stackplot(x, y)` # where y has shape (M, N) e.g. `y = [y1, y2, y3, y4]`

`stackplot(x, y1, y2, y3, y4)` # where y1, y2, y3, y4 have length N

baseline : {'zero', 'sym', 'wiggle', 'weighted_wiggle'}

Method used to calculate the baseline:

- ```zero```: Constant zero baseline, i.e. a simple stacked plot.

- ```sym```: Symmetric around zero and is sometimes called 'ThemeRiver'.

- ```wiggle```: Minimizes the sum of the squared slopes.

- ```weighted_wiggle```: Does the same but weights to account for size of each layer. It is also called 'Streamgraph'-layout. More details can be found at <http://leebyron.com/streamgraph/>.

labels : list of str, optional

A sequence of labels to assign to each data series. If unspecified, then no labels will be applied to artists.

colors : list of :mpltype:`color`, optional

A sequence of colors to be cycled through and used to color the stacked areas. The sequence need not be exactly the same length as the number of provided *y*, in which case the colors will repeat from the beginning.

If not specified, the colors from the Axes property cycle will be used.

hatch : list of str, default: None

A sequence of hatching styles. See

:doc:`/gallery/shapes_and_collections/hatch_style_reference`.

The sequence will be cycled through for filling the stacked areas from bottom to top.

It need not be exactly the same length as the number of provided *y*, in which case the styles will repeat from the beginning.

.. versionadded:: 3.9

Support for list input

data : indexable object, optional

If given, all parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``.

****kwargs**

All other keyword arguments are passed to `.Axes.fill_between``.

Returns

list of `.PolyCollection``

A list of `.PolyCollection`` instances, one for each element in the stacked area plot.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.stackplot``.

matplotlib.pyplot.stairs

```
stairs(values: 'ArrayLike', edges: 'ArrayLike | None' = None, *, orientation:
"Literal['vertical', 'horizontal']" = 'vertical', baseline: 'float | ArrayLike | None'
= 0, fill: 'bool' = False, data=None, **kwargs) -> 'StepPatch'
```

Draw a stepwise constant function as a line or a filled plot.

edges define the x-axis positions of the steps. *values* the function values between these steps. Depending on *fill*, the function is drawn either as a

continuous line with vertical segments at the edges, or as a filled area.

Parameters

values : array-like

The step heights.

edges : array-like

The step positions, with `len(edges) == len(vals) + 1`, between which the curve takes on vals values.

orientation : {'vertical', 'horizontal'}, default: 'vertical'

The direction of the steps. Vertical means that *values* are along the y-axis, and edges are along the x-axis.

baseline : float, array-like or None, default: 0

The bottom value of the bounding edges or when

`fill=True`, position of lower edge. If *fill* is

True or an array is passed to *baseline*, a closed path is drawn.

If None, then drawn as an unclosed Path.

fill : bool, default: False

Whether the area under the step curve should be filled.

Passing both `fill=True` and `baseline=None` will likely result in undesired filling: the first and last points will be connected with a straight line and the fill will be between this line and the stairs.

Returns

StepPatch : `~matplotlib.patches.StepPatch`

Other Parameters

data : indexable object, optional

If given, all parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`.

****kwargs**

`~matplotlib.patches.StepPatch` properties

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.stairs`.

matplotlib.pyplot.stem

```
stem(*args: 'ArrayLike | str', linefmt: 'str | None' = None, markerfmt: 'str | None' =
None, basefmt: 'str | None' = None, bottom: 'float' = 0, label: 'str | None' = None,
orientation: "Literal['vertical', 'horizontal']" = 'vertical', data=None) ->
'StemContainer'
```

Create a stem plot.

A stem plot draws lines perpendicular to a baseline at each location **locs** from the baseline to **heads**, and places a marker there. For vertical stem plots (the default), the **locs** are **x** positions, and the **heads** are **y** values. For horizontal stem plots, the **locs** are **y** positions, and the **heads** are **x** values.

Call signature::

```
stem([locs,] heads, linefmt=None, markerfmt=None, basefmt=None)
```

The **locs**-positions are optional. **linefmt** may be provided as positional, but all other formats must be provided as keyword arguments.

Parameters

locs : array-like, default: (0, 1, ..., len(heads) - 1)

For vertical stem plots, the x-positions of the stems.

For horizontal stem plots, the y-positions of the stems.

heads : array-like

For vertical stem plots, the y-values of the stem heads.

For horizontal stem plots, the x-values of the stem heads.

linefmt : str, optional

A string defining the color and/or linestyle of the vertical lines:

=====

Character Line Style

=====

``-`` solid line

``--`` dashed line

``-.-`` dash-dot line

``.:`` dotted line

=====

Default: 'C0-', i.e. solid line with the first color of the color cycle.

Note: Markers specified through this parameter (e.g. 'x') will be silently ignored. Instead, markers should be specified using **markerfmt**.

markerfmt : str, optional

A string defining the color and/or shape of the markers at the stem heads. If the marker is not given, use the marker 'o', i.e. filled circles. If the color is not given, use the color from **linefmt**.

basefmt : str, default: 'C3-' ('C2-' in classic mode)

A format string defining the properties of the baseline.

orientation : {'vertical', 'horizontal'}, default: 'vertical'
The orientation of the stems.

bottom : float, default: 0
The y/x-position of the baseline (depending on *orientation*).

label : str, optional
The label to use for the stems in legends.

data : indexable object, optional
If given, all parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``.

Returns

`.StemContainer`
The container may be treated like a tuple
(*markerline*, *stemlines*, *baseline*)

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.stem`.`

.. seealso::

The MATLAB function
``stem` <https://www.mathworks.com/help/matlab/ref/stem.html>`_`
which inspired this method.

matplotlib.pyplot.step

```
step(x: 'ArrayLike', y: 'ArrayLike', *args, where: "Literal['pre', 'post', 'mid']" = 'pre', data=None, **kwargs) -> 'list[Line2D]'
```

Make a step plot.

Call signatures::

```
step(x, y, [fmt], *, data=None, where='pre', **kwargs)  
step(x, y, [fmt], x2, y2, [fmt2], ..., *, where='pre', **kwargs)
```

This is just a thin wrapper around `.plot`` which changes some formatting options. Most of the concepts and parameters of plot can be used here as well.

.. note::

This method uses a standard plot with a step drawstyle: The *x* values are the reference positions and steps extend left/right/both directions depending on *where*.

For the common case where you know the values and edges of the steps, use `~.Axes.stairs`` instead.

Parameters

`x` : array-like

1D sequence of `x` positions. It is assumed, but not checked, that it is uniformly increasing.

`y` : array-like

1D sequence of `y` levels.

`fmt` : str, optional

A format string, e.g. `'g'` for a green line. See ``.plot`` for a more detailed description.

Note: While full format strings are accepted, it is recommended to only specify the color. Line styles are currently ignored (use the keyword argument `*linestyle*` instead). Markers are accepted and plotted on the given positions, however, this is a rarely needed feature for step plots.

`where` : {'pre', 'post', 'mid'}, default: 'pre'

Define where the steps should be placed:

- 'pre': The `y` value is continued constantly to the left from every `*x*` position, i.e. the interval `[(x[i-1], x[i])]` has the value `y[i]`.
- 'post': The `y` value is continued constantly to the right from every `*x*` position, i.e. the interval `[(x[i], x[i+1])]` has the value `y[i]`.
- 'mid': Steps occur half-way between the `*x*` positions.

`data` : indexable object, optional

An object with labelled data. If given, provide the label names to plot in `*x*` and `*y*`.

****kwargs**

Additional parameters are the same as those for ``.plot``.

Returns

list of ``.Line2D``

Objects representing the plotted data.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``.axes.Axes.step``.

matplotlib.pyplot.streamplot

```
streamplot(x, y, u, v, density=1, linewidth=None, color=None, cmap=None, norm=None,
arrowsize=1, arrowstyle='->', minlength=0.1, transform=None, zorder=None,
start_points=None, maxlength=4.0, integration_direction='both',
broken_streamlines=True, *, data=None)
```

Draw streamlines of a vector flow.

Parameters

`x, y` : 1D/2D arrays

Evenly spaced strictly increasing arrays to make a grid. If 2D, all rows of `*x*` must be equal and all columns of `*y*` must be equal; i.e., they must be as if generated by `np.meshgrid(x_1d, y_1d)`.

`u, v` : 2D arrays

`*x*` and `*y*`-velocities. The number of rows and columns must match the length of `*y*` and `*x*`, respectively.

`density` : float or (float, float)

Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid. `*density*` linearly scales this grid.

Each cell in the grid can have, at most, one traversing streamline.

For different densities in each direction, use a tuple

(`density_x`, `density_y`).

`linewidth` : float or 2D array

The width of the streamlines. With a 2D array the line width can be varied across the grid. The array must have the same shape as `*u*` and `*v*`.

`color` : `mpltype:color` or 2D array

The streamline color. If given an array, its values are converted to colors using `*cmap*` and `*norm*`. The array must have the same shape as `*u*` and `*v*`.

`cmap`, `norm`

Data normalization and colormapping parameters for `*color*`; only used if `*color*` is an array of floats. See `~.Axes.imshow` for a detailed description.

`arrowsize` : float

Scaling factor for the arrow size.

`arrowstyle` : str

Arrow style specification.

See `~matplotlib.patches.FancyArrowPatch`.

`minlength` : float

Minimum length of streamline in axes coordinates.

`start_points` : (N, 2) array

Coordinates of starting points for the streamlines in data coordinates (the same coordinates as the `*x*` and `*y*` arrays).

`zorder` : float

The zorder of the streamlines and arrows.

Artists with lower zorder values are drawn first.

`maxlength` : float

Maximum length of streamline in axes coordinates.

`integration_direction` : {'forward', 'backward', 'both'}, default: 'both'

Integrate the streamline in forward, backward or both directions.

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*y*`, `*u*`, `*v*`, `*start_points*`

`broken_streamlines` : boolean, default: True
If False, forces streamlines to continue until they leave the plot domain. If True, they may be terminated if they come too close to another streamline.

Returns

`StreamplotSet`

Container object with attributes

- `lines`: `.LineCollection` of streamlines

- `arrows`: `.PatchCollection` containing `.FancyArrowPatch` objects representing the arrows half-way along streamlines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.streamplot`.

matplotlib.pyplot.subplot

```
subplot(*args, **kwargs) -> 'Axes'
```

Add an Axes to the current figure or retrieve an existing Axes.

This is a wrapper of `.Figure.add_subplot` which provides additional behavior when working with the implicit API (see the notes section).

Call signatures::

```
subplot(nrows, ncols, index, **kwargs)
```

```
subplot(pos, **kwargs)
```

```
subplot(**kwargs)
```

```
subplot(ax)
```

Parameters

`*args` : int, (int, int, `*index*`), or `.SubplotSpec`, default: (1, 1, 1)

The position of the subplot described by one of

- Three integers (`*nrows*`, `*ncols*`, `*index*`). The subplot will take the `*index*` position on a grid with `*nrows*` rows and `*ncols*` columns. `*index*` starts at 1 in the upper left corner and increases to the right. `*index*` can also be a two-tuple specifying the (`*first*`, `*last*`) indices (1-based, and including `*last*`) of the subplot, e.g., `fig.add_subplot(3, 1, (1, 2))` makes a subplot that spans the upper 2/3 of the figure.

- A 3-digit integer. The digits are interpreted as if given separately as three single-digit integers, i.e. `fig.add_subplot(235)` is the same as `fig.add_subplot(2, 3, 5)`. Note that this can only be used if there are no more than 9 subplots.
- A `SubplotSpec`.

`projection` : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional
The projection type of the subplot (`~.axes.Axes`). *str* is the name of a custom projection, see `~matplotlib.projections`. The default None results in a 'rectilinear' projection.

`polar` : bool, default: False
If True, equivalent to `projection='polar'`.

`sharex`, `sharey` : `~matplotlib.axes.Axes`, optional
Share the x or y `~matplotlib.axis` with `sharex` and/or `sharey`. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

`label` : str
A label for the returned Axes.

Returns

`~.axes.Axes`

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as `.projections.polar.PolarAxes` for polar projections.

Other Parameters

**kwargs

This method also takes the keyword arguments for the returned Axes base class; except for the *figure* argument. The keyword arguments for the rectilinear base class `~.axes.Axes` can be found in the following table but there might also be other keyword arguments if another projection is used.

Properties:

- `adjustable`: {'box', 'datalim'}
- `agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
- `alpha`: float or None
- `anchor`: (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...}
- `animated`: bool
- `aspect`: {'auto', 'equal'} or float
- `autoscale_on`: bool
- `autoscalex_on`: unknown
- `autoscaley_on`: unknown
- `axes_locator`: Callable[[Axes, Renderer], Bbox]
- `axisbelow`: bool or 'line'
- `box_aspect`: float or None
- `clip_box`: `~matplotlib.transforms.BboxBase` or None
- `clip_on`: bool

clip_path: Patch or (Path, Transform) or None
 facecolor or fc: :mpltype:`color`
 figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`
 forward_navigation_events: bool or "auto"
 frame_on: bool
 gid: str
 in_layout: bool
 label: object
 mouseover: bool
 navigate: bool
 navigate_mode: unknown
 path_effects: list of `~matplotlib.path.AbstractPathEffect`
 picker: None or bool or float or callable
 position: [left, bottom, width, height] or `~matplotlib.transforms.Bbox`
 prop_cycle: `~matplotlib.colors.Cycler`
 rasterization_zorder: float or None
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 subplotspec: unknown
 title: str
 transform: `~matplotlib.transforms.Transform`
 url: str
 visible: bool
 xbound: (lower: float, upper: float)
 xlabel: str
 xlim: (left: float, right: float)
 xmargin: float greater than -0.5
 xscale: unknown
 xticklabels: unknown
 xticks: unknown
 ybound: (lower: float, upper: float)
 ylabel: str
 ylim: (bottom: float, top: float)
 ymargin: float greater than -0.5
 yscale: unknown
 yticklabels: unknown
 yticks: unknown
 zorder: float

Notes

.. versionchanged:: 3.8

In versions prior to 3.8, any preexisting Axes that overlap with the new Axes beyond sharing a boundary was deleted. Deletion does not happen in more recent versions anymore. Use `~.Axes.remove`` explicitly if needed.

If you do not want this behavior, use the `~.Figure.add_subplot`` method or the `~.pyplot.axes`` function instead.

If no `*kwargs*` are passed and there exists an Axes in the location specified by `*args*` then that Axes will be returned rather than a new Axes being created.

If `*kwargs*` are passed and there exists an Axes in the location

specified by `*args*`, the projection type is the same, and the `*kwargs*` match with the existing Axes, then the existing Axes is returned. Otherwise a new Axes is created with the specified parameters. We save a reference to the `*kwargs*` which we use for this comparison. If any of the values in `*kwargs*` are mutable we will not detect the case where they are mutated. In these cases we suggest using `.Figure.add_subplot` and the explicit Axes API rather than the implicit pyplot API.

See Also

`.Figure.add_subplot`
`.pyplot.subplots`
`.pyplot.axes`
`.Figure.subplots`

Examples

::

```
plt.subplot(221)

# equivalent but more general
ax1 = plt.subplot(2, 2, 1)

# add a subplot with no frame
ax2 = plt.subplot(222, frameon=False)

# add a polar subplot
plt.subplot(223, projection='polar')

# add a red subplot that shares the x-axis with ax1
plt.subplot(224, sharex=ax1, facecolor='red')

# delete ax2 from the figure
plt.delaxes(ax2)

# add ax2 to the figure again
plt.subplot(ax2)

# make the first Axes "current" again
plt.subplot(221)
```

matplotlib.pyplot.subplot2grid

```
subplot2grid(shape: 'tuple[int, int]', loc: 'tuple[int, int]', rowspan: 'int' = 1,
             colspan: 'int' = 1, fig: 'Figure | None' = None, **kwargs) -> 'matplotlib.axes.Axes'
```

Create a subplot at a specific location inside a regular grid.

Parameters

`shape` : (int, int)
Number of rows and of columns of the grid in which to place axis.
`loc` : (int, int)

Row number and column number of the axis location within the grid.
 rowspan : int, default: 1
 Number of rows for the axis to span downwards.
 colspan : int, default: 1
 Number of columns for the axis to span to the right.
 fig : `Figure`, optional
 Figure to place the subplot in. Defaults to the current figure.
 **kwargs
 Additional keyword arguments are handed to `~.Figure.add_subplot`.

Returns

`~.axes.Axes`

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as `~.projections.polar.PolarAxes` for polar projections.

Notes

 The following call ::

```
ax = subplot2grid((nrows, ncols), (row, col), rowspan, colspan)
```

is identical to ::

```
fig = gcf()
gs = fig.add_gridspec(nrows, ncols)
ax = fig.add_subplot(gs[row:row+rowspan, col:col+colspan])
```

matplotlib.pyplot.subplot_mosaic

```
subplot_mosaic(mosaic: 'str | list[HashableList[_T]] | list[HashableList[Hashable]]',
*, sharex: 'bool' = False, sharey: 'bool' = False, width_ratios: 'ArrayLike | None' =
None, height_ratios: 'ArrayLike | None' = None, empty_sentinel: 'Any' = '.',
subplot_kw: 'dict[str, Any] | None' = None, gridspec_kw: 'dict[str, Any] | None' =
None, per_subplot_kw: 'dict[str | tuple[str, ...], dict[str, Any]] | dict[_T |
tuple[_T, ...], dict[str, Any]] | dict[Hashable | tuple[Hashable, ...], dict[str,
Any]] | None' = None, **fig_kw: 'Any') -> 'tuple[Figure, dict[str,
matplotlib.axes.Axes]] | tuple[Figure, dict[_T, matplotlib.axes.Axes]] | tuple[Figure,
dict[Hashable, matplotlib.axes.Axes]]'
```

Build a layout of Axes based on ASCII art or nested lists.

This is a helper function to build complex GridSpec layouts visually.

See :ref:`mosaic`
 for an example and full API documentation

Parameters

 mosaic : list of list of {hashable or nested} or str

A visual layout of how you want your Axes to be arranged
 labeled as strings. For example ::

```
x = [['A panel', 'A panel', 'edge'],  
      ['C panel', '.', 'edge']]
```

produces 4 Axes:

- 'A panel' which is 1 row high and spans the first two columns
- 'edge' which is 2 rows high and is on the right edge
- 'C panel' which in 1 row and 1 column wide in the bottom left
- a blank space 1 row and 1 column wide in the bottom center

Any of the entries in the layout can be a list of lists of the same form to create nested layouts.

If input is a str, then it must be of the form ::

```
""  
AAE  
C.E  
""
```

where each character is a column and each line is a row.
This only allows only single character Axes labels and does not allow nesting but is very terse.

sharex, sharey : bool, default: False

If True, the x-axis (*sharex*) or y-axis (*sharey*) will be shared among all subplots. In that case, tick label visibility and axis units behave as for ``subplots``. If False, each subplot's x- or y-axis will be independent.

width_ratios : array-like of length *ncols*, optional

Defines the relative widths of the columns. Each column gets a relative width of ```width_ratios[i] / sum(width_ratios)```.

If not given, all columns will have the same width. Convenience for ```gridspec_kw={'width_ratios': [...]}```.

height_ratios : array-like of length *nrows*, optional

Defines the relative heights of the rows. Each row gets a relative height of ```height_ratios[i] / sum(height_ratios)```.

If not given, all rows will have the same height. Convenience for ```gridspec_kw={'height_ratios': [...]}```.

empty_sentinel : object, optional

Entry in the layout to mean "leave this space empty". Defaults to ```'. '```. Note, if *layout* is a string, it is processed via ```inspect.cleandoc``` to remove leading white space, which may interfere with using white-space as the empty sentinel.

subplot_kw : dict, optional

Dictionary with keywords passed to the ``.Figure.add_subplot`` call used to create each subplot. These values may be overridden by values in *per_subplot_kw*.

per_subplot_kw : dict, optional

A dictionary mapping the Axes identifiers or tuples of identifiers

to a dictionary of keyword arguments to be passed to the `Figure.add_subplot`` call used to create each subplot. The values in these dictionaries have precedence over the values in `*subplot_kw*`.

If `*mosaic*` is a string, and thus all keys are single characters, it is possible to use a single string instead of a tuple as keys; i.e. ```"AB"``` is equivalent to ```("A", "B")```.

.. versionadded:: 3.7

`gridspec_kw` : dict, optional

Dictionary with keywords passed to the `GridSpec`` constructor used to create the grid the subplots are placed on.

****fig_kw**

All additional keyword arguments are passed to the `pyplot.figure`` call.

Returns

`fig : Figure``

The new figure

`dict[label, Axes]`

A dictionary mapping the labels to the Axes objects. The order of the Axes is left-to-right and top-to-bottom of their position in the total layout.

matplotlib.pyplot.subplot_tool

```
subplot_tool(targetfig: 'Figure | None' = None) -> 'SubplotTool | None'
```

Launch a subplot tool window for a figure.

Returns

`matplotlib.widgets.SubplotTool``

matplotlib.pyplot.subplots

```
subplots(nrows: 'int' = 1, ncols: 'int' = 1, *, sharex: "bool | Literal['none', 'all', 'row', 'col']" = False, sharey: "bool | Literal['none', 'all', 'row', 'col']" = False, squeeze: 'bool' = True, width_ratios: 'Sequence[float] | None' = None, height_ratios: 'Sequence[float] | None' = None, subplot_kw: 'dict[str, Any] | None' = None, gridspec_kw: 'dict[str, Any] | None' = None, **fig_kw) -> 'tuple[Figure, Any]'
```

Create a figure and a set of subplots.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

Parameters

`nrows, ncols` : int, default: 1

Number of rows/columns of the subplot grid.

sharex, sharey : bool or {'none', 'all', 'row', 'col'}, default: False
Controls sharing of properties among x (*sharex*) or y (*sharey*) axes:

- True or 'all': x- or y-axis will be shared among all subplots.
- False or 'none': each subplot x- or y-axis will be independent.
- 'row': each subplot row will share an x- or y-axis.
- 'col': each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created. To later turn other subplots' ticklabels on, use `~matplotlib.axes.Axes.tick_params``.

When subplots have a shared axis that has units, calling `~.Axis.set_units`` will update each axis with the new units.

Note that it is not possible to unshare axes.

squeeze : bool, default: True

- If True, extra dimensions are squeezed out from the returned array of `~matplotlib.axes.Axes``:
- if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
- for Nx1 or 1xM subplots, the returned object is a 1D numpy object array of Axes objects.
- for NxM, subplots with N>1 and M>1 are returned as a 2D array.
- If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up being 1x1.

width_ratios : array-like of length *ncols*, optional
Defines the relative widths of the columns. Each column gets a relative width of ```width_ratios[i] / sum(width_ratios)```.
If not given, all columns will have the same width. Equivalent to ```gridspec_kw={'width_ratios': [...]}```.

height_ratios : array-like of length *nrows*, optional
Defines the relative heights of the rows. Each row gets a relative height of ```height_ratios[i] / sum(height_ratios)```.
If not given, all rows will have the same height. Convenience for ```gridspec_kw={'height_ratios': [...]}```.

subplot_kw : dict, optional
Dict with keywords passed to the `~matplotlib.figure.Figure.add_subplot`` call used to create each subplot.

gridspec_kw : dict, optional
Dict with keywords passed to the `~matplotlib.gridspec.GridSpec``

constructor used to create the grid the subplots are placed on.

****fig_kw**

All additional keyword arguments are passed to the ``pyplot.figure`` call.

Returns

fig : ``Figure``

ax : ``~matplotlib.axes.Axes`` or array of Axes

ax can be either a single ``~.axes.Axes`` object, or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the `squeeze` keyword, see above.

Typical idioms for handling the return value are::

```
# using the variable ax for single a Axes
fig, ax = plt.subplots()
```

```
# using the variable axs for multiple Axes
fig, axs = plt.subplots(2, 2)
```

```
# using tuple unpacking for multiple Axes
fig, (ax1, ax2) = plt.subplots(1, 2)
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
```

The names ```ax``` and pluralized ```axs``` are preferred over ```axes``` because for the latter it's not clear if it refers to a single ``~.axes.Axes`` instance or a collection of these.

See Also

`.pyplot.figure`
`.pyplot.subplot`
`.pyplot.axes`
`.Figure.subplots`
`.Figure.add_subplot`

Examples

::

```
# First create some toy data:
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)
```

```
# Create just a figure and only one subplot
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')
```

```
# Create two subplots and unpack the output array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
```

```

ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Create four polar Axes and access them through the returned array
fig, axs = plt.subplots(2, 2, subplot_kw=dict(projection="polar"))
axs[0, 0].plot(x, y)
axs[1, 1].scatter(x, y)

# Share a X axis with each column of subplots
plt.subplots(2, 2, sharex='col')

# Share a Y axis with each row of subplots
plt.subplots(2, 2, sharey='row')

# Share both X and Y axes with all subplots
plt.subplots(2, 2, sharex='all', sharey='all')

# Note that this is the same as
plt.subplots(2, 2, sharex=True, sharey=True)

# Create figure number 10 with a single subplot
# and clears it if it already exists.
fig, ax = plt.subplots(num=10, clear=True)

```

matplotlib.pyplot.subplots_adjust

```

subplots_adjust(left: 'float | None' = None, bottom: 'float | None' = None, right:
'float | None' = None, top: 'float | None' = None, wspace: 'float | None' = None,
hspace: 'float | None' = None) -> 'None'

```

Adjust the subplot layout parameters.

Unset parameters are left unmodified; initial values are given by
:rc:`figure.subplot.[name]`.

.. plot:: _embedded_plots/figure_subplots_adjust.py

Parameters

left : float, optional

The position of the left edge of the subplots,
as a fraction of the figure width.

right : float, optional

The position of the right edge of the subplots,
as a fraction of the figure width.

bottom : float, optional

The position of the bottom edge of the subplots,
as a fraction of the figure height.

top : float, optional

The position of the top edge of the subplots,
as a fraction of the figure height.

wspace : float, optional

The width of the padding between subplots,
as a fraction of the average Axes width.

hspace : float, optional

The height of the padding between subplots,
as a fraction of the average Axes height.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `Figure.subplots_adjust``.

matplotlib.pyplot.summer

```
summer() -> 'None'
```

Set the colormap to 'summer'.

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)`` for more information.

matplotlib.pyplot.suptitle

```
suptitle(t: 'str', **kwargs) -> 'Text'
```

Add a centered super title to the figure.

Parameters

t : str

The super title text.

x : float, default: 0.5

The x location of the text in figure coordinates.

y : float, default: 0.98

The y location of the text in figure coordinates.

horizontalalignment, ha : {'center', 'left', 'right'}, default: center

The horizontal alignment of the text relative to (*x*, *y*).

verticalalignment, va : {'top', 'center', 'bottom', 'baseline'}, default: top

The vertical alignment of the text relative to (*x*, *y*).

fontsize, size : default: :rc:`figure.titlesize`

The font size of the text. See `.Text.set_size`` for possible values.

fontweight, weight : default: :rc:`figure.titleweight`

The font weight of the text. See `.Text.set_weight`` for possible values.

Returns

text

The `.Text`` instance of the super title.

Other Parameters

fontproperties : None or dict, optional

A dict of font properties. If `*fontproperties*` is given the default values for font size and weight are taken from the

`.FontProperties` defaults. :rc:`figure.titlesize` and :rc:`figure.titleweight` are ignored in this case.`

****kwargs**

Additional kwargs are ``matplotlib.text.Text`` properties.

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for ``Figure.suptitle``.

matplotlib.pyplot.switch_backend

```
switch_backend(newbackend: 'str') -> 'None'
```

Set the pyplot backend.

Switching to an interactive backend is possible only if no event loop for another interactive backend has started. Switching to and from non-interactive backends is always possible.

If the new backend is different than the current backend then all open Figures will be closed via ``plt.close('all')``.

Parameters

`newbackend` : str

The case-insensitive name of the backend to use.

matplotlib.pyplot.table

```
table(cellText=None, cellColours=None, cellLoc='right', colWidths=None,
rowLabels=None, rowColours=None, rowLoc='left', colLabels=None, colColours=None,
colLoc='center', loc='bottom', bbox=None, edges='closed', **kwargs)
```

Add a table to an ``~.axes.Axes``.

At least one of `*cellText*` or `*cellColours*` must be specified. These parameters must be 2D lists, in which the outer lists define the rows and the inner list define the column values per row. Each row must have the same number of elements.

The table can optionally have row and column headers, which are configured using `*rowLabels*`, `*rowColours*`, `*rowLoc*` and `*colLabels*`, `*colColours*`, `*colLoc*` respectively.

For finer grained control over tables, use the ``Table`` class and add it to the Axes with ``Axes.add_table``.

Parameters

`cellText` : 2D list of str or pandas.DataFrame, optional

The texts to place into the table cells.

Note: Line breaks in the strings are currently not accounted for and will result in the text exceeding the cell boundaries.

cellColours : 2D list of :mpltype:`color`, optional
The background colors of the cells.

cellLoc : {'right', 'center', 'left'}
The alignment of the text within the cells.

colWidths : list of float, optional
The column widths in units of the axes. If not given, all columns will have a width of $\frac{1}{ncols}$.

rowLabels : list of str, optional
The text of the row header cells.

rowColours : list of :mpltype:`color`, optional
The colors of the row header cells.

rowLoc : {'left', 'center', 'right'}
The text alignment of the row header cells.

colLabels : list of str, optional
The text of the column header cells.

colColours : list of :mpltype:`color`, optional
The colors of the column header cells.

colLoc : {'center', 'left', 'right'}
The text alignment of the column header cells.

loc : str, default: 'bottom'
The position of the cell with respect to **ax**. This must be one of the `~.Table.codes``.

bbox : `~.Bbox`` or [xmin, ymin, width, height], optional
A bounding box to draw the table into. If this is not **None**, this overrides **loc**.

edges : {'closed', 'open', 'horizontal', 'vertical'} or substring of 'BRTL'
The cell edges to be drawn with a line. See also `~.Cell.visible_edges``.

Returns

`~matplotlib.table.Table``
The created table.

Other Parameters

****kwargs**
`~.Table`` properties.

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
 alpha: float or None
 animated: bool
 clip_box: `~matplotlib.transforms.BboxBase`` or None
 clip_on: bool
 clip_path: Patch or (Path, Transform) or None
 figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``
 fontsize: float
 gid: str
 in_layout: bool
 label: object
 mouseover: bool
 path_effects: list of `~.AbstractPathEffect``
 picker: None or bool or float or callable
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 transform: `~matplotlib.transforms.Transform``
 url: str
 visible: bool
 zorder: float

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.table``.

matplotlib.pyplot.text

```
text(x: 'float', y: 'float', s: 'str', fontdict: 'dict[str, Any] | None' = None,
**kwargs) -> 'Text'
```

Add text to the Axes.

Add the text `*s*` to the Axes at location `*x*`, `*y*` in data coordinates, with a default ```horizontalalignment``` on the ```left``` and ```verticalalignment``` at the ```baseline```. See :doc:`/gallery/text_labels_and_annotations/text_alignment`.

Parameters

`x, y` : float

The position to place the text. By default, this is in data coordinates. The coordinate system can be changed using the `*transform*` parameter.

`s` : str

The text.

`fontdict` : dict, default: None

.. admonition:: Discouraged

The use of `*fontdict*` is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `**text(..., **fontdict)**`.

A dictionary to override the default text properties. If `fontdict` is `None`, the defaults are determined by `.rcParams`.

Returns

`.Text``

The created `.Text`` instance.

Other Parameters

`**kwargs` : `~matplotlib.text.Text`` properties.
Other miscellaneous text parameters.

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
`alpha`: float or `None`
`animated`: bool
`antialiased`: bool
`backgroundcolor`: `:mpltype:`color``
`bbox`: dict with properties for `.patches.FancyBboxPatch``
`clip_box`: unknown
`clip_on`: unknown
`clip_path`: unknown
`color` or `c`: `:mpltype:`color``
`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``
`fontfamily` or `family` or `fontname`: {`FONTNAME`, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}
`fontproperties` or `font` or `font_properties`: `.font_manager.FontProperties`` or ``str`` or ``pathlib.Path``
`fontsize` or `size`: float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}
`fontstretch` or `stretch`: {a numeric value in range 0-1000, 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'}
`fontstyle` or `style`: {'normal', 'italic', 'oblique'}
`fontvariant` or `variant`: {'normal', 'small-caps'}
`fontweight` or `weight`: {a numeric value in range 0-1000, 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}
`gid`: str
`horizontalalignment` or `ha`: {'left', 'center', 'right'}
`in_layout`: bool
`label`: object
`linespacing`: float (multiple of font size)
`math_fontfamily`: str
`mouseover`: bool
`multialignment` or `ma`: {'left', 'right', 'center'}
`parse_math`: bool
`path_effects`: list of `.AbstractPathEffect``
`picker`: `None` or bool or float or callable
`position`: (float, float)
`rasterized`: bool
`rotation`: float or {'vertical', 'horizontal'}
`rotation_mode`: {`None`, 'default', 'anchor'}

```

sketch_params: (scale: float, length: float, randomness: float)
snap: bool or None
text: object
transform: ~matplotlib.transforms.Transform
transform_rotates_text: bool
url: str
usetex: bool, default: :rc:`text.usetex`
verticalalignment or va: {'baseline', 'bottom', 'center', 'center_baseline', 'top'}
visible: bool
wrap: bool
x: float
y: float
zorder: float

```

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.text`.

Examples

Individual keyword arguments can be used to override any given parameter::

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords ((0, 0) is lower-left and (1, 1) is upper-right). The example below places text in the center of the Axes::

```
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
... verticalalignment='center', transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword `*bbox*`. `*bbox*` is a dictionary of `~matplotlib.patches.Rectangle` properties. For example::

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

matplotlib.pyplot.thetagrids

```

thetagrids(angles: 'ArrayLike | None' = None, labels: 'Sequence[str | Text] | None' =
None, fmt: 'str | None' = None, **kwargs) -> 'tuple[list[Line2D], list[Text]]'

```

Get or set the theta gridlines on the current polar plot.

Call signatures::

```
lines, labels = thetagrids()
```

```
lines, labels = thetagrids(angles, labels=None, fmt=None, **kwargs)
```

When called with no arguments, `.thetagrids` simply returns the tuple `(*lines*, *labels*)`. When called with arguments, the labels will appear at the specified angles.

Parameters

`angles` : tuple with floats, degrees
The angles of the theta gridlines.

`labels` : tuple with strings or None
The labels to use at each radial gridline. The `.projections.polar.ThetaFormatter` will be used if None.

`fmt` : str or None
Format string used in `matplotlib.ticker.FormatStrFormatter`.
For example `'%f'`. Note that the angle in radians will be used.

Returns

`lines` : list of `.lines.Line2D`
The theta gridlines.

`labels` : list of `.text.Text`
The tick labels.

Other Parameters

`**kwargs`
`*kwargs*` are optional `.Text` properties for the labels.

See Also

`.pyplot.rgrids`
`.projections.polar.PolarAxes.set_thetagrids`
`.Axis.get_gridlines`
`.Axis.get_ticklabels`

Examples

::

```
# set the locations of the angular gridlines
lines, labels = thetagrids(range(45, 360, 90))
```

```
# set the locations and labels of the angular gridlines
lines, labels = thetagrids(range(45, 360, 90), ('NE', 'NW', 'SW', 'SE'))
```

matplotlib.pyplot.tick_params

```
tick_params(axis: "Literal['both', 'x', 'y']" = 'both', **kwargs) -> 'None'
```

Change the appearance of ticks, tick labels, and gridlines.

Tick properties that are not explicitly set using the keyword

arguments remain unchanged unless `*reset*` is True. For the current style settings, see ``Axis.get_tick_params``.

Parameters

`axis` : {'x', 'y', 'both'}, default: 'both'
The axis to which the parameters are applied.
`which` : {'major', 'minor', 'both'}, default: 'major'
The group of ticks to which the parameters are applied.
`reset` : bool, default: False
Whether to reset the ticks to defaults before updating them.

Other Parameters

`direction` : {'in', 'out', 'inout'}
Puts ticks inside the Axes, outside the Axes, or both.
`length` : float
Tick length in points.
`width` : float
Tick width in points.
`color` : :mpltype:`color`
Tick color.
`pad` : float
Distance in points between tick and label.
`labelsize` : float or str
Tick label font size in points or as a string (e.g., 'large').
`labelcolor` : :mpltype:`color`
Tick label color.
`labelfontfamily` : str
Tick label font.
`colors` : :mpltype:`color`
Tick color and label color.
`zorder` : float
Tick and label zorder.
`bottom`, `top`, `left`, `right` : bool
Whether to draw the respective ticks.
`labelbottom`, `labeltop`, `labelleft`, `labelright` : bool
Whether to draw the respective tick labels.
`labelrotation` : float
Tick label rotation.
`grid_color` : :mpltype:`color`
Gridline color.
`grid_alpha` : float
Transparency of gridlines: 0 (transparent) to 1 (opaque).
`grid_linewidth` : float
Width of gridlines in points.
`grid_linestyle` : str
Any valid ``Line2D`` line style spec.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.tick_params``.

Examples

::

```
ax.tick_params(direction='out', length=6, width=2, colors='r',  
grid_color='r', grid_alpha=0.5)
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red. Gridlines will be red and translucent.

matplotlib.pyplot.ticklabel_format

```
ticklabel_format(*, axis: "Literal['both', 'x', 'y']" = 'both', style: "Literal['',  
'sci', 'scientific', 'plain'] | None" = None, scilimits: 'tuple[int, int] | None' =  
None, useOffset: 'bool | float | None' = None, useLocale: 'bool | None' = None,  
useMathText: 'bool | None' = None) -> 'None'
```

Configure the ``ScalarFormatter`` used by default for linear Axes.

If a parameter is not set, the corresponding property of the formatter is left unchanged.

Parameters

axis : {'x', 'y', 'both'}, default: 'both'

The axis to configure. Only major ticks are affected.

style : {'sci', 'scientific', 'plain'}

Whether to use scientific notation.

The formatter default is to use scientific notation.

'sci' is equivalent to 'scientific'.

scilimits : pair of ints (m, n)

Scientific notation is used only for numbers outside the range

$10^{\text{sup:}m}$ to $10^{\text{sup:}n}$ (and only if the formatter is

configured to use scientific notation at all). Use (0, 0) to

include all numbers. Use (m, m) where $m \neq 0$ to fix the order of magnitude to $10^{\text{sup:}m}$.

The formatter default is `:rc:`axes.formatter.limits``.

useOffset : bool or float

If True, the offset is calculated as needed.

If False, no offset is used.

If a numeric value, it sets the offset.

The formatter default is `:rc:`axes.formatter.useoffset``.

useLocale : bool

Whether to format the number using the current locale or using the C (English) locale. This affects e.g. the decimal separator. The formatter default is `:rc:`axes.formatter.use_locale``.

useMathText : bool

Render the offset and scientific notation in mathtext.

The formatter default is :rc:`axes.formatter.use_mathtext`.

Raises

AttributeError

If the current formatter is not a `.ScalarFormatter``.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.ticklabel_format``.

matplotlib.pylab.tight_layout

```
tight_layout(*, pad: 'float' = 1.08, h_pad: 'float | None' = None, w_pad: 'float | None' = None, rect: 'tuple[float, float, float, float] | None' = None) -> 'None'
```

Adjust the padding between and around subplots.

To exclude an artist on the Axes from the bounding box calculation that determines the subplot parameters (i.e. legend, or annotation), set ```a.set_in_layout(False)``` for that artist.

Parameters

pad : float, default: 1.08

Padding between the figure edge and the edges of subplots, as a fraction of the font size.

h_pad, w_pad : float, default: *pad*

Padding (height/width) between edges of adjacent subplots, as a fraction of the font size.

rect : tuple (left, bottom, right, top), default: (0, 0, 1, 1)

A rectangle in normalized figure coordinates into which the whole subplots area (including labels) will fit.

See Also

`.Figure.set_layout_engine`

`.pyplot.tight_layout`

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.Figure.tight_layout``.

matplotlib.pylab.title

```
title(label: 'str', fontdict: 'dict[str, Any] | None' = None, loc: "Literal['left', 'center', 'right'] | None" = None, pad: 'float | None' = None, *, y: 'float | None' = None, **kwargs) -> 'Text'
```

Set a title for the Axes.

Set one of the three available Axes titles. The available titles are positioned above the Axes in the center, flush with the left edge, and flush with the right edge.

Parameters

label : str

Text to use for the title

fontdict : dict

.. admonition:: Discouraged

The use of `*fontdict*` is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking ```set_title(..., **fontdict)```.

A dictionary controlling the appearance of the title text, the default `*fontdict*` is::

```
{'fontsize': rcParams['axes.titlesize'],
'fontweight': rcParams['axes.titleweight'],
'color': rcParams['axes.titlecolor'],
'verticalalignment': 'baseline',
'horizontalalignment': 'loc'}
```

loc : {'center', 'left', 'right'}, default: `:rc:`axes.titlelocation``
Which title to set.

y : float, default: `:rc:`axes.titley``
Vertical Axes location for the title (1.0 is the top). If None (the default) and `:rc:`axes.titley`` is also None, y is determined automatically to avoid decorators on the Axes.

pad : float, default: `:rc:`axes.titlepad``
The offset of the title from the top of the Axes, in points.

Returns

``Text``

The matplotlib text instance representing the title

Other Parameters

`**kwargs` : ``~matplotlib.text.Text`` properties

Other keyword arguments are text properties, see ``Text`` for a list of valid text properties.

Notes

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``axes.Axes.set_title``.

matplotlib.pyplot.tricontour

```
tricontour(*args, **kwargs)
```

Draw contour lines on an unstructured triangular grid.

Call signatures::

```
tricontour(triangulation, z, [levels], ...)
```

```
tricontour(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a `Triangulation` object as the first parameter, or by passing the points `x`, `y` and optionally the `triangles` and a `mask`. See `Triangulation` for an explanation of these parameters. If neither of `triangulation` or `triangles` are given, the triangulation is calculated on the fly.

It is possible to pass `triangles` positionally, i.e. `tricontour(x, y, triangles, z, ...)`. However, this is discouraged. For more clarity, pass `triangles` via keyword argument.

Parameters

`triangulation` : `Triangulation`, optional
An already created triangular grid.

`x`, `y`, `triangles`, `mask`
Parameters defining the triangular grid. See `Triangulation`.
This is mutually exclusive with specifying `triangulation`.

`z` : array-like
The height values over which the contour is drawn. Color-mapping is controlled by `cmap`, `norm`, `vmin`, and `vmax`.

.. note::
All values in `z` must be finite. Hence, nan and inf values must either be removed or `Triangulation.set_mask` be used.

`levels` : int or array-like, optional
Determines the number and positions of the contour lines / regions.

If an int `n`, use `matplotlib.ticker.MaxNLocator`, which tries to automatically choose no more than `n+1` "nice" contour levels between minimum and maximum numeric values of `Z`.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

`~matplotlib.tri.TriContourSet`

Other Parameters

`colors` : `mpltype:color` or list of `mpltype:color`, optional

The colors of the levels, i.e., the contour lines.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `''red''` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value `*None*`), the colormap specified by `*cmap*` will be used.

`alpha` : float, default: 1

The alpha blending value, between 0 (transparent) and 1 (opaque).

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*colors*` is set.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin`, `vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`origin` : {`*None*`, 'upper', 'lower', 'image'}, default: None

Determines the orientation and exact position of `*z*` by specifying the position of ```z[0, 0]```. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: ```z[0, 0]``` is at X=0, Y=0 in the lower left corner.
- 'lower': ```z[0, 0]``` is at X=0.5, Y=0.5 in the lower left corner.

- 'upper': `z[0, 0]` is at $X=N+0.5$, $Y=0.5$ in the upper left corner.
- 'image': Use the value from `rc.image.origin`.

extent : (x0, x1, y0, y1), optional

If `*origin*` is not `*None*`, then `*extent*` is interpreted as in `.imshow`: it gives the outer pixel boundaries. In this case, the position of `z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then `(*x0*, *y0*)` is the position of `z[0, 0]`, and `(*x1*, *y1*)` is the position of `z[-1, -1]`.

This argument is ignored if `*X*` and `*Y*` are specified in the call to `contour`.

locator : `ticker.Locator` subclass, optional

The locator is used to determine the contour levels if they are not given explicitly via `*levels*`.

Defaults to `~.ticker.MaxNLocator`.

extend : {'neither', 'both', 'min', 'max'}, default: 'neither'

Determines the `tricontour`-coloring of values that are outside the `*levels*` range.

If 'neither', values outside the `*levels*` range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the `*levels*` range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the `.Colormap`. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using `.Colormap.set_under` and `.Colormap.set_over`.

.. note::

An existing `.TriContourSet` does not get notified if properties of its colormap are changed. Therefore, an explicit call to `.ContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the `.TriContourSet` because it internally calls `.ContourSet.changed()`.

xunits, yunits : registered units, optional

Override axis units by specifying an instance of a `:class:`matplotlib.units.ConversionInterface``.

antialiased : bool, optional

Enable antialiasing, overriding the defaults. For filled contours, the default is `*True*`. For line contours, it is taken from `rc:lines.antialiased`.

linewidths : float or array-like, default: `rc:contour.linewidth`

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with

the linewidths in the order specified.

If None, this falls back to `:rc:`lines.linewidth``.

`linestyles` : `{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}`, optional
If `*linestyles*` is `*None*`, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from `:rc:`contour.negative_linestyle`` setting.

`*linestyles*` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

Notes

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``axes.Axes.tricontour``.

matplotlib.pyplot.tricontourf

```
tricontourf(*args, **kwargs)
```

Draw contour regions on an unstructured triangular grid.

Call signatures::

```
tricontourf(triangulation, z, [levels], ...)  
tricontourf(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a ``Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. See ``Triangulation`` for an explanation of these parameters. If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly.

It is possible to pass `*triangles*` positionally, i.e. ```tricontourf(x, y, triangles, z, ...)``. However, this is discouraged. For more clarity, pass *triangles* via keyword argument.`

Parameters

`triangulation` : ``Triangulation``, optional
An already created triangular grid.

`x`, `y`, `triangles`, `mask`
Parameters defining the triangular grid. See ``Triangulation``.
This is mutually exclusive with specifying `*triangulation*`.

`z` : array-like
The height values over which the contour is drawn. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

.. note::

All values in `*z*` must be finite. Hence, nan and inf values must either be removed or `~.Triangulation.set_mask`` be used.

levels : int or array-like, optional

Determines the number and positions of the contour lines / regions.

If an int `*n*`, use `~matplotlib.ticker.MaxNLocator``, which tries to automatically choose no more than `*n+1*` "nice" contour levels between between minimum and maximum numeric values of `*Z*`.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

`~matplotlib.tri.TriContourSet``

Other Parameters

colors : `:mpltype:`color`` or list of `:mpltype:`color``, optional

The colors of the levels, i.e., the contour regions.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. ```red``` instead of ```[red]``` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value `*None*`), the colormap specified by `*cmap*` will be used.

alpha : float, default: 1

The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*colors*` is set.

norm : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `~.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `~.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin, vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/*vmax*` when a `*norm*` instance is given (but using a ``str`` `*norm*` name together with `*vmin*/*vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`origin` : {`*None*`, `'upper'`, `'lower'`, `'image'`}, default: `None`

Determines the orientation and exact position of `*z*` by specifying the position of ```z[0, 0]```. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: ```z[0, 0]``` is at `X=0, Y=0` in the lower left corner.
- `'lower'`: ```z[0, 0]``` is at `X=0.5, Y=0.5` in the lower left corner.
- `'upper'`: ```z[0, 0]``` is at `X=N+0.5, Y=0.5` in the upper left corner.
- `'image'`: Use the value from `:rc:'image.origin'`.

`extent` : (`x0`, `x1`, `y0`, `y1`), optional

If `*origin*` is not `*None*`, then `*extent*` is interpreted as in `.imshow``: it gives the outer pixel boundaries. In this case, the position of `z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then `(*x0*, *y0*)` is the position of `z[0, 0]`, and `(*x1*, *y1*)` is the position of `z[-1, -1]`.

This argument is ignored if `*X*` and `*Y*` are specified in the call to `contour`.

`locator` : `ticker.Locator` subclass, optional

The locator is used to determine the contour levels if they are not given explicitly via `*levels*`. Defaults to `~.ticker.MaxNLocator``.

`extend` : {`'neither'`, `'both'`, `'min'`, `'max'`}, default: `'neither'`

Determines the ```tricontourf```-coloring of values that are outside the `*levels*` range.

If `'neither'`, values outside the `*levels*` range are not colored. If `'min'`, `'max'` or `'both'`, color the values below, above or below and above the `*levels*` range.

Values below ```min(levels)``` and above ```max(levels)``` are mapped to the under/over values of the `.Colormap``. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using `.Colormap.set_under`` and `.Colormap.set_over``.

.. note::

An existing `.TriContourSet`` does not get notified if properties of its colormap are changed. Therefore, an explicit call to `.ContourSet.changed()`` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the `.TriContourSet`` because it internally calls `.ContourSet.changed()``.

xunits, yunits : registered units, optional
Override axis units by specifying an instance of a
:class:`matplotlib.units.ConversionInterface`.

antialiased : bool, optional
Enable antialiasing, overriding the defaults. For filled contours, the default is `*True*`. For line contours, it is taken from `:rc:`lines.antialiased``.

hatches : list[str], optional
A list of crosshatch patterns to use on the filled areas.
If None, no hatching will be added to the contour.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.tricontourf``.

`.tricontourf`` fills intervals that are closed at the top; that is, for boundaries `*z1*` and `*z2*`, the filled region is::

$z1 < Z \leq z2$

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

matplotlib.pyplot.tripcolor

```
tripcolor(*args, alpha=1.0, norm=None, cmap=None, vmin=None, vmax=None, shading='flat', facecolors=None, **kwargs)
```

Create a pseudocolor plot of an unstructured triangular grid.

Call signatures::

```
tripcolor(triangulation, c, *, ...)  
tripcolor(x, y, c, *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a `.Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. See `.Triangulation`` for an explanation of these parameters.

It is possible to pass the triangles positionally, i.e. `tripcolor(x, y, triangles, c, ...)``. However, this is discouraged. For more clarity, pass `*triangles*` via keyword argument.

If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly. In this case, it does not make sense to provide colors at the triangle faces via `*c*` or `*facecolors*` because there are multiple possible triangulations for a group of points and you don't know which triangles will be constructed.

Parameters

`triangulation` : ``~.Triangulation``

An already created triangular grid.

`x`, `y`, `triangles`, `mask`

Parameters defining the triangular grid. See ``~.Triangulation``.

This is mutually exclusive with specifying `*triangulation*`.

`c` : array-like

The color values, either for the points or for the triangles. Which one is automatically inferred from the length of `*c*`, i.e. does it match the number of points or the number of triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the keyword argument ```facecolors=c``` instead of just ```c```.

This parameter is position-only.

`facecolors` : array-like, optional

Can be used alternatively to `*c*` to specify colors at the triangle faces. This parameter takes precedence over `*c*`.

`shading` : `{'flat', 'gouraud'}`, default: `'flat'`

If `'flat'` and the color values `*c*` are defined at points, the color values used for each triangle are from the mean `c` of the triangle's three points. If `*shading*` is `'gouraud'` then color values must be defined at points.

`cmap` : str or ``~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

`norm` : str or ``~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the `[0, 1]` range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of ``~matplotlib.colors.Normalize`` or one of its subclasses

(see :ref:`colormapnorms`).

- A scale name, i.e. one of `"linear"`, `"log"`, `"symlog"`, `"logit"`, etc. For a list of available scales, call ``~matplotlib.scale.get_scale_names()``.

In that case, a suitable ``~matplotlib.colors.Normalize`` subclass is dynamically generated and instantiated.

`vmin`, `vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str`` `*norm*` name together with `*vmin*/vmax*` is acceptable).

colorizer : `~matplotlib.colorbar.Colorizer`` or None, default: None
The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

Returns

`~matplotlib.collections.PolyCollection`` or `~matplotlib.collections.TriMesh``
The result depends on `*shading*`: For ```shading='flat'``` the result is a `~.PolyCollection``, for ```shading='gouraud'``` the result is a `~.TriMesh``.

Other Parameters

`**kwargs : ~matplotlib.collections.Collection`` properties

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
`alpha`: array-like or float or None
`animated`: bool
`antialiased` or `aa` or `antialiaseds`: bool or list of bools
`array`: array-like or None
`capstyle`: `~.CapStyle`` or {'butt', 'projecting', 'round'}
`clim`: (vmin: float, vmax: float)
`clip_box`: `~matplotlib.transforms.BboxBase`` or None
`clip_on`: bool
`clip_path`: Patch or (Path, Transform) or None
`cmap`: `~.Colormap`` or str or None
`color`: `:mpltype:`color`` or list of RGBA tuples
`edgecolor` or `ec` or `edgecolors`: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'
`facecolor` or `facecolors` or `fc`: `:mpltype:`color`` or list of `:mpltype:`color``
`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``
`gid`: str
`hatch`: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
`hatch_linewidth`: unknown
`in_layout`: bool
`joinstyle`: `~.JoinStyle`` or {'miter', 'round', 'bevel'}
`label`: object
`linestyle` or `dashes` or `linestyles` or `ls`: str or tuple or list thereof
`linewidth` or `linewidths` or `lw`: float or list of floats
`mouseover`: bool
`norm`: `~.Normalize`` or str or None
`offset_transform` or `transOffset`: `~.Transform``
`offsets`: (N, 2) or (2,) array-like
`path_effects`: list of `~.AbstractPathEffect``
`paths`: unknown
`picker`: None or bool or float or callable
`pickradius`: float
`rasterized`: bool
`sketch_params`: (scale: float, length: float, randomness: float)
`snap`: bool or None
`transform`: `~matplotlib.transforms.Transform``
`url`: str
`urls`: list of str or None
`visible`: bool
`zorder`: float

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.tripcolor``.

matplotlib.pyplot.triplot

```
triplot(*args, **kwargs)
```

Draw an unstructured triangular grid as lines and/or markers.

Call signatures::

```
triplot(triangulation, ...)  
triplot(x, y, [triangles], *, [mask=mask], ...)
```

The triangular grid can be specified either by passing a ``Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly.

Parameters

`triangulation` : ``Triangulation``

An already created triangular grid.

`x`, `y`, `triangles`, `mask`

Parameters defining the triangular grid. See ``Triangulation``.

This is mutually exclusive with specifying `*triangulation*`.

`other_parameters`

All other args and kwargs are forwarded to ``~.Axes.plot``.

Returns

`lines` : ``~matplotlib.lines.Line2D``

The drawn triangles edges.

`markers` : ``~matplotlib.lines.Line2D``

The drawn marker nodes.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.triplot``.

matplotlib.pyplot.twinx

```
twinx(ax: 'matplotlib.axes.Axes | None' = None) -> '_AxesBase'
```

Make and return a second Axes that shares the `*x*`-axis. The new Axes will overlay `*ax*` (or the current Axes if `*ax*` is `*None*`), and its ticks will be

on the right.

Examples

:doc:`gallery/subplots_axes_and_figures/two_scales`

matplotlib.pyplot.twinx

```
twinx(ax: 'matplotlib.axes.Axes | None' = None) -> '_AxesBase'
```

Make and return a second Axes that shares the *y*-axis. The new Axes will overlay *ax* (or the current Axes if *ax* is *None*), and its ticks will be on the top.

Examples

:doc:`gallery/subplots_axes_and_figures/two_scales`

matplotlib.pyplot.uninstall_repl_displayhook

```
uninstall_repl_displayhook() -> 'None'
```

Disconnect from the display hook of the current shell.

matplotlib.pyplot.violinplot

```
violinplot(dataset: 'ArrayLike | Sequence[ArrayLike]', positions: 'ArrayLike | None' = None, *, vert: 'bool | None' = None, orientation: "Literal['vertical', 'horizontal']" = 'vertical', widths: 'float | ArrayLike' = 0.5, showmeans: 'bool' = False, showextrema: 'bool' = True, showmedians: 'bool' = False, quantiles: 'Sequence[float | Sequence[float]] | None' = None, points: 'int' = 100, bw_method: "Literal['scott', 'silverman'] | float | Callable[[GaussianKDE], float] | None" = None, side: "Literal['both', 'low', 'high']" = 'both', data=None) -> 'dict[str, Collection]'
```

Make a violin plot.

Make a violin plot for each column of *dataset* or each vector in sequence *dataset*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, the maximum, and user-specified quantiles.

Parameters

dataset : Array or a sequence of vectors.
The input data.

positions : array-like, default: [1, 2, ..., n]
The positions of the violins; i.e. coordinates on the x-axis for vertical violins (or y-axis for horizontal violins).

vert : bool, optional
.. deprecated:: 3.10
Use *orientation* instead.

If this is given during the deprecation period, it overrides

the **orientation** parameter.

If True, plots the violins vertically.

If False, plots the violins horizontally.

orientation : {'vertical', 'horizontal'}, default: 'vertical'

If 'horizontal', plots the violins horizontally.

Otherwise, plots the violins vertically.

.. versionadded:: 3.10

widths : float or array-like, default: 0.5

The maximum width of each violin in units of the **positions** axis.

The default is 0.5, which is half the available space when using default **positions**.

showmeans : bool, default: False

Whether to show the mean with a line.

showextrema : bool, default: True

Whether to show extrema with a line.

showmedians : bool, default: False

Whether to show the median with a line.

quantiles : array-like, default: None

If not None, set a list of floats in interval [0, 1] for each violin, which stands for the quantiles that will be rendered for that violin.

points : int, default: 100

The number of points to evaluate each of the gaussian kernel density estimations at.

bw_method : {'scott', 'silverman'} or float or callable, default: 'scott'

The method used to calculate the estimator bandwidth. If a

float, this will be used directly as ``kde.factor``. If a

callable, it should take a ``matplotlib.mlab.GaussianKDE`` instance as its only parameter and return a float.

side : {'both', 'low', 'high'}, default: 'both'

'both' plots standard violins. 'low'/'high' only

plots the side below/above the positions value.

data : indexable object, optional

If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

dataset

Returns

dict

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The

dictionary has the following keys:

- ``bodies``: A list of the ``~.collections.PolyCollection`` instances containing the filled area of each violin.
- ``cmeans``: A ``~.collections.LineCollection`` instance that marks the mean values of each of the violin's distribution.
- ``cmins``: A ``~.collections.LineCollection`` instance that marks the bottom of each violin's distribution.
- ``cmaxes``: A ``~.collections.LineCollection`` instance that marks the top of each violin's distribution.
- ``cbars``: A ``~.collections.LineCollection`` instance that marks the centers of each violin's distribution.
- ``cmedians``: A ``~.collections.LineCollection`` instance that marks the median values of each of the violin's distribution.
- ``cquantiles``: A ``~.collections.LineCollection`` instance created to identify the quantile values of each of the violin's distribution.

See Also

`.Axes.violin` : Draw a violin from pre-computed statistics.

`boxplot` : Draw a box and whisker plot.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.violinplot`.

matplotlib.pyplot.viridis

```
viridis() -> 'None'
```

Set the colormap to 'viridis'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

matplotlib.pyplot.vlines

```
vlines(x: 'float | ArrayLike', ymin: 'float | ArrayLike', ymax: 'float | ArrayLike',  
       colors: 'ColorType | Sequence[ColorType] | None' = None, linestyle: 'LineStyleType' =  
       'solid', *, label: 'str' = '', data=None, **kwargs) -> 'LineCollection'
```

Plot vertical lines at each `*x*` from `*ymin*` to `*ymax*`.

Parameters

x : float or array-like
x-indexes where to plot the lines.

ymin, ymax : float or array-like
Respective beginning and end of each line. If scalars are provided, all lines will have the same length.

colors : :mpltype:`color` or list of color, default: :rc:`lines.color`

linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, default: 'solid'

label : str, default: ''

Returns

~matplotlib.collections.LineCollection`

Other Parameters

data : indexable object, optional
If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

x, *ymin*, *ymax*, *colors*
**kwargs : ~matplotlib.collections.LineCollection` properties.

See Also

hlines : horizontal lines
axvline : vertical line across the Axes

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.vlines``.

matplotlib.pyplot.waitforbuttonpress

```
waitforbuttonpress(timeout: 'float' = -1) -> 'None | bool'
```

Blocking call to interact with the figure.

Wait for user input and return True if a key was pressed, False if a mouse button was pressed and None if no input was given within *timeout* seconds. Negative values deactivate *timeout*.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.Figure.waitforbuttonpress``.

matplotlib.pyplot.window_hanning

```
window_hanning(x)
```

Return x times the Hanning (or Hann) window of $\text{len}(x)$.

See Also

window_none : Another window algorithm.

matplotlib.pyplot.window_none

```
window_none(x)
```

No window function; simply return x .

See Also

window_hanning : Another window algorithm.

matplotlib.pyplot.winter

```
winter() -> 'None'
```

Set the colormap to 'winter'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

matplotlib.pyplot.xcorr

```
xcorr(x: 'ArrayLike', y: 'ArrayLike', *, normed: 'bool' = True, detrend: 'Callable[[ArrayLike], ArrayLike]' = , usevlines: 'bool' = True, maxlags: 'int' = 10, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray, LineCollection | Line2D, Line2D | None]'
```

Plot the cross correlation between x and y .

The correlation with lag k is defined as

$\sum_n x[n+k] \cdot y^*[n]$, where y^* is the complex conjugate of y .

Parameters

x, y : array-like of length n

Neither x nor y are run through Matplotlib's unit conversion, so these should be unit-less arrays.

detrend : callable, default: `.mlab.detrend_none` (no detrending)

A detrending function applied to x and y . It must have the signature ::

`detrend(x: np.ndarray) -> np.ndarray`

normed : bool, default: True

If ``True``, input vectors are normalised to unit length.

usevlines : bool, default: True
Determines the plot style.

If ``True``, vertical lines are plotted from 0 to the xcorr value using `.Axes.vlines``. Additionally, a horizontal line is plotted at $y=0$ using `.Axes.axhline``.

If ``False``, markers are plotted at the xcorr values using `.Axes.plot``.

maxlags : int, default: 10
Number of lags to show. If None, will return all $2 * \text{len}(x) - 1$ lags.

Returns

lags : array (length $2 * \text{maxlags} + 1$)
The lag vector.
c : array (length $2 * \text{maxlags} + 1$)
The auto correlation vector.
line : `.LineCollection`` or `.Line2D``
`.Artist`` added to the Axes of the correlation:

- `.LineCollection`` if `*usevlines*` is True.
- `.Line2D`` if `*usevlines*` is False.
b : `~matplotlib.lines.Line2D`` or None
Horizontal line at 0 if `*usevlines*` is True
None `*usevlines*` is False.

Other Parameters

linestyle : `~matplotlib.lines.Line2D`` property, optional
The linestyle for plotting the data points.
Only used if `*usevlines*` is ``False``.

marker : str, default: 'o'
The marker for plotting the data points.
Only used if `*usevlines*` is ``False``.

data : indexable object, optional
If given, the following parameters also accept a string ``s``, which is interpreted as `data[s]` if ``s`` is a key in `data`:

x, *y*

**kwargs
Additional parameters are passed to `.Axes.vlines`` and `.Axes.axhline`` if `*usevlines*` is ``True``; otherwise they are passed to `.Axes.plot``.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.xcorr``.

The cross correlation is performed with ``numpy.correlate`` with ```mode = "full"```.

matplotlib.pyplot.xkcd

```
xkcd(scale: 'float' = 1, length: 'float' = 100, randomness: 'float' = 2) ->
'ExitStack'
```

Turn on ``xkcd`` <<https://xkcd.com/>> ``_`` sketch-style drawing mode.

This will only have an effect on things drawn after this function is called.

For best results, install the ``xkcd`` script <<https://github.com/ipython/xkcd-font/>> ``_`` font; `xkcd`` fonts are not packaged with Matplotlib.

Parameters

scale : float, optional
The amplitude of the wiggle perpendicular to the source line.
length : float, optional
The length of the wiggle along the line.
randomness : float, optional
The scale factor by which the length is shrunk or expanded.

Notes

This function works by a number of rcParams, so it will probably override others you have set before.

If you want the effects of this function to be temporary, it can be used as a context manager, for example::

```
with plt.xkcd():
    # This figure will be in XKCD-style
    fig1 = plt.figure()
    # ...
```

```
# This figure will be in regular style
fig2 = plt.figure()
```

matplotlib.pyplot.xlabel

```
xlabel(xlabel: 'str', fontdict: 'dict[str, Any] | None' = None, labelpad: 'float |
None' = None, *, loc: "Literal['left', 'center', 'right'] | None" = None, **kwargs) ->
'Text'
```

Set the label for the x-axis.

Parameters

xlabel : str

The label text.

`labelpad` : float, default: :rc:`axes.labelpad`
Spacing in points from the Axes bounding box including ticks and tick labels. If None, the previous value is left as is.

`loc` : {'left', 'center', 'right'}, default: :rc:`axis.labellocation`
The label position. This is a high-level alternative for passing parameters `*x*` and `*horizontalalignment*`.

Other Parameters

`**kwargs` : ~matplotlib.text.Text` properties
`.Text`` properties control the appearance of the label.

See Also

`text` : Documents the properties supported by `.Text``.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.set_xlabel``.

matplotlib.pyplot.xlim

```
xlim(*args, **kwargs) -> 'tuple[float, float]'
```

Get or set the x limits of the current Axes.

Call signatures::

```
left, right = xlim() # return the current xlim  
xlim((left, right)) # set the xlim to left, right  
xlim(left, right) # set the xlim to left, right
```

If you do not specify args, you can pass `*left*` or `*right*` as kwargs, i.e.::

```
xlim(right=3) # adjust the right leaving left unchanged  
xlim(left=1) # adjust the left leaving right unchanged
```

Setting limits turns autoscaling off for the x-axis.

Returns

`left, right`
A tuple of the new x-axis limits.

Notes

Calling this function with no arguments (e.g. ```xlim()```) is the pyplot

equivalent of calling `~.Axes.get_xlim`` on the current Axes.
Calling this function with arguments is the pyplot equivalent of calling `~.Axes.set_xlim`` on the current Axes. All arguments are passed though.

matplotlib.pyplot.xscale

```
xscale(value: 'str | ScaleBase', **kwargs) -> 'None'
```

Set the xaxis' scale.

Parameters

value : str or `~.ScaleBase``

The axis scale type to apply. Valid string values are the names of scale classes ("linear", "log", "function",...). These may be the names of any of the :ref:`built-in scales<builtin_scales>` or of any custom scales registered using `matplotlib.scale.register_scale``.

****kwargs**

If **value** is a string, keywords are passed to the instantiation method of the respective class.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.set_xscale``.

matplotlib.pyplot.xticks

```
xticks(ticks: 'ArrayLike | None' = None, labels: 'Sequence[str] | None' = None, *, minor: 'bool' = False, **kwargs) -> 'tuple[list[Tick] | np.ndarray, list[Text]]'
```

Get or set the current tick locations and labels of the x-axis.

Pass no arguments to return the current values without modifying them.

Parameters

ticks : array-like, optional

The list of xtick locations. Passing an empty list removes all xticks.

labels : array-like, optional

The labels to place at the given **ticks** locations. This argument can only be passed if **ticks** is passed as well.

minor : bool, default: False

If ```False```, get/set the major ticks/labels; if ```True```, the minor ticks/labels.

****kwargs**

`~.Text`` properties can be used to control the appearance of the labels.

.. warning::

This only sets the properties of the current ticks, which is only sufficient if you either pass **ticks**, resulting in a

fixed list of ticks, or if the plot is static.

Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).

Use `~.pyplot.tick_params`` instead if possible.

Returns

locs

The list of xtick locations.

labels

The list of xlabel `.Text`` objects.

Notes

Calling this function with no arguments (e.g. ```xticks()```) is the pyplot equivalent of calling `~.Axes.get_xticks`` and `~.Axes.get_xticklabels`` on the current Axes.

Calling this function with arguments is the pyplot equivalent of calling `~.Axes.set_xticks`` and `~.Axes.set_xticklabels`` on the current Axes.

Examples

```
>>> locs, labels = xticks() # Get the current locations and labels.
>>> xticks(np.arange(0, 1, step=0.2)) # Set label locations.
>>> xticks(np.arange(3), ['Tom', 'Dick', 'Sue']) # Set text labels.
>>> xticks([0, 1, 2], ['January', 'February', 'March'],
... rotation=20) # Set text labels and properties.
>>> xticks([]) # Disable xticks.
```

matplotlib.pyplot.ylabel

```
ylabel(ylabel: 'str', fontdict: 'dict[str, Any] | None' = None, labelpad: 'float | None' = None, *, loc: "Literal['bottom', 'center', 'top'] | None" = None, **kwargs) -> 'Text'
```

Set the label for the y-axis.

Parameters

ylabel : str

The label text.

labelpad : float, default: `:rc:`axes.labelpad``

Spacing in points from the Axes bounding box including ticks and tick labels. If None, the previous value is left as is.

loc : {'bottom', 'center', 'top'}, default: `:rc:`yaxis.labellocation``

The label position. This is a high-level alternative for passing parameters `*y*` and `*horizontalalignment*`.

Other Parameters

**kwargs : `~matplotlib.text.Text` properties
`.Text` properties control the appearance of the label.

See Also

text : Documents the properties supported by `.Text`.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.set_ylabel`.

matplotlib.pyplot.ylim

```
ylim(*args, **kwargs) -> 'tuple[float, float]'
```

Get or set the y-limits of the current Axes.

Call signatures::

```
bottom, top = ylim() # return the current ylim  
ylim((bottom, top)) # set the ylim to bottom, top  
ylim(bottom, top) # set the ylim to bottom, top
```

If you do not specify args, you can alternatively pass **bottom** or **top** as kwargs, i.e.::

```
ylim(top=3) # adjust the top leaving bottom unchanged  
ylim(bottom=1) # adjust the bottom leaving top unchanged
```

Setting limits turns autoscaling off for the y-axis.

Returns

bottom, top
A tuple of the new y-axis limits.

Notes

Calling this function with no arguments (e.g. ``ylim()``) is the pyplot equivalent of calling `~.Axes.get_ylim` on the current Axes.
Calling this function with arguments is the pyplot equivalent of calling `~.Axes.set_ylim` on the current Axes. All arguments are passed though.

matplotlib.pyplot.yscale

```
yscale(value: 'str | ScaleBase', **kwargs) -> 'None'
```

Set the yaxis' scale.

Parameters

value : str or ``ScaleBase``

The axis scale type to apply. Valid string values are the names of scale classes ("linear", "log", "function",...). These may be the names of any of the :ref:`built-in scales<builtin_scales>` or of any custom scales registered using ``matplotlib.scale.register_scale``.

****kwargs**

If `*value*` is a string, keywords are passed to the instantiation method of the respective class.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.set_yscale``.

matplotlib.pyplot.yticks

```
yticks(ticks: 'ArrayLike | None' = None, labels: 'Sequence[str] | None' = None, *,
minor: 'bool' = False, **kwargs) -> 'tuple[list[Tick] | np.ndarray, list[Text]]'
```

Get or set the current tick locations and labels of the y-axis.

Pass no arguments to return the current values without modifying them.

Parameters

ticks : array-like, optional

The list of ytick locations. Passing an empty list removes all yticks.

labels : array-like, optional

The labels to place at the given `*ticks*` locations. This argument can only be passed if `*ticks*` is passed as well.

minor : bool, default: False

If `False`, get/set the major ticks/labels; if `True`, the minor ticks/labels.

****kwargs**

``Text`` properties can be used to control the appearance of the labels.

.. warning::

This only sets the properties of the current ticks, which is only sufficient if you either pass `*ticks*`, resulting in a fixed list of ticks, or if the plot is static.

Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).

Use ``~.pyplot.tick_params`` instead if possible.

Returns

locs

The list of ytick locations.

labels

The list of ylabel ``Text`` objects.

Notes

Calling this function with no arguments (e.g. ``yticks()``) is the pyplot equivalent of calling ``~.Axes.get_yticks`` and ``~.Axes.get_yticklabels`` on the current Axes.

Calling this function with arguments is the pyplot equivalent of calling ``~.Axes.set_yticks`` and ``~.Axes.set_yticklabels`` on the current Axes.

Examples

```
>>> locs, labels = yticks() # Get the current locations and labels.
>>> yticks(np.arange(0, 1, step=0.2)) # Set label locations.
>>> yticks(np.arange(3), ['Tom', 'Dick', 'Sue']) # Set text labels.
>>> yticks([0, 1, 2], ['January', 'February', 'March'],
... rotation=45) # Set text labels and properties.
>>> yticks([]) # Disable yticks.
```

matplotlib.pyplot

`pyplot(...)`

``matplotlib.pyplot`` is a state-based interface to matplotlib. It provides an implicit, MATLAB-like, way of plotting. It also opens figures on your screen, and acts as the figure GUI manager.

pyplot is mainly intended for interactive plots and simple cases of programmatic plot generation::

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.arange(0, 5, 0.1)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```

The explicit object-oriented API is recommended for complex plots, though pyplot is still usually used to create the figure and often the Axes in the figure. See ``~.pyplot.figure``, ``~.pyplot.subplots``, and ``~.pyplot.subplot_mosaic`` to create figures, and :doc:`Axes API </api/axes_api>` for the plotting methods on an Axes::

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.arange(0, 5, 0.1)
```

```
y = np.sin(x)
fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
```

See :ref:`api_interfaces` for an explanation of the tradeoffs between the implicit and explicit interfaces.

matplotlib.pyplot.Annotation

```
Annotation(text, xy, xytext=None, xycoords='data', textcoords=None, arrowprops=None,
annotation_clip=None, **kwargs)
```

An `.Annotation` is a `.Text` that can refer to a specific position `*xy*`. Optionally an arrow pointing from the text to `*xy*` can be drawn.

Attributes

`xy`

The annotated position.

`xycoords`

The coordinate system for `*xy*`.

`arrow_patch`

A `.FancyArrowPatch` to point from `*xytext*` to `*xy*`.

matplotlib.pyplot.Arrow

```
Arrow(x, y, dx, dy, *, width=1.0, **kwargs)
```

An arrow patch.

matplotlib.pyplot.Artist

```
Artist()
```

Abstract base class for objects that render into a `FigureCanvas`.

Typically, all visible elements in a figure are subclasses of `Artist`.

matplotlib.pyplot.AutoLocator

```
AutoLocator()
```

Place evenly spaced ticks, with the step size and maximum number of ticks chosen automatically.

This is a subclass of `~matplotlib.ticker.MaxNLocator`, with parameters `*nbins = 'auto'` and `*steps = [1, 2, 2.5, 5, 10]*`.

matplotlib.pyplot.AxLine

```
AxLine(xy1, xy2, slope, **kwargs)
```

A helper class that implements `~.Axes.axline`, by recomputing the artist transform at draw time.

matplotlib.pyplot.Axes

```
Axes(fig, *args, facecolor=None, frameon=True, sharex=None, sharey=None, label='',
      xscale=None, yscale=None, box_aspect=None, forward_navigation_events='auto', **kwargs)
```

An Axes object encapsulates all the elements of an individual (sub-)plot in a figure.

It contains most of the (sub-)plot elements: `~.axis.Axis`, `~.axis.Tick`, `~.lines.Line2D`, `~.text.Text`, `~.patches.Polygon`, etc., and sets the coordinate system.

Like all visible elements in a figure, Axes is an `~.Artist` subclass.

The `~Axes` instance supports callbacks through a `callbacks` attribute which is a `~.cbook.CallbackRegistry` instance. The events you can connect to are `'xlim_changed'` and `'ylim_changed'` and the callback will be called with `func(*ax*)` where `*ax*` is the `~Axes` instance.

.. note::

As a user, you do not instantiate Axes directly, but use Axes creation methods instead; e.g. from `~.pyplot` or `~.Figure`:

`~.pyplot.subplots`, `~.pyplot.subplot_mosaic` or `~.Figure.add_axes`.

matplotlib.pyplot.BackendFilter

```
BackendFilter(value, names=None, *, module=None, qualname=None, type=None, start=1,
              boundary=None)
```

Filter used with :meth:`~matplotlib.backends.registry.BackendRegistry.list_builtin`

.. versionadded:: 3.9

matplotlib.pyplot.Button

```
Button(ax, label, image=None, color='0.85', hovercolor='0.95', *, useblit=True)
```

A GUI neutral button.

For the button to remain responsive you must keep a reference to it. Call `~.on_clicked` to connect to the button.

Attributes

`ax`

The `~.axes.Axes` the button renders into.

`label`

A `~.Text` instance.

`color`

The color of the button when not hovering.

`hovercolor`

The color of the button when hovering.

matplotlib.pyplot.Circle

```
Circle(xy, radius=5, **kwargs)
```

A circle patch.

matplotlib.pyplot.Colorizer

```
Colorizer(cmap=None, norm=None)
```

Data to color pipeline.

This pipeline is accessible via `.Colorizer.to_rgba`` and executed via the `.Colorizer.norm`` and `.Colorizer.cmap`` attributes.

Parameters

cmap: `colorbar.Colorbar` or str or None, default: None
The colormap used to color data.

norm: `colors.Normalize` or str or None, default: None
The normalization used to normalize the data

matplotlib.pyplot.ColorizingArtist

```
ColorizingArtist(colorizer, **kwargs)
```

Base class for artists that make map data to color using a `.colorizer.Colorizer``.

The `.colorizer.Colorizer`` applies data normalization before returning RGBA colors from a `~matplotlib.colors.Colormap``.

matplotlib.pyplot.Colormap

```
Colormap(name, N=256)
```

Baseclass for all scalar to RGBA mappings.

Typically, Colormap instances are used to convert data values (floats) from the interval ```[0, 1]``` to the RGBA color that the respective Colormap represents. For scaling of data into the ```[0, 1]``` interval see `matplotlib.colors.Normalize``. Subclasses of `matplotlib.cm.ScalarMappable`` make heavy use of this ```data -> normalize -> map-to-color``` processing chain.

matplotlib.pyplot.Figure

```
Figure(figsize=None, dpi=None, *, facecolor=None, edgecolor=None, linewidth=0.0, frameon=None, subplotpars=None, tight_layout=None, constrained_layout=None, layout=None, **kwargs)
```

The top level container for all the plot elements.

See `matplotlib.figure`` for an index of class methods.

Attributes

patch

The `.Rectangle`` instance representing the figure background patch.

suppressComposite

For multiple images, the figure will make composite images depending on the renderer option `_image_nocomposite` function. If `*suppressComposite*` is a boolean, this will override the renderer.

matplotlib.pyplot.FigureBase

```
FigureBase(**kwargs)
```

Base class for `.Figure`` and `.SubFigure`` containing the methods that add artists to the figure or subfigure, create Axes, etc.

matplotlib.pyplot.FigureCanvasBase

```
FigureCanvasBase(figure=None)
```

The canvas the figure renders into.

Attributes

`figure` : `~matplotlib.figure.Figure``
A high-level figure instance.

matplotlib.pyplot.FigureManagerBase

```
FigureManagerBase(canvas, num)
```

A backend-independent abstraction of a figure container and controller.

The figure manager is used by pyplot to interact with the window in a backend-independent way. It's an adapter for the real (GUI) framework that represents the visual figure on screen.

The figure manager is connected to a specific canvas instance, which in turn is connected to a specific figure instance. To access a figure manager for a given figure in user code, you typically use `fig.canvas.manager``.

GUI backends derive from this class to translate common operations such as `*show*` or `*resize*` to the GUI-specific code. Non-GUI backends do not support these operations and can just use the base class.

This following basic operations are accessible:

****Window operations****

- `~.FigureManagerBase.show``
- `~.FigureManagerBase.destroy``
- `~.FigureManagerBase.full_screen_toggle``
- `~.FigureManagerBase.resize``

```
- ~.FigureManagerBase.get_window_title`  
- ~.FigureManagerBase.set_window_title`
```

****Key and mouse button press handling****

The figure manager sets up default key and mouse button press handling by hooking up the `~.key_press_handler`` to the matplotlib event system. This ensures the same shortcuts and mouse actions across backends.

****Other operations****

Subclasses will have additional attributes and functions to access additional functionality. This is of course backend-specific. For example, most GUI backends have ```window``` and ```toolbar``` attributes that give access to the native GUI widgets of the respective framework.

Attributes

`canvas : `FigureCanvasBase``
The backend-specific canvas instance.

`num : int or str`
The figure number.

`key_press_handler_id : int`
The default key handler cid, when using the toolmanager.
To disable the default key press handling use::

```
figure.canvas.mpl_disconnect(  
figure.canvas.manager.key_press_handler_id)
```

`button_press_handler_id : int`
The default mouse button handler cid, when using the toolmanager.
To disable the default button press handling use::

```
figure.canvas.mpl_disconnect(  
figure.canvas.manager.button_press_handler_id)
```

matplotlib.pyplot.FixedFormatter

```
FixedFormatter(seq)
```

Return fixed strings for tick labels based only on position, not value.

.. note::
`~.FixedFormatter`` should only be used together with `~.FixedLocator``.
Otherwise, the labels may end up in unexpected positions.

matplotlib.pyplot.FixedLocator

```
FixedLocator(locs, nbins=None)
```

Place ticks at a set of fixed values.

If `*nbins*` is `None` ticks are placed at all values. Otherwise, the `*locs*` array of possible positions will be subsampled to keep the number of ticks $\leq \text{nbins} + 1$. The subsampling will be done to include the smallest absolute value; for example, if zero is included in the array of possibilities, then it will be included in the chosen ticks.

matplotlib.pyplot.FormatStrFormatter

```
FormatStrFormatter(fmt)
```

Use an old-style ('%' operator) format string to format the tick.

The format string should have a single variable format (%) in it. It will be applied to the value (not the position) of the tick.

Negative numeric values (e.g., -1) will use a dash, not a Unicode minus; use `mathtext` to get a Unicode minus by wrapping the format specifier with \$ (e.g. "\$%g\$").

matplotlib.pyplot.Formatter

```
Formatter()
```

Create a string based on a tick value and location.

matplotlib.pyplot.FuncFormatter

```
FuncFormatter(func)
```

Use a user-defined function for formatting.

The function should take in two inputs (a tick value ```x``` and a position ```pos```), and return a string containing the corresponding tick label.

matplotlib.pyplot.GridSpec

```
GridSpec(nrows, ncols, figure=None, left=None, bottom=None, right=None, top=None,
         wspace=None, hspace=None, width_ratios=None, height_ratios=None)
```

A grid layout to place subplots within a figure.

The location of the grid cells is determined in a similar way to ``.SubplotParams`` using `*left*`, `*right*`, `*top*`, `*bottom*`, `*wspace*` and `*hspace*`.

Indexing a `GridSpec` instance returns a ``.SubplotSpec``.

matplotlib.pyplot.IndexLocator

```
IndexLocator(base, offset)
```

Place ticks at every `n`th point plotted.

IndexLocator assumes index plotting; i.e., that the ticks are placed at integer values in the range between 0 and len(data) inclusive.

matplotlib.pyplot.Line2D

```
Line2D(xdata, ydata, *, linewidth=None, linestyle=None, color=None, gapcolor=None,
marker=None, markersize=None, markeredgewidth=None, markeredgewidth=None,
markerfacecolor=None, markerfacecoloralt='none', fillstyle=None, antialiased=None,
dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None,
pickradius=5, drawstyle=None, markevery=None, **kwargs)
```

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create "stepped" lines in various styles.

matplotlib.pyplot.LinearLocator

```
LinearLocator(numticks=None, presets=None)
```

Place ticks at evenly spaced values.

The first time this function is called it will try to set the number of ticks to make a nice tick partitioning. Thereafter, the number of ticks will be fixed so that interactive navigation will be nice

matplotlib.pyplot.Locator

```
Locator()
```

Determine tick locations.

Note that the same locator should not be used across multiple `~matplotlib.axis.Axis` because the locator stores references to the Axis data and view limits.

matplotlib.pyplot.LogFormatter

```
LogFormatter(base=10.0, labelOnlyBase=False, minor_thresholds=None, linthresh=None)
```

Base class for formatting ticks on a log or symlog scale.

It may be instantiated directly, or subclassed.

Parameters

base : float, default: 10.

Base of the logarithm used in all calculations.

labelOnlyBase : bool, default: False

If True, label ticks only at integer powers of base.

This is normally True for major ticks and False for minor ticks.

`minor_thresholds` : (subset, all), default: (1, 0.4)
If `labelOnlyBase` is `False`, these two numbers control the labeling of ticks that are not at integer powers of base; normally these are the minor ticks. The controlling parameter is the log of the axis data range. In the typical case where base is 10 it is the number of decades spanned by the axis, so we can call it 'numdec'. If `numdec <= all`, all minor ticks will be labeled. If `all < numdec <= subset`, then only a subset of minor ticks will be labeled, so as to avoid crowding. If `numdec > subset` then no minor ticks will be labeled.

`linthresh` : None or float, default: None
If a symmetric log scale is in use, its `linthresh` parameter must be supplied here.

Notes

The `set_locs` method must be called to enable the subsetting logic controlled by the `minor_thresholds` parameter.

In some cases such as the colorbar, there is no distinction between major and minor ticks; the tick locations might be set manually, or by a locator that puts ticks at integer powers of base and at intermediate locations. For this situation, disable the `minor_thresholds` logic by using `minor_thresholds=(np.inf, np.inf)`, so that all ticks will be labeled.

To disable labeling of minor ticks when `labelOnlyBase` is `False`, use `minor_thresholds=(0, 0)`. This is the default for the "classic" style.

Examples

To label a subset of minor ticks when the view limits span up to 2 decades, and all of the ticks when zoomed in to 0.5 decades or less, use `minor_thresholds=(2, 0.5)`.

To label all minor ticks when the view limits span up to 1.5 decades, use `minor_thresholds=(1.5, 1.5)`.

matplotlib.pyplot.LogFormatterExponent

```
LogFormatterExponent(base=10.0, labelOnlyBase=False, minor_thresholds=None,
linthresh=None)
```

Format values for log axis using `exponent = log_base(value)`.

matplotlib.pyplot.LogFormatterMathtext

```
LogFormatterMathtext(base=10.0, labelOnlyBase=False, minor_thresholds=None,
linthresh=None)
```

Format values for log axis using `exponent = log_base(value)`.

matplotlib.pyplot.LogLocator

```
LogLocator(base=10.0, subs=(1.0,), *, numticks=None)
```

Place logarithmically spaced ticks.

Places ticks at the values `subs[j] * base**i`.

matplotlib.pyplot.MaxNLocator

```
MaxNLocator(nbins=None, **kwargs)
```

Place evenly spaced ticks, with a cap on the total number of ticks.

Finds nice tick locations with no more than `nbins + 1` ticks being within the view limits. Locations beyond the limits are added to support autoscaling.

matplotlib.pyplot.MouseButton

```
MouseButton(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
```

Enum where members are also (and must be) ints

matplotlib.pyplot.MultipleLocator

```
MultipleLocator(base=1.0, offset=0.0)
```

Place ticks at every integer multiple of a base plus an offset.

matplotlib.pyplot.Normalize

```
Normalize(vmin=None, vmax=None, clip=False)
```

A class which, when called, maps values within the interval `[vmin, vmax]` linearly to the interval `[0.0, 1.0]`. The mapping of values outside `[vmin, vmax]` depends on `*clip*`.

Examples

::

```
x = [-2, -1, 0, 1, 2]
```

```
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=False)
```

```
norm(x) # [-0.5, 0., 0.5, 1., 1.5]
```

```
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=True)
```

```
norm(x) # [0., 0., 0.5, 1., 1.]
```

See Also

:ref:`colormapnorms`

matplotlib.pyplot.NullFormatter

```
NullFormatter()
```

Always return the empty string.

matplotlib.pyplot.NullLocator

```
NullLocator()
```

No ticks

matplotlib.pyplot.PolarAxes

```
PolarAxes(*args, theta_offset=0, theta_direction=1, rlabel_position=22.5, **kwargs)
```

A polar graph projection, where the input dimensions are **theta**, **r**.

Theta starts pointing east and goes anti-clockwise.

matplotlib.pyplot.Polygon

```
Polygon(xy, *, closed=True, **kwargs)
```

A general polygon patch.

matplotlib.pyplot.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point **xy** and its **width** and **height**.

The rectangle extends from `xy[0]` to `xy[0] + width` in x-direction and from `xy[1]` to `xy[1] + height` in y-direction. ::

```
: +-----+
: |
: |
: height |
: |
: |
: (xy)--- width ---->
```

One may picture **xy** as the bottom left corner, but which corner **xy** is actually depends on the direction of the axis and the sign of **width** and **height**; e.g. **xy** would be the bottom right corner if the x-axis was inverted or if **width** was negative.

matplotlib.pyplot.ScalarFormatter

```
ScalarFormatter(useOffset=None, useMathText=None, useLocale=None, *, useTex=None)
```

Format tick values as a number.

Parameters

```
-----
useOffset : bool or float, default: rc:`axes.formatter.useoffset`
Whether to use offset notation. See .set_useOffset.
useMathText : bool, default: rc:`axes.formatter.use_mathtext`
Whether to use fancy math formatting. See .set_useMathText.
useLocale : bool, default: rc:`axes.formatter.use_locale`.
```

Whether to use locale settings for decimal sign and positive sign.
See `.set_useLocale``.
`usetex`` : bool, default: `:rc:`text.usetex``
To enable/disable the use of TeX's math mode for rendering the numbers in the formatter.

.. versionadded:: 3.10

Notes

In addition to the parameters above, the formatting of scientific vs. floating point representation can be configured via `.set_scientific`` and `.set_powerlimits``).

****Offset notation and scientific notation****

Offset notation and scientific notation look quite similar at first sight. Both split some information from the formatted tick values and display it at the end of the axis.

- The scientific notation splits up the order of magnitude, i.e. a multiplicative scaling factor, e.g. ```1e6```.

- The offset notation separates an additive constant, e.g. ```+1e6```. The offset notation label is always prefixed with a ```+``` or ```-``` sign and is thus distinguishable from the order of magnitude label.

The following plot with x limits ```1_000_000``` to ```1_000_010``` illustrates the different formatting. Note the labels at the right edge of the x axis.

.. plot::

```
lim = (1_000_000, 1_000_010)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, gridspec_kw={'hspace': 2})
ax1.set(title='offset notation', xlim=lim)
ax2.set(title='scientific notation', xlim=lim)
ax2.xaxis.get_major_formatter().set_useOffset(False)
ax3.set(title='floating-point notation', xlim=lim)
ax3.xaxis.get_major_formatter().set_useOffset(False)
ax3.xaxis.get_major_formatter().set_scientific(False)
```

matplotlib.pyplot.Slider

```
Slider(ax, label, valmin, valmax, *, valinit=0.5, valfmt=None, closedmin=True,
closedmax=True, slidermin=None, slidermax=None, dragging=True, valstep=None,
orientation='horizontal', initcolor='r', track_color='lightgrey', handle_style=None,
**kwargs)
```

A slider representing a floating point range.

Create a slider from `*valmin*` to `*valmax*` in Axes `*ax*`. For the slider to remain responsive you must maintain a reference to it. Call `:meth:`on_changed`` to connect to the slider event.

Attributes

val : float

Slider value.

matplotlib.pyplot.Subplot

```
Subplot(fig, *args, facecolor=None, frameon=True, sharex=None, sharey=None, label='',
xscale=None, yscale=None, box_aspect=None, forward_navigation_events='auto', **kwargs)
```

An Axes object encapsulates all the elements of an individual (sub-)plot in a figure.

It contains most of the (sub-)plot elements: `~.axis.Axis``, `~.axis.Tick``, `~.lines.Line2D``, `~.text.Text``, `~.patches.Polygon``, etc., and sets the coordinate system.

Like all visible elements in a figure, Axes is an `~.Artist`` subclass.

The `~.Axes`` instance supports callbacks through a `callbacks` attribute which is a `~.cbook.CallbackRegistry`` instance. The events you can connect to are `'xlim_changed'` and `'ylim_changed'` and the callback will be called with `func(*ax*)` where `*ax*` is the `~.Axes`` instance.

.. note::

As a user, you do not instantiate Axes directly, but use Axes creation methods instead; e.g. from `~.pyplot`` or `~.Figure``: `~.pyplot.subplots``, `~.pyplot.subplot_mosaic`` or `~.Figure.add_axes``.

matplotlib.pyplot.SubplotSpec

```
SubplotSpec(gridspec, num1, num2=None)
```

The location of a subplot in a `~.GridSpec``.

.. note::

Likely, you will never instantiate a `~.SubplotSpec`` yourself. Instead, you will typically obtain one from a `~.GridSpec`` using item-access.

Parameters

`gridspec` : `~.matplotlib.gridspec.GridSpec``

The `GridSpec`, which the subplot is referencing.

`num1, num2` : int

The subplot will occupy the `*num1*-th` cell of the given

`*gridspec*`. If `*num2*` is provided, the subplot will span between `*num1*-th` cell and `*num2*-th` cell ****inclusive****.

The index starts from 0.

matplotlib.pyplot.Text

```
Text(x=0, y=0, text='', *, color=None, verticalalignment='baseline',
horizontalalignment='left', multialignment=None, fontproperties=None, rotation=None,
linespacing=None, rotation_mode=None, usetex=None, wrap=False,
transform_rotates_text=False, parse_math=None, antialiased=None, **kwargs)
```

Handle storing and drawing of text in window or data coordinates.

matplotlib.pyplot.TickHelper

```
TickHelper()
```

No description available.

matplotlib.pyplot.Widget

```
Widget()
```

Abstract base class for GUI neutral widgets.

matplotlib.pyplot.acorr

```
acorr(x: 'ArrayLike', *, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray,
LineCollection | Line2D, Line2D | None]'
```

Plot the autocorrelation of *x*.

Parameters

x : array-like

Not run through Matplotlib's unit conversion, so this should be a unit-less array.

detrend : callable, default: ``mlab.detrend_none`` (no detrending)

A detrending function applied to *x*. It must have the signature ::

`detrend(x: np.ndarray) -> np.ndarray`

normed : bool, default: True

If `True``, input vectors are normalised to unit length.

usevlines : bool, default: True

Determines the plot style.

If `True``, vertical lines are plotted from 0 to the acorr value using ``Axes.vlines``. Additionally, a horizontal line is plotted at `y=0` using ``Axes.axhline``.

If `False``, markers are plotted at the acorr values using ``Axes.plot``.

maxlags : int, default: 10

Number of lags to show. If `None``, will return all `2 * len(x) - 1`` lags.

Returns

lags : array (length ``2*maxlags+1``)
The lag vector.
c : array (length ``2*maxlags+1``)
The auto correlation vector.
line : ``LineCollection`` or ``Line2D``
``Artist`` added to the Axes of the correlation:

- ``LineCollection`` if `*usevlines*` is True.
- ``Line2D`` if `*usevlines*` is False.

b : ``~matplotlib.lines.Line2D`` or None
Horizontal line at 0 if `*usevlines*` is True
None `*usevlines*` is False.

Other Parameters

linestyle : ``~matplotlib.lines.Line2D`` property, optional
The linestyle for plotting the data points.
Only used if `*usevlines*` is `False``.

marker : str, default: 'o'
The marker for plotting the data points.
Only used if `*usevlines*` is `False``.

data : indexable object, optional
If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`

`**kwargs`
Additional parameters are passed to ``Axes.vlines`` and ``Axes.axhline`` if `*usevlines*` is `True``; otherwise they are passed to ``Axes.plot``.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.acorr``.

The cross correlation is performed with ``numpy.correlate`` with ```mode = "full"```.

matplotlib.pyplot.angle_spectrum

```
angle_spectrum(x: 'ArrayLike', *, Fs: 'float | None' = None, Fc: 'int | None' = None,
window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, pad_to: 'int |
None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None,
data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the angle spectrum.

Compute the angle spectrum (wrapped phase spectrum) of `*x*`.

Data is padded to a length of `*pad_to*` and the windowing function `*window*` is applied to the signal.

Parameters

`x` : 1-D array or sequence

Array or sequence containing the data.

`Fs` : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

`window` : callable or ndarray, default: ``.window_hanning``

A function or a vector of length `*NFFT*`. To create window vectors see ``.window_hanning``, ``.window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to `~numpy.fft.fft`. The default is None, which sets `*pad_to*` equal to the length of the input signal (i.e. no padding).

`Fc` : int, default: 0

The center frequency of `*x*`, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

`spectrum` : 1-D array

The values for the angle spectrum in radians (real valued).

`freqs` : 1-D array

The frequencies corresponding to the elements in `*spectrum*`.

`line` : `~matplotlib.lines.Line2D``

The line created by this function.

Other Parameters

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`

****kwargs**

Keyword arguments control the ``Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: ``~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, ``~.path.Path`` or ``~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

`markersize` or `ms`: float

`markevery`: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

`mouseover`: bool

`path_effects`: list of ``AbstractPathEffect``

`picker`: float or callable[[Artist, Event], tuple[bool, dict]]

`pickradius`: float

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`solid_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`solid_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`transform`: unknown

`url`: str

`visible`: bool

`xdata`: 1D array

`ydata`: 1D array

`zorder`: float

See Also

`magnitude_spectrum`

Plots the magnitudes of the corresponding frequencies.

`phase_spectrum`

Plots the unwrapped version of this function.

specgram

Can plot the angle spectrum of segments within the signal in a colormap.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``.axes.Axes.angle_spectrum``.

matplotlib.pyplot.annotate

```
annotate(text: 'str', xy: 'tuple[float, float]', xytext: 'tuple[float, float] | None'
         = None, xycoords: 'CoordsType' = 'data', textcoords: 'CoordsType | None' = None,
         arrowprops: 'dict[str, Any] | None' = None, annotation_clip: 'bool | None' = None,
         **kwargs) -> 'Annotation'
```

Annotate the point `*xy*` with text `*text*`.

In the simplest form, the text is placed at `*xy*`.

Optionally, the text can be displayed in another position `*xytext*`.

An arrow pointing from the text to the annotated point `*xy*` can then be added by defining `*arrowprops*`.

Parameters

text : str

The text of the annotation.

xy : (float, float)

The point `*(x, y)*` to annotate. The coordinate system is determined by `*xycoords*`.

xytext : (float, float), default: `*xy*`

The position `*(x, y)*` to place the text at. The coordinate system is determined by `*textcoords*`.

xycoords : single or two-tuple of str or ``.Artist`` or ``.Transform`` or callable, default: `'data'`

The coordinate system that `*xy*` is given in. The following types of values are supported:

- One of the following strings:

```
=====
Value Description
=====
'figure points' Points from the lower left of the figure
'figure pixels' Pixels from the lower left of the figure
'figure fraction' Fraction of figure from lower left
'subfigure points' Points from the lower left of the subfigure
'subfigure pixels' Pixels from the lower left of the subfigure
```

'subfigure fraction' Fraction of subfigure from lower left
 'axes points' Points from lower left corner of the Axes
 'axes pixels' Pixels from lower left corner of the Axes
 'axes fraction' Fraction of Axes from lower left
 'data' Use the coordinate system of the object
 being annotated (default)
 'polar' $*(\theta, r)*$ if not native 'data'
 coordinates

=====

Note that 'subfigure pixels' and 'figure pixels' are the same for the parent figure, so users who want code that is usable in a subfigure can use 'subfigure pixels'.

- An ``Artist``: `*xy*` is interpreted as a fraction of the artist's `~matplotlib.transforms.Bbox``. E.g. `*(0, 0)*` would be the lower left corner of the bounding box and `*(0.5, 1)*` would be the center top of the bounding box.

- A ``Transform`` to transform `*xy*` to screen coordinates.

- A function with one of the following signatures::

```
def transform(renderer) -> Bbox
def transform(renderer) -> Transform
```

where `*renderer*` is a ``RendererBase`` subclass.

The result of the function is interpreted like the ``Artist`` and ``Transform`` cases above.

- A tuple `*(xcoords, ycoords)*` specifying separate coordinate systems for `*x*` and `*y*`. `*xcoords*` and `*ycoords*` must each be of one of the above described types.

See :ref:`plotting-guide-annotation` for more details.

`textcoords` : single or two-tuple of str or ``Artist`` or ``Transform`` or callable, default: value of `*xycoords*`
 The coordinate system that `*xytext*` is given in.

All `*xycoords*` values are valid as well as the following strings:

=====

Value Description

=====

'offset points' Offset, in points, from the `*xy*` value
 'offset pixels' Offset, in pixels, from the `*xy*` value
 'offset fontsize' Offset, relative to fontsize, from the `*xy*` value

=====

`arrowprops` : dict, optional

The properties used to draw a ``FancyArrowPatch`` arrow between the positions `*xy*` and `*xytext*`. Defaults to None, i.e. no arrow is drawn.

For historical reasons there are two different ways to specify arrows, "simple" and "fancy":

****Simple arrow:****

If `*arrowprops*` does not contain the key 'arrowstyle' the allowed keys are:

=====	
Key Description	
=====	
width	The width of the arrow in points
headwidth	The width of the base of the arrow head in points
headlength	The length of the arrow head in points
shrink	Fraction of total length to shrink from both ends
? Any <code>`FancyArrowPatch`</code> property	
=====	

The arrow is attached to the edge of the text box, the exact position (corners or centers) depending on where it's pointing to.

****Fancy arrow:****

This is used if 'arrowstyle' is provided in the `*arrowprops*`.

Valid keys are the following ``FancyArrowPatch`` parameters:

=====	
Key Description	
=====	
arrowstyle	The arrow style
connectionstyle	The connection style
relpos	See below; default is (0.5, 0.5)
patchA	Default is bounding box of the text
patchB	Default is None
shrinkA	In points. Default is 2 points
shrinkB	In points. Default is 2 points
mutation_scale	Default is text size (in points)
mutation_aspect	Default is 1
? Any <code>`FancyArrowPatch`</code> property	
=====	

The exact starting point position of the arrow is defined by `*relpos*`. It's a tuple of relative coordinates of the text box, where (0, 0) is the lower left corner and (1, 1) is the upper right corner. Values <0 and >1 are supported and specify points outside the text box. By default (0.5, 0.5), so the starting point is centered in the text box.

`annotation_clip` : bool or None, default: None
Whether to clip (i.e. not draw) the annotation when the annotation point `*xy*` is outside the Axes area.

- If `*True*`, the annotation will be clipped when `*xy*` is outside the Axes.

- If `*False*`, the annotation will always be drawn.
- If `*None*`, the annotation will be clipped when `*xy*` is outside the Axes and `*xycoords*` is 'data'.

****kwargs**

Additional kwargs are passed to ``Text``.

Returns

``Annotation``

See Also

`:ref:`annotations``

Notes

 .. note::

This is the `:ref:`pyplot` wrapper <pyplot_interface> for `axes.Axes.annotate`.`

matplotlib.pyplot.arrow

```
arrow(x: 'float', y: 'float', dx: 'float', dy: 'float', **kwargs) -> 'FancyArrow'
```

[*Discouraged*] Add an arrow to the Axes.

This draws an arrow from ``(x, y)`` to ``(x+dx, y+dy)``.

.. admonition:: Discouraged

The use of this method is discouraged because it is not guaranteed that the arrow renders reasonably. For example, the resulting arrow is affected by the Axes aspect ratio and limits, which may distort the arrow.

Consider using ``~.Axes.annotate`` without a text instead, e.g. ::

```
ax.annotate("", xytext=(0, 0), xy=(0.5, 0.5),
arrowprops=dict(arrowstyle="->"))
```

Parameters

x, y : float

The x and y coordinates of the arrow base.

dx, dy : float

The length of the arrow along x and y direction.

width : float, default: 0.001

Width of full arrow tail.

length_includes_head : bool, default: False

True if head is to be counted in calculating the length.

head_width : float or None, default: 3*width
Total width of the full arrow head.

head_length : float or None, default: 1.5*head_width
Length of arrow head.

shape : {'full', 'left', 'right'}, default: 'full'
Draw the left-half, right-half, or full arrow.

overhang : float, default: 0
Fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

head_starts_at_zero : bool, default: False
If True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

****kwargs**
`.Patch` properties:

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: unknown

animated: bool

antialiased or aa: bool or None

capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}

clip_box: `~matplotlib.transforms.BboxBase` or None

clip_on: bool

clip_path: Patch or (Path, Transform) or None

color: `mplttype:`color`

edgecolor or ec: `mplttype:`color` or None

facecolor or fc: `mplttype:`color` or None

figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`

fill: bool

gid: str

hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}

hatch_linewidth: unknown

in_layout: bool

joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}

label: object

linestyle or ls: {'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}

linewidth or lw: float or None

mouseover: bool

path_effects: list of `.AbstractPathEffect`

picker: None or bool or float or callable

rasterized: bool

sketch_params: (scale: float, length: float, randomness: float)

snap: bool or None

transform: `~matplotlib.transforms.Transform`

url: str

visible: bool

zorder: float

Returns

``FancyArrow``

The created ``FancyArrow`` object.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.arrow``.

matplotlib.pyplot.autoscale

```
autoscale(enable: 'bool' = True, axis: "Literal['both', 'x', 'y']" = 'both', tight:
'bool | None' = None) -> 'None'
```

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling.

It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or Axes.

Parameters

`enable` : bool or None, default: True

True turns autoscaling on, False turns it off.

None leaves the autoscaling state unchanged.

`axis` : {'both', 'x', 'y'}, default: 'both'

The axis on which to operate. (For 3D Axes, `*axis*` can also be set to 'z', and 'both' refers to all three Axes.)

`tight` : bool or None, default: None

If True, first set the margins to zero. Then, this argument is forwarded to ``~.axes.Axes.autoscale_view`` (regardless of its value); see the description of its behavior there.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.autoscale``.

matplotlib.pyplot.autumn

```
autumn() -> 'None'
```

Set the colormap to 'autumn'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

```
axes(arg: 'None | tuple[float, float, float, float]' = None, **kwargs) ->
matplotlib.axes.Axes'
```

Add an Axes to the current figure and make it the current Axes.

Call signatures::

```
plt.axes()
plt.axes(rect, projection=None, polar=False, **kwargs)
plt.axes(ax)
```

Parameters

arg : None or 4-tuple

The exact behavior of this function depends on the type:

- *None*: A new full window Axes is added using `subplot(**kwargs)`.
- 4-tuple of floats *rect* = `((left, bottom, width, height))`.
A new Axes is added with dimensions *rect* in normalized (0, 1) units using `~.Figure.add_axes`` on the current figure.

projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional
The projection type of the `~.axes.Axes``. *str* is the name of a custom projection, see `~matplotlib.projections``. The default None results in a 'rectilinear' projection.

polar : bool, default: False

If True, equivalent to `projection='polar'`.

sharex, sharey : `~matplotlib.axes.Axes``, optional
Share the x or y `~matplotlib.axis`` with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

label : str

A label for the returned Axes.

Returns

`~.axes.Axes``, or a subclass of `~.axes.Axes``
The returned Axes class depends on the projection used. It is `~.axes.Axes`` if rectilinear projection is used and `~.projections.polar.PolarAxes`` if polar projection is used.

Other Parameters

**kwargs

This method also takes the keyword arguments for the returned Axes class. The keyword arguments for the rectilinear Axes class `~.axes.Axes`` can be found in the following table but there might also be other keyword arguments if another projection is used, see the actual Axes class.

Properties:

adjustable: {'box', 'datalim'}
agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
alpha: float or None
anchor: (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...}
animated: bool
aspect: {'auto', 'equal'} or float
autoscale_on: bool
autoscalex_on: unknown
autoscaley_on: unknown
axes_locator: Callable[[Axes, Renderer], Bbox]
axisbelow: bool or 'line'
box_aspect: float or None
clip_box: ~matplotlib.transforms.BboxBase or None
clip_on: bool
clip_path: Patch or (Path, Transform) or None
facecolor or fc: :mpltype:color
figure: ~matplotlib.figure.Figure or ~matplotlib.figure.SubFigure
forward_navigation_events: bool or "auto"
frame_on: bool
gid: str
in_layout: bool
label: object
mouseover: bool
navigate: bool
navigate_mode: unknown
path_effects: list of ~.AbstractPathEffect
picker: None or bool or float or callable
position: [left, bottom, width, height] or ~matplotlib.transforms.Bbox
prop_cycle: ~cycler.Cycler
rasterization_zorder: float or None
rasterized: bool
sketch_params: (scale: float, length: float, randomness: float)
snap: bool or None
subplotspec: unknown
title: str
transform: ~matplotlib.transforms.Transform
url: str
visible: bool
xbound: (lower: float, upper: float)
xlabel: str
xlim: (left: float, right: float)
xmargin: float greater than -0.5
xscale: unknown
xticklabels: unknown
xticks: unknown
ybound: (lower: float, upper: float)
ylabel: str
ylim: (bottom: float, top: float)
ymargin: float greater than -0.5
yscale: unknown
yticklabels: unknown
yticks: unknown

zorder: float

See Also

.Figure.add_axes
.pyplot.subplot
.Figure.add_subplot
.Figure.subplots
.pyplot.subplots

Examples

::

```
# Creating a new full window Axes
```

```
plt.axes()
```

```
# Creating a new Axes with specified dimensions and a grey background
```

```
plt.axes((left, bottom, width, height), facecolor='grey')
```

matplotlib.pyplot.axhline

```
axhline(y: 'float' = 0, xmin: 'float' = 0, xmax: 'float' = 1, **kwargs) -> 'Line2D'
```

Add a horizontal line spanning the whole or fraction of the Axes.

Note: If you want to set x-limits in data coordinates, use

`~.Axes.hlines` instead.`

Parameters

y : float, default: 0

y position in :ref:`data coordinates <coordinate-systems>`.

xmin : float, default: 0

The start x-position in :ref:`axes coordinates <coordinate-systems>`.

Should be between 0 and 1, 0 being the far left of the plot,

1 the far right of the plot.

xmax : float, default: 1

The end x-position in :ref:`axes coordinates <coordinate-systems>`.

Should be between 0 and 1, 0 being the far left of the plot,

1 the far right of the plot.

Returns

`~matplotlib.lines.Line2D``

A `~matplotlib.lines.Line2D`` specified via two points `((xmin, y))` , ((xmax, y))`.`

Its transform is set such that `*x*` is in

:ref:`axes coordinates <coordinate-systems>` and `*y*` is in

:ref:`data coordinates <coordinate-systems>`.

This is still a generic line and the horizontal character is only realized through using identical `*y*` values for both points. Thus,

if you want to change the `*y*` value later, you have to provide two values `line.set_ydata([3, 3])`.

Other Parameters

****kwargs**

Valid keyword arguments are `.Line2D`` properties, except for `'transform'`:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: `.CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: `.JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

`markersize` or `ms`: float

`markevery`: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

`mouseover`: bool

`path_effects`: list of `.AbstractPathEffect``

`picker`: float or callable[[Artist, Event], tuple[bool, dict]]

`pickradius`: float

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`solid_capstyle`: `.CapStyle`` or {'butt', 'projecting', 'round'}

`solid_joinstyle`: `.JoinStyle`` or {'miter', 'round', 'bevel'}

`transform`: unknown

`url`: str

`visible`: bool

`xdata`: 1D array

`ydata`: 1D array

`zorder`: float

See Also

`hlines` : Add horizontal lines in data coordinates.

`axhspan` : Add a horizontal span (rectangle) across the axis.

`axline` : Add a line with an arbitrary slope.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.axhline`.

Examples

* draw a thick red hline at `'y' = 0` that spans the xrange::

```
>>> axhline(linewidth=4, color='r')
```

* draw a default hline at `'y' = 1` that spans the xrange::

```
>>> axhline(y=1)
```

* draw a default hline at `'y' = .5` that spans the middle half of the xrange::

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

matplotlib.pyplot.axhspan

```
axhspan(ymin: 'float', ymax: 'float', xmin: 'float' = 0, xmax: 'float' = 1, **kwargs)
-> 'Rectangle'
```

Add a horizontal span (rectangle) across the Axes.

The rectangle spans from `*ymin*` to `*ymax*` vertically, and, by default, the whole x-axis horizontally. The x-span can be set using `*xmin*` (default: 0) and `*xmax*` (default: 1) which are in axis units; e.g.

```xmin = 0.5``` always refers to the middle of the x-axis regardless of the limits set by `~.Axes.set_xlim`.

Parameters

-----

`ymin` : float

Lower y-coordinate of the span, in data units.

`ymax` : float

Upper y-coordinate of the span, in data units.

`xmin` : float, default: 0

Lower x-coordinate of the span, in x-axis (0-1) units.

`xmax` : float, default: 1

Upper x-coordinate of the span, in x-axis (0-1) units.

Returns

-----

`~matplotlib.patches.Rectangle`

Horizontal span (rectangle) from (xmin, ymin) to (xmax, ymax).

#### Other Parameters

-----  
\*\*kwargs : `~matplotlib.patches.Rectangle` properties

#### Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None

angle: unknown

animated: bool

antialiased or aa: bool or None

bounds: (left, bottom, width, height)

capstyle: `~matplotlib.patches.Path` or {'butt', 'projecting', 'round'}

clip\_box: `~matplotlib.transforms.BboxBase` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

color: :mpltype:`color`

edgecolor or ec: :mpltype:`color` or None

facecolor or fc: :mpltype:`color` or None

figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`

fill: bool

gid: str

hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '\*'}

hatch\_linewidth: unknown

height: unknown

in\_layout: bool

joinstyle: `~matplotlib.patches.Path` or {'miter', 'round', 'bevel'}

label: object

linestyle or ls: {'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}

linewidth or lw: float or None

mouseover: bool

path\_effects: list of `~matplotlib.patches.Path`

picker: None or bool or float or callable

rasterized: bool

sketch\_params: (scale: float, length: float, randomness: float)

snap: bool or None

transform: `~matplotlib.transforms.Transform`

url: str

visible: bool

width: unknown

x: unknown

xy: (float, float)

y: unknown

zorder: float

#### See Also

-----  
axvspan : Add a vertical span across the Axes.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.axhspan``.

## matplotlib.pyplot.axis

```
axis(arg: 'tuple[float, float, float, float] | bool | str | None' = None, /, *, emit:
'bool' = True, **kwargs) -> 'tuple[float, float, float, float]'
```

Convenience method to get or set some axis properties.

Call signatures::

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

Parameters

-----

xmin, xmax, ymin, ymax : float, optional

The axis limits to be set. This can also be achieved using ::

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

option : bool or str

If a bool, turns axis lines and labels on or off. If a string, possible values are:

=====

Value	Description
-------	-------------

=====

'off' or 'False' Hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines.

This is the same as `~.Axes.set_axis_off()`.

'on' or 'True' Do not hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines.

This is the same as `~.Axes.set_axis_on()`.

'equal' Set equal scaling (i.e., make circles circular) by changing the axis limits. This is the same as

```
``ax.set_aspect('equal', adjustable='datalim')``
```

Explicit data limits may not be respected in this case.

'scaled' Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as

```
``ax.set_aspect('equal', adjustable='box', anchor='C')``
```

Additionally, further autoscaling will be disabled.

'tight' Set limits just large enough to show all data, then disable further autoscaling.

'auto' Automatic scaling (fill plot box with data).

'image' 'scaled' with axis limits equal to data limits.

'square' Square plot; similar to 'scaled', but initially forcing ```xmax-xmin == ymax-ymin```.

=====

emit : bool, default: True

Whether observers are notified of the axis limit change.

This option is passed on to `~.Axes.set_xlim`` and `~.Axes.set_ylim``.

#### Returns

-----  
xmin, xmax, ymin, ymax : float  
The axis limits.

#### See Also

-----  
`matplotlib.axes.Axes.set_xlim`  
`matplotlib.axes.Axes.set_ylim`

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.axis``.

For 3D Axes, this method additionally takes `*zmin*`, `*zmax*` as parameters and likewise returns them.

## matplotlib.pyplot.axline

```
axline(xy1: 'tuple[float, float]', xy2: 'tuple[float, float] | None' = None, *, slope: 'float | None' = None, **kwargs) -> 'AxLine'
```

Add an infinitely long straight line.

The line can be defined either by two points `*xy1*` and `*xy2*`, or by one point `*xy1*` and a `*slope*`.

This draws a straight line "on the screen", regardless of the x and y scales, and is thus also suitable for drawing exponential decays in semilog plots, power laws in loglog plots, etc. However, `*slope*` should only be used with linear scales; It has no clear meaning for all other scales, and thus the behavior is undefined. Please specify the line using the points `*xy1*`, `*xy2*` for non-linear scales.

The `*transform*` keyword argument only applies to the points `*xy1*`, `*xy2*`. The `*slope*` (if given) is always in data coordinates. This can be used e.g. with ```ax.transAxes``` for drawing grid lines with a fixed slope.

#### Parameters

-----  
`xy1, xy2` : (float, float)  
Points for the line to pass through.  
Either `*xy2*` or `*slope*` has to be given.  
`slope` : float, optional  
The slope of the line. Either `*xy2*` or `*slope*` has to be given.

#### Returns

-----

``AxLine``

## Other Parameters

-----

**\*\*kwargs**

Valid kwargs are ``Line2D`` properties

### Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

`markersize` or `ms`: float

`markevery`: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

`mouseover`: bool

`path_effects`: list of ``AbstractPathEffect``

`picker`: float or callable[[Artist, Event], tuple[bool, dict]]

`pickradius`: float

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`solid_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`solid_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`transform`: unknown

`url`: str

`visible`: bool

`xdata`: 1D array

`ydata`: 1D array

`zorder`: float

### See Also

-----

axhline : for horizontal lines  
axvline : for vertical lines

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.axline``.

#### Examples

-----

Draw a thick red line passing through (0, 0) and (1, 1)::

```
>>> axline((0, 0), (1, 1), linewidth=4, color='r')
```

## matplotlib.pyplot.axvline

```
axvline(x: 'float' = 0, ymin: 'float' = 0, ymax: 'float' = 1, **kwargs) -> 'Line2D'
```

Add a vertical line spanning the whole or fraction of the Axes.

Note: If you want to set y-limits in data coordinates, use  
`~.Axes.vlines`` instead.

#### Parameters

-----

x : float, default: 0

x position in :ref:`data coordinates <coordinate-systems>`.

ymin : float, default: 0

The start y-position in :ref:`axes coordinates <coordinate-systems>`.  
Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

ymax : float, default: 1

The end y-position in :ref:`axes coordinates <coordinate-systems>`.  
Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

#### Returns

-----

`~matplotlib.lines.Line2D``

A `.Line2D`` specified via two points ``(x, ymin)``, ``(x, ymax)``.

Its transform is set such that `*x*` is in

:ref:`data coordinates <coordinate-systems>` and `*y*` is in

:ref:`axes coordinates <coordinate-systems>`.

This is still a generic line and the vertical character is only realized through using identical `*x*` values for both points. Thus, if you want to change the `*x*` value later, you have to provide two values ```line.set_xdata([3, 3])```.

#### Other Parameters

-----  
\*\*kwargs

Valid keyword arguments are ``Line2D`` properties, except for `'transform'`:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: ``~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, ``~.path.Path`` or ``~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

`markersize` or `ms`: float

`markevery`: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

`mouseover`: bool

`path_effects`: list of ``AbstractPathEffect``

`picker`: float or callable[[Artist, Event], tuple[bool, dict]]

`pickradius`: float

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`solid_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`solid_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`transform`: unknown

`url`: str

`visible`: bool

`xdata`: 1D array

`ydata`: 1D array

`zorder`: float

See Also

-----  
`vlines` : Add vertical lines in data coordinates.

`axvspan` : Add a vertical span (rectangle) across the axis.

axline : Add a line with an arbitrary slope.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.axvline``.

#### Examples

-----

\* draw a thick red vline at `*x* = 0` that spans the yrange::

```
>>> axvline(linewidth=4, color='r')
```

\* draw a default vline at `*x* = 1` that spans the yrange::

```
>>> axvline(x=1)
```

\* draw a default vline at `*x* = .5` that spans the middle half of the yrange::

```
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

## matplotlib.pyplot.axvspan

```
axvspan(xmin: 'float', xmax: 'float', ymin: 'float' = 0, ymax: 'float' = 1, **kwargs)
-> 'Rectangle'
```

Add a vertical span (rectangle) across the Axes.

The rectangle spans from `*xmin*` to `*xmax*` horizontally, and, by default, the whole y-axis vertically. The y-span can be set using `*ymin*` (default: 0) and `*ymax*` (default: 1) which are in axis units; e.g. ```ymin = 0.5``` always refers to the middle of the y-axis regardless of the limits set by `~.Axes.set_ylim``.

#### Parameters

-----

`xmin` : float

Lower x-coordinate of the span, in data units.

`xmax` : float

Upper x-coordinate of the span, in data units.

`ymin` : float, default: 0

Lower y-coordinate of the span, in y-axis units (0-1).

`ymax` : float, default: 1

Upper y-coordinate of the span, in y-axis units (0-1).

#### Returns

-----

`~matplotlib.patches.Rectangle``

Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

#### Other Parameters

-----

**\*\*kwargs** : `~matplotlib.patches.Rectangle`` properties

Properties:

**agg\_filter**: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

**alpha**: float or None

**angle**: unknown

**animated**: bool

**antialiased** or **aa**: bool or None

**bounds**: (left, bottom, width, height)

**capstyle**: ``.CapStyle`` or `{'butt', 'projecting', 'round'}`

**clip\_box**: `~matplotlib.transforms.BboxBase`` or None

**clip\_on**: bool

**clip\_path**: Patch or (Path, Transform) or None

**color**: `:mpltype:`color``

**edgecolor** or **ec**: `:mpltype:`color`` or None

**facecolor** or **fc**: `:mpltype:`color`` or None

**figure**: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

**fill**: bool

**gid**: str

**hatch**: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`

**hatch\_linewidth**: unknown

**height**: unknown

**in\_layout**: bool

**joinstyle**: ``.JoinStyle`` or `{'miter', 'round', 'bevel'}`

**label**: object

**linestyle** or **ls**: `{'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}`

**linewidth** or **lw**: float or None

**mouseover**: bool

**path\_effects**: list of ``.AbstractPathEffect``

**picker**: None or bool or float or callable

**rasterized**: bool

**sketch\_params**: (scale: float, length: float, randomness: float)

**snap**: bool or None

**transform**: `~matplotlib.transforms.Transform``

**url**: str

**visible**: bool

**width**: unknown

**x**: unknown

**xy**: (float, float)

**y**: unknown

**zorder**: float

See Also

-----

**axhspan** : Add a horizontal span across the Axes.

Notes

-----

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for ``.axes.Axes.axvspan``.

Examples

-----  
Draw a vertical, green, translucent rectangle from  $x = 1.25$  to  $x = 1.55$  that spans the yrange of the Axes.

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

## matplotlib.pyplot.bar

```
bar(x: 'float | ArrayLike', height: 'float | ArrayLike', width: 'float | ArrayLike' = 0.8, bottom: 'float | ArrayLike | None' = None, *, align: "Literal['center', 'edge']" = 'center', data=None, **kwargs) -> 'BarContainer'
```

Make a bar plot.

The bars are positioned at *x* with the given *align*ment. Their dimensions are given by *height* and *width*. The vertical baseline is *bottom* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

### Parameters

-----  
*x* : float or array-like

The x coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

Bars are often used for categorical data, i.e. string labels below the bars. You can provide a list of strings directly to *x*.  
``bar(['A', 'B', 'C'], [1, 2, 3])`` is often a shorter and more convenient notation compared to  
``bar(range(3), [1, 2, 3], tick\_label=['A', 'B', 'C'])``. They are equivalent as long as the names are unique. The explicit *tick\_label* notation draws the names in the sequence given. However, when having duplicate values in categorical *x* data, these values map to the same numerical x coordinate, and hence the corresponding bars are drawn on top of each other.

*height* : float or array-like

The height(s) of the bars.

Note that if *bottom* has units (e.g. datetime), *height* should be in units that are a difference from the value of *bottom* (e.g. timedelta).

*width* : float or array-like, default: 0.8

The width(s) of the bars.

Note that if *x* has units (e.g. datetime), then *width* should be in units that are a difference (e.g. timedelta) around the *x* values.

*bottom* : float or array-like, default: 0

The y coordinate(s) of the bottom side(s) of the bars.

Note that if *bottom* has units, then the y-axis will get a Locator and Formatter appropriate for the units (e.g. dates, or categorical).

align : {'center', 'edge'}, default: 'center'

Alignment of the bars to the *\*x\** coordinates:

- 'center': Center the base on the *\*x\** positions.
- 'edge': Align the left edges of the bars with the *\*x\** positions.

To align the bars on the right edge pass a negative *\*width\** and ``align='edge'``.

Returns

-----

`.BarContainer``

Container with all the bars and optionally errorbars.

Other Parameters

-----

color : :mpltype:`color` or list of :mpltype:`color`, optional

The colors of the bar faces. This is an alias for *\*facecolor\**.

If both are given, *\*facecolor\** takes precedence.

facecolor : :mpltype:`color` or list of :mpltype:`color`, optional

The colors of the bar faces.

If both *\*color\** and *\*facecolor\** are given, *\*facecolor\** takes precedence.

edgecolor : :mpltype:`color` or list of :mpltype:`color`, optional

The colors of the bar edges.

linewidth : float or array-like, optional

Width of the bar edge(s). If 0, don't draw edges.

tick\_label : str or list of str, optional

The tick labels of the bars.

Default: None (Use default numeric labels.)

label : str or list of str, optional

A single label is attached to the resulting `.BarContainer`` as a label for the whole dataset.

If a list is provided, it must be the same length as *\*x\** and labels the individual bars. Repeated labels are not de-duplicated and will cause repeated label entries, so this is best used when bars also differ in style (e.g., by passing a list to *\*color\**.)

xerr, yerr : float or array-like of shape(N,) or shape(2, N), optional

If not *\*None\**, add horizontal / vertical errorbars to the bar tips.

The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *\*None\**: No errorbar. (Default)

See :doc:`/gallery/statistics/errorbar\_features` for an example on

the usage of `*xerr*` and `*yerr*`.

`ecolor` : `:mpltype:`color`` or list of `:mpltype:`color``, default: `'black'`  
The line color of the errorbars.

`capsize` : float, default: `:rc:`errorbar.capsize``  
The length of the error bar caps in points.

`error_kw` : dict, optional  
Dictionary of keyword arguments to be passed to the  
`~.Axes.errorbar`` method. Values of `*ecolor*` or `*capsize*` defined  
here take precedence over the independent keyword arguments.

`log` : bool, default: `False`  
If `*True*`, set the y-axis to be log scale.

`data` : indexable object, optional  
If given, all parameters also accept a string ```s```, which is  
interpreted as ```data[s]``` if ```s``` is a key in ```data```.

`**kwargs` : `~.Rectangle`` properties

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array  
and two offsets from the bottom left corner of the image

`alpha`: float or `None`

`angle`: unknown

`animated`: bool

`antialiased` or `aa`: bool or `None`

`bounds`: (left, bottom, width, height)

`capstyle`: `~.CapStyle`` or `{'butt', 'projecting', 'round'}`

`clip_box`: `~matplotlib.transforms.BboxBase`` or `None`

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or `None`

`color`: `:mpltype:`color``

`edgecolor` or `ec`: `:mpltype:`color`` or `None`

`facecolor` or `fc`: `:mpltype:`color`` or `None`

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fill`: bool

`gid`: str

`hatch`: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`

`hatch_linewidth`: unknown

`height`: unknown

`in_layout`: bool

`joinstyle`: `~.JoinStyle`` or `{'miter', 'round', 'bevel'}`

`label`: object

`linestyle` or `ls`: `{'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}`

`linewidth` or `lw`: float or `None`

`mouseover`: bool

`path_effects`: list of `~.AbstractPathEffect``

`picker`: `None` or bool or float or callable

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or `None`

`transform`: `~matplotlib.transforms.Transform``

url: str  
visible: bool  
width: unknown  
x: unknown  
xy: (float, float)  
y: unknown  
zorder: float

See Also

-----  
barh : Plot a horizontal bar plot.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.bar``.

Stacked bars can be achieved by passing individual *\*bottom\** values per bar. See :doc:`gallery/lines\_bars\_and\_markers/bar\_stacked`.

## matplotlib.pyplot.bar\_label

```
bar_label(container: 'BarContainer', labels: 'ArrayLike | None' = None, *, fmt: 'str | Callable[[float], str]' = '%g', label_type: "Literal['center', 'edge']" = 'edge', padding: 'float' = 0, **kwargs) -> 'list[Annotation]'
```

Label a bar plot.

Adds labels to bars in the given `.BarContainer``.  
You may need to adjust the axis limits to fit the labels.

Parameters

-----

container : `.BarContainer``  
Container with all the bars and optionally errorbars, likely  
returned from `.bar`` or `.barh``.

labels : array-like, optional  
A list of label texts, that should be displayed. If not given, the  
label texts will be the data values formatted with *\*fmt\**.

fmt : str or callable, default: '%g'  
An unnamed %-style or {}-style format string for the label or a  
function to call with the value as the first argument.  
When *\*fmt\** is a string and can be interpreted in both formats,  
%-style takes precedence over {}-style.

.. versionadded:: 3.7  
Support for {}-style format string and callables.

label\_type : {'edge', 'center'}, default: 'edge'  
The label type. Possible values:

- 'edge': label placed at the end-point of the bar segment, and the value displayed will be the position of that end-point.
- 'center': label placed in the center of the bar segment, and the value displayed will be the length of that segment.

(useful for stacked bars, i.e.,  
:doc:`gallery/lines\_bars\_and\_markers/bar\_label\_demo`)

padding : float, default: 0  
Distance of label from the end of the bar, in points.

**\*\*kwargs**  
Any remaining keyword arguments are passed through to ``Axes.annotate``. The alignment parameters (`*horizontalalignment* / *ha*`, `*verticalalignment* / *va*`) are not supported because the labels are automatically aligned to the bars.

**Returns**  
-----  
list of ``Annotation``  
A list of ``Annotation`` instances for the labels.

**Notes**  
-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``axes.Axes.bar_label``.

## matplotlib.pyplot.barbs

```
barbs(*args, data=None, **kwargs) -> 'Barbs'
```

Plot a 2D field of wind barbs.

Call signature::

`barbs([X, Y], U, V, [C], /, **kwargs)`

Where `*X*`, `*Y*` define the barb locations, `*U*`, `*V*` define the barb directions, and `*C*` optionally sets the color.

The arguments `*X*`, `*Y*`, `*U*`, `*V*`, `*C*` are positional-only and may be 1D or 2D. `*U*`, `*V*`, `*C*` may be masked arrays, but masked `*X*`, `*Y*` are not supported at present.

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below::

```

: /\
: /\ \
: /\ \ \
: /\ \ \
: -----

```

The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

See also [https://en.wikipedia.org/wiki/Wind\\_barb](https://en.wikipedia.org/wiki/Wind_barb).

#### Parameters

-----

X, Y : 1D or 2D array-like, optional

The x and y coordinates of the barb locations. See *\*pivot\** for how the barbs are drawn to the x, y positions.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *\*U\** and *\*V\**.

If *\*X\** and *\*Y\** are 1D but *\*U\**, *\*V\** are 2D, *\*X\**, *\*Y\** are expanded to 2D using ``X, Y = np.meshgrid(X, Y)``. In this case ``len(X)`` and ``len(Y)`` must match the column and row dimensions of *\*U\** and *\*V\**.

U, V : 1D or 2D array-like

The x and y components of the barb shaft.

C : 1D or 2D array-like, optional

Numeric data that defines the barb colors by colormapping via *\*norm\** and *\*cmap\**.

This does not support explicit colors. If you want to set colors directly, use *\*barbcolor\** instead.

length : float, default: 7

Length of the barb in points; the other parts of the barb are scaled against this.

pivot : {'tip', 'middle'} or float, default: 'tip'

The part of the arrow that is anchored to the *\*X\**, *\*Y\** grid. The barb rotates about this point. This can also be a number, which shifts the start of the barb that many points away from grid point.

barbcolor : :mpltype:`color` or color sequence

The color of all parts of the barb except for the flags. This parameter is analogous to the *\*edgecolor\** parameter for polygons, which can be used instead. However this parameter will override *facecolor*.

flagcolor : :mpltype:`color` or color sequence

The color of any flags on the barb. This parameter is analogous to the `*facecolor*` parameter for polygons, which can be used instead. However, this parameter will override `facecolor`. If this is not set (and `*C*` has not either) then `*flagcolor*` will be set to match `*barbcolor*` so that the barb has a uniform color. If `*C*` has been set, `*flagcolor*` has no effect.

`sizes` : dict, optional

A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

`fill_empty` : bool, default: False

Whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, the center is transparent.

`rounding` : bool, default: True

Whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple.

`barb_increments` : dict, optional

A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

`flip_barb` : bool or array-like of bool, default: False

Whether the lines and flags should point opposite to normal.

Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere).

A single value is applied to all barbs. Individual barbs can be flipped by passing a bool array of the same size as `*U*` and `*V*`.

Returns

-----  
barbs : `~matplotlib.quiver.Barbs``

Other Parameters

-----  
`data` : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

**\*\*kwargs**

The barbs can further be customized using ``PolyCollection`` keyword arguments:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image  
`alpha`: array-like or float or None  
`animated`: bool  
`antialiased` or `aa` or `antialiaseds`: bool or list of bools  
`array`: array-like or None  
`capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}  
`clim`: (vmin: float, vmax: float)  
`clip_box`: ``~matplotlib.transforms.BboxBase`` or None  
`clip_on`: bool  
`clip_path`: Patch or (Path, Transform) or None  
`cmap`: ``Colormap`` or str or None  
`color`: `:mpltype:`color`` or list of RGBA tuples  
`edgecolor` or `ec` or `edgecolors`: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'  
`facecolor` or `facecolors` or `fc`: `:mpltype:`color`` or list of `:mpltype:`color``  
`figure`: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``  
`gid`: str  
`hatch`: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '\*'}  
`hatch_linewidth`: unknown  
`in_layout`: bool  
`joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}  
`label`: object  
`linestyle` or `dashes` or `linestyles` or `ls`: str or tuple or list thereof  
`linewidth` or `linewidths` or `lw`: float or list of floats  
`mouseover`: bool  
`norm`: ``Normalize`` or str or None  
`offset_transform` or `transOffset`: ``Transform``  
`offsets`: (N, 2) or (2,) array-like  
`path_effects`: list of ``AbstractPathEffect``  
`paths`: list of array-like  
`picker`: None or bool or float or callable  
`pickradius`: float  
`rasterized`: bool  
`sizes`: ``numpy.ndarray`` or None  
`sketch_params`: (scale: float, length: float, randomness: float)  
`snap`: bool or None  
`transform`: ``~matplotlib.transforms.Transform``  
`url`: str  
`urls`: list of str or None  
`verts`: list of array-like  
`verts_and_codes`: unknown  
`visible`: bool  
`zorder`: float

Notes

-----

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``axes.Axes.barbs``.

## matplotlib.pyplot.barh

```
barh(y: 'float | ArrayLike', width: 'float | ArrayLike', height: 'float | ArrayLike' = 0.8, left: 'float | ArrayLike | None' = None, *, align: "Literal['center', 'edge']" = 'center', data=None, **kwargs) -> 'BarContainer'
```

Make a horizontal bar plot.

The bars are positioned at *y* with the given *align*ment. Their dimensions are given by *width* and *height*. The horizontal baseline is *left* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

### Parameters

*y* : float or array-like

The y coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

Bars are often used for categorical data, i.e. string labels below the bars. You can provide a list of strings directly to *y*.  
``barh(['A', 'B', 'C'], [1, 2, 3])`` is often a shorter and more convenient notation compared to  
``barh(range(3), [1, 2, 3], tick\_label=['A', 'B', 'C'])``. They are equivalent as long as the names are unique. The explicit *tick\_label* notation draws the names in the sequence given. However, when having duplicate values in categorical *y* data, these values map to the same numerical y coordinate, and hence the corresponding bars are drawn on top of each other.

*width* : float or array-like

The width(s) of the bars.

Note that if *left* has units (e.g. datetime), *width* should be in units that are a difference from the value of *left* (e.g. timedelta).

*height* : float or array-like, default: 0.8

The heights of the bars.

Note that if *y* has units (e.g. datetime), then *height* should be in units that are a difference (e.g. timedelta) around the *y* values.

*left* : float or array-like, default: 0

The x coordinates of the left side(s) of the bars.

Note that if *left* has units, then the x-axis will get a Locator and Formatter appropriate for the units (e.g. dates, or categorical).

*align* : {'center', 'edge'}, default: 'center'

Alignment of the base to the *y* coordinates:

- 'center': Center the bars on the *y* positions.
- 'edge': Align the bottom edges of the bars with the *y*

positions.

To align the bars on the top edge pass a negative *\*height\** and `align='edge'`.

#### Returns

-----  
`.BarContainer``  
Container with all the bars and optionally errorbars.

#### Other Parameters

-----  
`color` : `:mpltype:`color`` or list of `:mpltype:`color``, optional  
The colors of the bar faces.

`edgecolor` : `:mpltype:`color`` or list of `:mpltype:`color``, optional  
The colors of the bar edges.

`linewidth` : float or array-like, optional  
Width of the bar edge(s). If 0, don't draw edges.

`tick_label` : str or list of str, optional  
The tick labels of the bars.  
Default: None (Use default numeric labels.)

`label` : str or list of str, optional  
A single label is attached to the resulting `.BarContainer`` as a label for the whole dataset.  
If a list is provided, it must be the same length as *\*y\** and labels the individual bars. Repeated labels are not de-duplicated and will cause repeated label entries, so this is best used when bars also differ in style (e.g., by passing a list to *\*color\**.)

`xerr`, `yerr` : float or array-like of shape(N,) or shape(2, N), optional  
If not *\*None\**, add horizontal / vertical errorbars to the bar tips.  
The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *\*None\**: No errorbar. (default)

See `:doc:`/gallery/statistics/errorbar_features`` for an example on the usage of *\*xerr\** and *\*yerr\**.

`ecolor` : `:mpltype:`color`` or list of `:mpltype:`color``, default: 'black'  
The line color of the errorbars.

`capsize` : float, default: `:rc:`errorbar.capsize``  
The length of the error bar caps in points.

`error_kw` : dict, optional  
Dictionary of keyword arguments to be passed to the

`~.Axes.errorbar` method. Values of *ecolor* or *capsize* defined here take precedence over the independent keyword arguments.`

`log` : bool, default: False

If `True`, set the x-axis to be log scale.

`data` : indexable object, optional

If given, all parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`.

`**kwargs` : `~.Rectangle` properties`

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`angle`: unknown

`animated`: bool

`antialiased` or `aa`: bool or None

`bounds`: (left, bottom, width, height)

`capstyle`: `~.CapStyle` or {'butt', 'projecting', 'round'}`

`clip_box`: `~matplotlib.transforms.BboxBase` or None`

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color`: `:mpltype:color``

`edgecolor` or `ec`: `:mpltype:color` or None`

`facecolor` or `fc`: `:mpltype:color` or None`

`figure`: `~matplotlib.figure.Figure` or ~matplotlib.figure.SubFigure``

`fill`: bool

`gid`: str

`hatch`: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '\*'}

`hatch_linewidth`: unknown

`height`: unknown

`in_layout`: bool

`joinstyle`: `~.JoinStyle` or {'miter', 'round', 'bevel'}`

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}

`linewidth` or `lw`: float or None

`mouseover`: bool

`path_effects`: list of `~.AbstractPathEffect``

`picker`: None or bool or float or callable

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`transform`: `~matplotlib.transforms.Transform``

`url`: str

`visible`: bool

`width`: unknown

`x`: unknown

`xy`: (float, float)

`y`: unknown

`zorder`: float

See Also

-----

bar : Plot a vertical bar plot.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.barh``.

Stacked bars can be achieved by passing individual `*left*` values per bar. See

:doc:`gallery/lines\_bars\_and\_markers/horizontal\_barchart\_distribution`.

## matplotlib.pyplot.bone

```
bone() -> 'None'
```

Set the colormap to 'bone'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.box

```
box(on: 'bool | None' = None) -> 'None'
```

Turn the Axes box on or off on the current Axes.

Parameters

-----

on : bool or None

The new `~matplotlib.axes.Axes`` box state. If ```None```, toggle the state.

See Also

-----

:meth:`matplotlib.axes.Axes.set\_frame\_on`

:meth:`matplotlib.axes.Axes.get\_frame\_on`

## matplotlib.pyplot.boxplot

```
boxplot(x: 'ArrayLike | Sequence[ArrayLike]', *, notch: 'bool | None' = None, sym: 'str | None' = None, vert: 'bool | None' = None, orientation: "Literal['vertical', 'horizontal']" = 'vertical', whis: 'float | tuple[float, float] | None' = None, positions: 'ArrayLike | None' = None, widths: 'float | ArrayLike | None' = None, patch_artist: 'bool | None' = None, bootstrap: 'int | None' = None, usermedians: 'ArrayLike | None' = None, conf_intervals: 'ArrayLike | None' = None, meanline: 'bool | None' = None, showmeans: 'bool | None' = None, showcaps: 'bool | None' = None, showbox: 'bool | None' = None, showfliers: 'bool | None' = None, boxprops: 'dict[str, Any] | None' = None, tick_labels: 'Sequence[str] | None' = None, flierprops: 'dict[str, Any] | None' = None, medianprops: 'dict[str, Any] | None' = None, meanprops: 'dict[str, Any] | None' = None, capprops: 'dict[str, Any] | None' = None, whiskerprops: 'dict[str, Any] | None' = None, manage_ticks: 'bool' = True, autorange: 'bool' = False, zorder: 'float | None' = None, capwidths: 'float | ArrayLike | None' = None, label: 'Sequence[str] | None' = None, data=None) -> 'dict[str, Any]'
```

Draw a box and whisker plot.

The box extends from the first quartile (Q1) to the third quartile (Q3) of the data, with a line at the median. The whiskers extend from the box to the farthest data point lying within 1.5x the inter-quartile range (IQR) from the box. Flier points are those past the end of the whiskers. See [https://en.wikipedia.org/wiki/Box\\_plot](https://en.wikipedia.org/wiki/Box_plot) for reference.

.. code-block:: none

```
Q1-1.5IQR Q1 median Q3 Q3+1.5IQR
|----:----|
o |-----| : |-----| o o
|----:----|
flier <-----> fliers
IQR
```

#### Parameters

-----

x : Array or a sequence of vectors.

The input data. If a 2D array, a boxplot is drawn for each column in \*x\*. If a sequence of 1D arrays, a boxplot is drawn for each array in \*x\*.

notch : bool, default: :rc:`boxplot.notch`

Whether to draw a notched boxplot (`True`), or a rectangular boxplot (`False`). The notches represent the confidence interval (CI) around the median. The documentation for *\*bootstrap\** describes how the locations of the notches are computed by default, but their locations may also be overridden by setting the *\*conf\_intervals\** parameter.

.. note::

In cases where the values of the CI are less than the lower quartile or greater than the upper quartile, the notches will extend beyond the box, giving it a distinctive "flipped" appearance. This is expected behavior and consistent with other statistical visualization packages.

sym : str, optional

The default symbol for flier points. An empty string (") hides the fliers. If `None`, then the fliers default to 'b+'. More control is provided by the *\*flierprops\** parameter.

vert : bool, optional

.. deprecated:: 3.11

Use *\*orientation\** instead.

This is a pending deprecation for 3.10, with full deprecation in 3.11 and removal in 3.13.

If this is given during the deprecation period, it overrides the *\*orientation\** parameter.

If True, plots the boxes vertically.  
If False, plots the boxes horizontally.

orientation : {'vertical', 'horizontal'}, default: 'vertical'  
If 'horizontal', plots the boxes horizontally.  
Otherwise, plots the boxes vertically.

.. versionadded:: 3.10

whis : float or (float, float), default: 1.5  
The position of the whiskers.

If a float, the lower whisker is at the lowest datum above  $Q1 - whis(Q3 - Q1)$ , and the upper whisker at the highest datum below  $Q3 + whis(Q3 - Q1)$ , where  $Q1$  and  $Q3$  are the first and third quartiles. The default value of  $whis = 1.5$  corresponds to Tukey's original definition of boxplots.

If a pair of floats, they indicate the percentiles at which to draw the whiskers (e.g., (5, 95)). In particular, setting this to (0, 100) results in whiskers covering the whole range of the data.

In the edge case where  $Q1 == Q3$ , *whis* is automatically set to (0, 100) (cover the whole range of the data) if *autorange* is True.

Beyond the whiskers, data are considered outliers and are plotted as individual points.

bootstrap : int, optional  
Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If *bootstrap* is None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.

usermedians : 1D array-like, optional  
A 1D array-like of length  $\text{len}(x)$ . Each entry that is not `None` forces the value of the median for the corresponding dataset. For entries that are `None`, the medians are computed by Matplotlib as normal.

conf\_intervals : array-like, optional  
A 2D array-like of shape  $(\text{len}(x), 2)$ . Each entry that is not None forces the location of the corresponding notch (which is only drawn if *notch* is `True`). For entries that are `None`, the notches are computed by the method specified by the other parameters (e.g., *bootstrap*).

positions : array-like, optional

The positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to ``range(1, N+1)`` where N is the number of boxes to be drawn.

`widths` : float or array-like

The widths of the boxes. The default is 0.5, or ``0.15\*(distance between extreme positions)``, if that is smaller.

`patch_artist` : bool, default: :rc:`boxplot.patchartist`

If ``False`` produces boxes with the Line2D artist. Otherwise, boxes are drawn with Patch artists.

`tick_labels` : list of str, optional

The tick labels of each boxplot.

Ticks are always placed at the box `*positions*`. If `*tick_labels*` is given, the ticks are labelled accordingly. Otherwise, they keep their numeric values.

.. versionchanged:: 3.9

Renamed from `*labels*`, which is deprecated since 3.9 and will be removed in 3.11.

`manage_ticks` : bool, default: True

If True, the tick locations and labels will be adjusted to match the boxplot positions.

`autorange` : bool, default: False

When ``True`` and the data are distributed such that the 25th and 75th percentiles are equal, `*whis*` is set to (0, 100) such that the whisker ends are at the minimum and maximum of the data.

`meanline` : bool, default: :rc:`boxplot.meanline`

If ``True`` (and `*showmeans*` is ``True``), will try to render the mean as a line spanning the full width of the box according to `*meanprops*` (see below). Not recommended if `*shownotches*` is also True. Otherwise, means will be shown as points.

`zorder` : float, default: ``Line2D.zorder = 2``

The zorder of the boxplot.

## Returns

-----

dict

A dictionary mapping each component of the boxplot to a list of the `.Line2D`` instances created. That dictionary has the following keys (assuming vertical boxplots):

- ```boxes```: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- ```medians```: horizontal lines at the median of each box.
- ```whiskers```: the vertical lines extending to the most extreme, non-outlier data points.

- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

#### Other Parameters

-----

`showcaps` : bool, default: `rc['boxplot.showcaps']`  
 Show the caps on the ends of whiskers.

`showbox` : bool, default: `rc['boxplot.showbox']`  
 Show the central box.

`showfliers` : bool, default: `rc['boxplot.showfliers']`  
 Show the outliers beyond the caps.

`showmeans` : bool, default: `rc['boxplot.showmeans']`  
 Show the arithmetic means.

`capprops` : dict, default: None  
 The style of the caps.

`capwidths` : float or array, default: None  
 The widths of the caps.

`boxprops` : dict, default: None  
 The style of the box.

`whiskerprops` : dict, default: None  
 The style of the whiskers.

`flierprops` : dict, default: None  
 The style of the fliers.

`medianprops` : dict, default: None  
 The style of the median.

`meanprops` : dict, default: None  
 The style of the mean.

`label` : str or list of str, optional  
 Legend labels. Use a single string when all boxes have the same style and you only want a single legend entry for them. Use a list of strings to label all boxes individually. To be distinguishable, the boxes should be styled individually, which is currently only possible by modifying the returned artists, see e.g. `doc:/gallery/statistics/boxplot_demo`.

In the case of a single string, the legend entry will technically be associated with the first box only. By default, the legend will show the median line (`result["medians"]`); if `patch_artist` is True, the legend will show the box `result["boxes"]` instead.

.. versionadded:: 3.9

`data` : indexable object, optional  
 If given, all parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`.

#### See Also

-----

`.Axes.bxp` : Draw a boxplot from pre-computed statistics.  
`violinplot` : Draw an estimate of the probability density function.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.boxplot``.

## matplotlib.pyplot.broken\_barh

```
broken_barh(xranges: 'Sequence[tuple[float, float]]', yrange: 'tuple[float, float]',
 *, data=None, **kwargs) -> 'PolyCollection'
```

Plot a horizontal sequence of rectangles.

A rectangle is drawn for each element of `*xranges*`. All rectangles have the same vertical position and size defined by `*yrange*`.

Parameters

-----

`xranges` : sequence of tuples (`*xmin*`, `*xwidth*`)

The x-positions and extents of the rectangles. For each tuple (`*xmin*`, `*xwidth*`) a rectangle is drawn from `*xmin*` to `*xmin* + *xwidth*`.

`yrange` : (`*ymin*`, `*yheight*`)

The y-position and extent for all the rectangles.

Returns

-----

`~.collections.PolyCollection``

Other Parameters

-----

`data` : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

`**kwargs` : `~.PolyCollection`` properties

Each `*kwarg*` can be either a single argument applying to all rectangles, e.g.::

```
facecolors='black'
```

or a sequence of arguments over which is cycled, e.g.::

```
facecolors=('black', 'blue')
```

would create interleaving black and blue rectangles.

Supported keywords:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: array-like or float or None

animated: bool  
 antialiased or aa or antialiaseds: bool or list of bools  
 array: array-like or None  
 capstyle: `~.CapStyle`` or `{'butt', 'projecting', 'round'}`  
 clim: (vmin: float, vmax: float)  
 clip\_box: `~matplotlib.transforms.BboxBase`` or None  
 clip\_on: bool  
 clip\_path: Patch or (Path, Transform) or None  
 cmap: `~.Colormap`` or str or None  
 color: :mpltype:``color`` or list of RGBA tuples  
 edgecolor or ec or edgecolors: :mpltype:``color`` or list of :mpltype:``color`` or 'face'  
 facecolor or facecolors or fc: :mpltype:``color`` or list of :mpltype:``color``  
 figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``  
 gid: str  
 hatch: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`  
 hatch\_linewidth: unknown  
 in\_layout: bool  
 joinstyle: `~.JoinStyle`` or `{'miter', 'round', 'bevel'}`  
 label: object  
 linestyle or dashes or linestyles or ls: str or tuple or list thereof  
 linewidth or linewidths or lw: float or list of floats  
 mouseover: bool  
 norm: `~.Normalize`` or str or None  
 offset\_transform or transOffset: `~.Transform``  
 offsets: (N, 2) or (2,) array-like  
 path\_effects: list of `~.AbstractPathEffect``  
 paths: list of array-like  
 picker: None or bool or float or callable  
 pickradius: float  
 rasterized: bool  
 sizes: `~numpy.ndarray`` or None  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 transform: `~matplotlib.transforms.Transform``  
 url: str  
 urls: list of str or None  
 verts: list of array-like  
 verts\_and\_codes: unknown  
 visible: bool  
 zorder: float

## Notes

-----

.. note::

This is the :ref:`pyplot` wrapper `<pyplot_interface>` for `~.axes.Axes.broken_barh``.

## matplotlib.pyplot.cla

```
cla() -> 'None'
```

Clear the current Axes.

## matplotlib.pyplot.clabel

```
clabel(CS: 'ContourSet', levels: 'ArrayLike | None' = None, **kwargs) -> 'list[Text]'
```

Label a contour plot.

Adds labels to line contours in given `ContourSet`.

Parameters

-----

`CS` : `ContourSet` instance

Line contours to label.

`levels` : array-like, optional

A list of level values, that should be labeled. The list must be a subset of `CS.levels`. If not given, all levels are labeled.

`**kwargs`

All other parameters are documented in `~.ContourLabeler.clabel`.

Notes

-----

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.clabel`.

## matplotlib.pyplot.clf

```
clf() -> 'None'
```

Clear the current figure.

## matplotlib.pyplot.clim

```
clim(vmin: 'float | None' = None, vmax: 'float | None' = None) -> 'None'
```

Set the color limits of the current image.

If either `*vmin*` or `*vmax*` is `None`, the image min/max respectively will be used for color scaling.

If you want to set the `clim` of multiple images, use `~.ScalarMappable.set_clim` on every image, for example::

```
for im in gca().get_images():
 im.set_clim(0, 0.5)
```

## matplotlib.pyplot.close

```
close(fig: "None | int | str | Figure | Literal['all']" = None) -> 'None'
```

Close a figure window, and unregister it from pyplot.

Parameters

-----

fig : None or int or str or ``Figure``

The figure to close. There are a number of ways to specify this:

- `*None*`: the current figure
- ``Figure``: the given ``Figure`` instance
- ``int``: a figure number
- ``str``: a figure name
- `'all'`: all figures

#### Notes

-----

pyplot maintains a reference to figures created with `figure()`. When work on the figure is completed, it should be closed, i.e. deregistered from pyplot, to free its memory (see also `:rc:figure.max_open_warning`). Closing a figure window created by `show()` automatically deregisters the figure. For all other use cases, most prominently `savefig()` without `show()`, the figure must be deregistered explicitly using `close()`.

## matplotlib.pyplot.cohere

```
cohere(x: 'ArrayLike', y: 'ArrayLike', *, NFFT: 'int' = 256, Fs: 'float' = 2, Fc:
'int' = 0, detrend: "Literal['none', 'mean', 'linear'] | Callable[[ArrayLike],
ArrayLike]" = , window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike' = , noverlap:
'int' = 0, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided',
'twosided']" = 'default', scale_by_freq: 'bool | None' = None, data=None, **kwargs) ->
'tuple[np.ndarray, np.ndarray]'
```

Plot the coherence between `*x*` and `*y*`.

Coherence is the normalized cross spectral density:

.. math::

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

#### Parameters

-----

Fs : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

window : callable or ndarray, default: ``window_hanning``

A function or a vector of length `*NFFT*`. To create window vectors see ``window_hanning``, ``window_none``, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad\_to : int, optional

The number of points to which the data segment is padded when performing

the FFT. This can be different from `*NFFT*`, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to `~numpy.fft.fft`. The default is `None`, which sets `*pad_to*` equal to `*NFFT*`

`NFFT` : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use `*pad_to*` for this instead.

`detrend` : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the `*detrend*` parameter is a vector, in Matplotlib it is a function. The `:mod:`~matplotlib.mlab`` module defines `~.detrend_none`, `~.detrend_mean`, and `~.detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `~.detrend_none`. 'mean' calls `~.detrend_mean`. 'linear' calls `~.detrend_linear`.

`scale_by_freq` : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

`noverlap` : int, default: 0 (no overlap)

The number of points of overlap between blocks.

`Fc` : int, default: 0

The center frequency of `*x*`, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

-----

`Cxy` : 1-D array

The coherence vector.

`freqs` : 1-D array

The frequencies for the elements in `*Cxy*`.

Other Parameters

-----

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*y*`

`**kwargs`

Keyword arguments control the `~.Line2D`` properties:

Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image  
 alpha: float or None  
 animated: bool  
 antialiased or aa: bool  
 clip\_box: `~matplotlib.transforms.BboxBase`` or None  
 clip\_on: bool  
 clip\_path: Patch or (Path, Transform) or None  
 color or c: `:mpltype:`color``  
 dash\_capstyle: ``.CapStyle`` or {'butt', 'projecting', 'round'}  
 dash\_joinstyle: ``.JoinStyle`` or {'miter', 'round', 'bevel'}  
 dashes: sequence of floats (on/off ink in points) or (None, None)  
 data: (2, N) array or two 1D arrays  
 drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'  
 figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``  
 fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}  
 gapcolor: `:mpltype:`color`` or None  
 gid: str  
 in\_layout: bool  
 label: object  
 linestyle or ls: {'-', '--', '-.', ':', ''}, (offset, on-off-seq), ...}  
 linewidth or lw: float  
 marker: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``  
 markeredgewidth or mec: `:mpltype:`color``  
 markeredgewidth or mew: float  
 markerfacecolor or mfc: `:mpltype:`color``  
 markerfacecoloralt or mfcalt: `:mpltype:`color``  
 markersize or ms: float  
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]  
 mouseover: bool  
 path\_effects: list of ``.AbstractPathEffect``  
 picker: float or callable[[Artist, Event], tuple[bool, dict]]  
 pickradius: float  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 solid\_capstyle: ``.CapStyle`` or {'butt', 'projecting', 'round'}  
 solid\_joinstyle: ``.JoinStyle`` or {'miter', 'round', 'bevel'}  
 transform: unknown  
 url: str  
 visible: bool  
 xdata: 1D array  
 ydata: 1D array  
 zorder: float

## Notes

-----

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for `~.axes.Axes.cohere``.

## References

-----

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,

**matplotlib.pyplot.colorbar**

```
colorbar(mappable: 'ScalarMappable | ColorizingArtist | None' = None, cax:
'matplotlib.axes.Axes | None' = None, ax: 'matplotlib.axes.Axes |
Iterable[matplotlib.axes.Axes] | None' = None, **kwargs) -> 'Colorbar'
```

Add a colorbar to a plot.

**Parameters**

-----

**mappable**

The `matplotlib.cm.ScalarMappable` (i.e., `.AxesImage`, `.ContourSet`, etc.) described by this colorbar. This argument is mandatory for the `.Figure.colorbar` method but optional for the `.pyplot.colorbar` function, which sets the default to the current image.

Note that one can create a `.ScalarMappable` "on-the-fly" to generate colorbars not attached to a previously drawn artist, e.g.  
::

```
fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap), ax=ax)
```

**cax** : `~matplotlib.axes.Axes`, optional

Axes into which the colorbar will be drawn. If `None`, then a new Axes is created and the space for it will be stolen from the Axes(s) specified in `*ax*`.

**ax** : `~matplotlib.axes.Axes` or iterable or `numpy.ndarray` of Axes, optional

The one or more parent Axes from which space for a new colorbar Axes will be stolen. This parameter is only used if `*cax*` is not set.

Defaults to the Axes that contains the mappable used to create the colorbar.

**use\_gridspec** : bool, optional

If `*cax*` is `None`, a new `*cax*` is created as an instance of Axes. If `*ax*` is positioned with a `subplotspec` and `*use_gridspec*` is `True`, then `*cax*` is also positioned with a `subplotspec`.

**Returns**

-----

colorbar : `~matplotlib.colorbar.Colorbar`

**Other Parameters**

-----

**location** : None or {'left', 'right', 'top', 'bottom'}

The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the `*orientation*` of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the `*orientation*` if it is set (vertical colorbars on the right, horizontal

ones at the bottom), or default to 'right' if *\*orientation\** is unset.

*orientation* : None or {'vertical', 'horizontal'}

The orientation of the colorbar. It is preferable to set the *\*location\** of the colorbar, as that also determines the *\*orientation\**; passing incompatible values for *\*location\** and *\*orientation\** raises an exception.

*fraction* : float, default: 0.15

Fraction of original Axes to use for colorbar.

*shrink* : float, default: 1.0

Fraction by which to multiply the size of the colorbar.

*aspect* : float, default: 20

Ratio of long to short dimensions.

*pad* : float, default: 0.05 if vertical, 0.15 if horizontal

Fraction of original Axes between colorbar and new image Axes.

*anchor* : (float, float), optional

The anchor point of the colorbar Axes.

Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

*panchor* : (float, float), or *\*False\**, optional

The anchor point of the colorbar parent Axes. If *\*False\**, the parent axes' anchor will be unchanged.

Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

*extend* : {'neither', 'both', 'min', 'max'}

Make pointed end(s) for out-of-range values (unless 'neither'). These are set for a given colormap using the colormap *set\_under* and *set\_over* methods.

*extendfrac* : {'\*None\*', 'auto', length, lengths}

If set to *\*None\**, both the minimum and maximum triangular colorbar extensions will have a length of 5% of the interior colorbar length (this is the default setting).

If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *\*spacing\** is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *\*spacing\** is set to 'proportional').

If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.

*extendrect* : bool

If *\*False\** the minimum and maximum colorbar extensions will be triangular (the default). If *\*True\** the extensions will be rectangular.

*ticks* : None or list of ticks or Locator

If None, ticks are determined automatically from the input.

format : None or str or Formatter  
If None, `~.ticker.ScalarFormatter` is used.  
Format strings, e.g., ```"%4.2e"``` or ```"{x:.2e}"```, are supported.  
An alternative `~.ticker.Formatter` may be given instead.

drawedges : bool  
Whether to draw lines at color boundaries.

label : str  
The label on the colorbar's long axis.

boundaries, values : None or a sequence  
If unset, the colormap will be displayed on a 0-1 scale.  
If sequences, *\*values\** must have a length 1 less than *\*boundaries\**. For each region delimited by adjacent entries in *\*boundaries\**, the color mapped to the corresponding value in *\*values\** will be used. The size of each region is determined by the *\*spacing\** parameter.  
Normally only useful for indexed colors (i.e. ```norm=NoNorm()```) or other unusual circumstances.

spacing : {'uniform', 'proportional'}  
For discrete colorbars (`.BoundaryNorm` or contours), 'uniform' gives each color the same space; 'proportional' makes the space proportional to the data interval.

Notes  
-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.Figure.colorbar`.

If *\*mappable\** is a `~.contour.ContourSet`, its *\*extend\** kwarg is included automatically.

The *\*shrink\** kwarg provides a simple way to scale the colorbar with respect to the Axes. Note that if *\*cax\** is specified, it determines the size of the colorbar, and *\*shrink\** and *\*aspect\** are ignored.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the Axes properties kwargs.

It is known that some vector graphics viewers (svg and pdf) render white gaps between segments of the colorbar. This is due to bugs in the viewers, not Matplotlib. As a workaround, the colorbar can be rendered with overlapping segments::

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However, this has negative consequences in other circumstances, e.g. with semi-transparent images ( $\alpha < 1$ ) and colorbar extensions; therefore, this workaround is not used by default (see issue #1188).

## matplotlib.pyplot.connect

```
connect(s: 'str', func: 'Callable[[Event], Any]') -> 'int'
```

Bind function \*func\* to event \*s\*.

### Parameters

-----

s : str

One of the following events ids:

- 'button\_press\_event'
- 'button\_release\_event'
- 'draw\_event'
- 'key\_press\_event'
- 'key\_release\_event'
- 'motion\_notify\_event'
- 'pick\_event'
- 'resize\_event'
- 'scroll\_event'
- 'figure\_enter\_event',
- 'figure\_leave\_event',
- 'axes\_enter\_event',
- 'axes\_leave\_event'
- 'close\_event'.

func : callable

The callback function to be executed, which must have the signature::

```
def func(event: Event) -> Any
```

For the location events (button and key press/release), if the mouse is over the Axes, the ``inaxes`` attribute of the event will be set to the `~matplotlib.axes.Axes` the event occurs is over, and additionally, the variables ``xdata`` and ``ydata`` attributes will be set to the mouse location in data coordinates. See `.KeyEvent` and `.MouseEvent` for more info.

.. note::

If func is a method, this only stores a weak reference to the method. Thus, the figure does not influence the lifetime of the associated object. Usually, you want to make sure that the object is kept alive throughout the lifetime of the figure by holding a reference to it.

### Returns

-----

cid

A connection id that can be used with `.FigureCanvasBase.mpl_disconnect`.

### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.FigureCanvasBase.mpl_connect``.

Examples

-----

::

```
def on_press(event):
 print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

## matplotlib.pyplot.contour

```
contour(*args, data=None, **kwargs) -> 'QuadContourSet'
```

Plot contour lines.

Call signature::

```
contour([X, Y,] Z, /, [levels], **kwargs)
```

The arguments `*X*`, `*Y*`, `*Z*` are positional-only.

`.contour`` and `.contourf`` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

Parameters

-----

X, Y : array-like, optional

The coordinates of the values in `*Z*`.

`*X*` and `*Y*` must both be 2D with the same shape as `*Z*` (e.g. created via ``numpy.meshgrid``), or they must both be 1-D such that ``len(X) == N`` is the number of columns in `*Z*` and ``len(Y) == M`` is the number of rows in `*Z*`.

`*X*` and `*Y*` must both be ordered monotonically.

If not given, they are assumed to be integer indices, i.e.

``X = range(N)``, ``Y = range(M)``.

Z : (M, N) array-like

The height values over which the contour is drawn. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

levels : int or array-like, optional

Determines the number and positions of the contour lines / regions.

If an int `*n*`, use ``~matplotlib.ticker.MaxNLocator``, which tries to automatically choose no more than `*n+1*` "nice" contour levels

between minimum and maximum numeric values of *\*Z\**.

If array-like, draw contour lines at the specified levels.  
The values must be in increasing order.

Returns

-----  
``~.contour.QuadContourSet``

Other Parameters

-----  
`corner_mask` : bool, default: `:rc:`contour.corner_mask``  
Enable/disable corner masking, which only has an effect if *\*Z\** is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

`colors` : :mpltype:`color` or list of :mpltype:`color`, optional  
The colors of the levels, i.e. the lines for ``~.contour`` and the areas for ``~.contourf``.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, a single color may be used in place of one-element lists, i.e. `red` instead of `[red]` to color all levels with the same color.

.. versionchanged:: 3.10

Previously a single color had to be expressed as a string, but now any valid color format may be passed.

By default (value *\*None\**), the colormap specified by *\*cmap\** will be used.

`alpha` : float, default: 1  
The alpha blending value, between 0 (transparent) and 1 (opaque).

`cmap` : str or ``~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``  
The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *\*colors\** is set.

`norm` : str or ``~matplotlib.colors.Normalize``, optional  
The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *\*cmap\**. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of ``~matplotlib.colors.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call ``~matplotlib.scale.get_scale_names``.

In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin, vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/*vmax*` when a `*norm*` instance is given (but using a ``str`` `*norm*` name together with `*vmin*/*vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`colorizer` : `~matplotlib.colorbar.Colorizer`` or None, default: None  
The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

This parameter is ignored if `*colors*` is set.

`origin` : {`*None*`, `'upper'`, `'lower'`, `'image'`}, default: None  
Determines the orientation and exact position of `*Z*` by specifying the position of ```Z[0, 0]```. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: ```Z[0, 0]``` is at `X=0, Y=0` in the lower left corner.
- `'lower'`: ```Z[0, 0]``` is at `X=0.5, Y=0.5` in the lower left corner.
- `'upper'`: ```Z[0, 0]``` is at `X=N+0.5, Y=0.5` in the upper left corner.
- `'image'`: Use the value from `:rc:`image.origin``.

`extent` : (`x0`, `x1`, `y0`, `y1`), optional

If `*origin*` is not `*None*`, then `*extent*` is interpreted as in `.imshow``: it gives the outer pixel boundaries. In this case, the position of `Z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then (`*x0*`, `*y0*`) is the position of `Z[0, 0]`, and (`*x1*`, `*y1*`) is the position of `Z[-1, -1]`.

This argument is ignored if `*X*` and `*Y*` are specified in the call to contour.

`locator` : `ticker.Locator` subclass, optional

The locator is used to determine the contour levels if they are not given explicitly via `*levels*`.  
Defaults to `~.ticker.MaxNLocator``.

`extend` : {`'neither'`, `'both'`, `'min'`, `'max'`}, default: `'neither'`

Determines the ```contourf```-coloring of values that are outside the `*levels*` range.

If `'neither'`, values outside the `*levels*` range are not colored.

If `'min'`, `'max'` or `'both'`, color the values below, above or below

and above the `*levels*` range.

Values below ```min(levels)``` and above ```max(levels)``` are mapped to the under/over values of the `.Colormap``. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using `.Colormap.set_under`` and `.Colormap.set_over``.

.. note::

An existing `.QuadContourSet`` does not get notified if properties of its colormap are changed. Therefore, an explicit call `~.ContourSet.changed()``` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the `.QuadContourSet`` because it internally calls `~.ContourSet.changed()```.

Example::

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
 colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

xunits, yunits : registered units, optional  
Override axis units by specifying an instance of a  
:class:`matplotlib.units.ConversionInterface`.

antialiased : bool, optional  
Enable antialiasing, overriding the defaults. For  
filled contours, the default is `*False*`. For line contours,  
it is taken from `:rc:`lines.antialiased``.

nchunk : int >= 0, optional  
If 0, no subdivision of the domain. Specify a positive integer to  
divide the domain into subdomains of `*nchunk*` by `*nchunk*` quads.  
Chunking reduces the maximum length of polygons generated by the  
contouring algorithm which reduces the rendering workload passed  
on to the backend and also requires slightly less RAM. It can  
however introduce rendering artifacts at chunk boundaries depending  
on the backend, the `*antialiased*` flag and value of `*alpha*`.

linewidths : float or array-like, default: `:rc:`contour.linewidth``  
`*Only applies to* ``.contour``.`

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to `:rc:`lines.linewidth``.

`linestyles` : `{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}`, optional  
\*Only applies to\* ``contour``.

If `*linestyles*` is `*None*`, the default is 'solid' unless the lines are monochrome. In that case, negative contours will instead take their linestyle from the `*negative_linestyles*` argument.

`*linestyles*` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

`negative_linestyles` : `{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}`, optional  
\*Only applies to\* ``contour``.

If `*linestyles*` is `*None*` and the lines are monochrome, this argument specifies the line style for negative contours.

If `*negative_linestyles*` is `*None*`, the default is taken from `:rc:`contour.negative_linestyle``.

`*negative_linestyles*` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

`hatches` : `list[str]`, optional  
\*Only applies to\* ``contourf``.

A list of cross hatch patterns to use on the filled areas.  
If None, no hatching will be added to the contour.

`algorithm` : `{'mpl2005', 'mpl2014', 'serial', 'threaded'}`, optional  
Which contouring algorithm to use to calculate the contour lines and polygons. The algorithms are implemented in  
``ContourPy`` <<https://github.com/contourpy/contourpy>>`, consult the  
``ContourPy`` documentation <<https://contourpy.readthedocs.io>>` for further information.

The default is taken from `:rc:`contour.algorithm``.

`clip_path` : `~matplotlib.patches.Patch`` or ``Path`` or ``TransformedPath``  
Set the clip path. See `~matplotlib.artist.Artist.set_clip_path``.

.. versionadded:: 3.8

`data` : indexable object, optional  
If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.contour``.

1. `.contourf`` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to `.contour``.

2. `.contourf`` fills intervals that are closed at the top; that is, for boundaries `*z1*` and `*z2*`, the filled region is::

`z1 < Z <= z2`

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

3. `.contour`` and `.contourf`` use a ``marching squares <https://en.wikipedia.org/wiki/Marching\_squares>`` algorithm to compute contour locations. More information can be found in ``ContourPy documentation <https://contourpy.readthedocs.io>``.

## matplotlib.pyplot.contourf

```
contourf(*args, data=None, **kwargs) -> 'QuadContourSet'
```

Plot filled contours.

Call signature::

`contourf([X, Y,] Z, /, [levels], **kwargs)`

The arguments `*X*`, `*Y*`, `*Z*` are positional-only.

`.contour`` and `.contourf`` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

Parameters

-----

`X, Y` : array-like, optional

The coordinates of the values in `*Z*`.

`*X*` and `*Y*` must both be 2D with the same shape as `*Z*` (e.g. created via ``numpy.meshgrid``), or they must both be 1-D such that ``len(X) == N`` is the number of columns in `*Z*` and ``len(Y) == M`` is the number of rows in `*Z*`.

`*X*` and `*Y*` must both be ordered monotonically.

If not given, they are assumed to be integer indices, i.e. ``X = range(N)``, ``Y = range(M)``.

`Z` : (M, N) array-like

The height values over which the contour is drawn. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

`levels` : int or array-like, optional

Determines the number and positions of the contour lines / regions.

If an int `*n*`, use `~matplotlib.ticker.MaxNLocator``, which tries to automatically choose no more than `*n+1*` "nice" contour levels between minimum and maximum numeric values of `*Z*`.

If array-like, draw contour lines at the specified levels.

The values must be in increasing order.

Returns

-----  
`~.contour.QuadContourSet``

Other Parameters

-----  
`corner_mask` : bool, default: `:rc:`contour.corner_mask``  
Enable/disable corner masking, which only has an effect if `*Z*` is a masked array. If ```False```, any quad touching a masked point is masked out. If ```True```, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

`colors` : `:mpltype:`color`` or list of `:mpltype:`color``, optional  
The colors of the levels, i.e. the lines for `~.contour`` and the areas for `~.contourf``.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, a single color may be used in place of one-element lists, i.e. ```red``` instead of ```[red]``` to color all levels with the same color.

.. versionchanged:: 3.10

Previously a single color had to be expressed as a string, but now any valid color format may be passed.

By default (value `*None*`), the colormap specified by `*cmap*` will be used.

`alpha` : float, default: 1

The alpha blending value, between 0 (transparent) and 1 (opaque).

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``  
The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*colors*` is set.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is

used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call ``matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin, vmax : float, optional`

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str`` `*norm*` name together with `*vmin*/vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`colorizer : ~matplotlib.colorbar.Colorizer` or None, default: None`

The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

This parameter is ignored if `*colors*` is set.

`origin : {None, 'upper', 'lower', 'image'}, default: None`

Determines the orientation and exact position of `*Z*` by specifying the position of ``Z[0, 0]``. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: ``Z[0, 0]`` is at `X=0, Y=0` in the lower left corner.
- `'lower'`: ``Z[0, 0]`` is at `X=0.5, Y=0.5` in the lower left corner.
- `'upper'`: ``Z[0, 0]`` is at `X=N+0.5, Y=0.5` in the upper left corner.
- `'image'`: Use the value from `:rc:`image.origin``.

`extent : (x0, x1, y0, y1), optional`

If `*origin*` is not `*None*`, then `*extent*` is interpreted as in `.imshow``: it gives the outer pixel boundaries. In this case, the position of `Z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then `(*x0*, *y0*)` is the position of `Z[0, 0]`, and `(*x1*, *y1*)` is the position of `Z[-1, -1]`.

This argument is ignored if `*X*` and `*Y*` are specified in the call to contour.

`locator : ticker.Locator subclass, optional`

The locator is used to determine the contour levels if they are not given explicitly via `*levels*`.

Defaults to `~.ticker.MaxNLocator`.

`extend` : {'neither', 'both', 'min', 'max'}, default: 'neither'

Determines the `contourf`-coloring of values that are outside the `levels` range.

If 'neither', values outside the `levels` range are not colored.

If 'min', 'max' or 'both', color the values below, above or below and above the `levels` range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the `.Colormap`. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using `.Colormap.set_under` and `.Colormap.set_over`.

.. note::

An existing `.QuadContourSet` does not get notified if properties of its colormap are changed. Therefore, an explicit call `~.ContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the `.QuadContourSet` because it internally calls `~.ContourSet.changed()`.

Example::

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y
```

```
cs = plt.contourf(h, levels=[10, 30, 50],
 colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

`xunits`, `yunits` : registered units, optional  
Override axis units by specifying an instance of a `:class:`matplotlib.units.ConversionInterface``.

`antialiased` : bool, optional  
Enable antialiasing, overriding the defaults. For filled contours, the default is `*False*`. For line contours, it is taken from `:rc:`lines.antialiased``.

`nchunk` : int  $\geq 0$ , optional  
If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of `*nchunk*` by `*nchunk*` quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the `*antialiased*` flag and value of `*alpha*`.

linewidths : float or array-like, default: :rc:`contour.linewidth`  
\*Only applies to\* `contour`.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to :rc:`lines.linewidth`.

linestyles : {`None`, 'solid', 'dashed', 'dashdot', 'dotted'}, optional  
\*Only applies to\* `contour`.

If `linestyles` is `None`, the default is 'solid' unless the lines are monochrome. In that case, negative contours will instead take their linestyle from the `negative_linestyles` argument.

`linestyles` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

negative\_linestyles : {`None`, 'solid', 'dashed', 'dashdot', 'dotted'}, optional  
\*Only applies to\* `contour`.

If `linestyles` is `None` and the lines are monochrome, this argument specifies the line style for negative contours.

If `negative_linestyles` is `None`, the default is taken from :rc:`contour.negative\_linestyle`.

`negative_linestyles` can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

hatches : list[str], optional  
\*Only applies to\* `contourf`.

A list of cross hatch patterns to use on the filled areas.

If None, no hatching will be added to the contour.

algorithm : {'mpl2005', 'mpl2014', 'serial', 'threaded'}, optional  
Which contouring algorithm to use to calculate the contour lines and polygons. The algorithms are implemented in `ContourPy` <<https://github.com/contourpy/contourpy>>, consult the `ContourPy` documentation <<https://contourpy.readthedocs.io>> for further information.

The default is taken from :rc:`contour.algorithm`.

clip\_path : ~matplotlib.patches.Patch or `Path` or `TransformedPath`  
Set the clip path. See ~matplotlib.artist.Artist.set\_clip\_path.

.. versionadded:: 3.8

data : indexable object, optional

If given, all parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.contourf`.

1. `.contourf` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to `.contour`.

2. `.contourf` fills intervals that are closed at the top; that is, for boundaries `*z1*` and `*z2*`, the filled region is::

$z1 < Z \leq z2$

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

3. `.contour` and `.contourf` use a ``marching squares <https://en.wikipedia.org/wiki/Marching\_squares>`` algorithm to compute contour locations. More information can be found in ``ContourPy documentation <https://contourpy.readthedocs.io>``.

## matplotlib.pyplot.cool

```
cool() -> 'None'
```

Set the colormap to 'cool'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

## matplotlib.pyplot.copper

```
copper() -> 'None'
```

Set the colormap to 'copper'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

## matplotlib.pyplot.csd

```
csd(x: 'ArrayLike', y: 'ArrayLike', *, NFFT: 'int | None' = None, Fs: 'float | None' = None, Fc: 'int | None' = None, detrend: "Literal['none', 'mean', 'linear'] | Callable[[ArrayLike], ArrayLike] | None" = None, window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, noverlap: 'int | None' = None, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None, scale_by_freq: 'bool | None' = None, return_line: 'bool | None' = None, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray] | tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the cross-spectral density.

The cross spectral density :math:`P\_{xy}` by Welch's average periodogram method. The vectors  $x$  and  $y$  are divided into  $NFFT$  length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The product of the direct FFTs of  $x$  and  $y$  are averaged over each segment to compute :math:`P\_{xy}`, with a scaling to correct for power loss due to windowing.

If  $\text{len}(x) < NFFT$  or  $\text{len}(y) < NFFT$ , they will be zero padded to  $NFFT$ .

#### Parameters

`x, y` : 1-D arrays or sequences  
Arrays or sequences containing the data.

`Fs` : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `freqs`, in cycles per time unit.

`window` : callable or ndarray, default: `~.window_hanning`

A function or a vector of length  $NFFT$ . To create window vectors see `~.window_hanning`, `~.window_none`, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. This can be different from  $NFFT$ , which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `n` parameter in the call to `~numpy.fft.fft`. The default is None, which sets `pad_to` equal to  $NFFT$ .

`NFFT` : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should NOT be used to get zero padding, or the scaling of the result will be incorrect; use `pad_to` for this instead.

`detrend` : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the `*detrend*` parameter is a vector, in Matplotlib it is a function. The `:mod:`~matplotlib.mlab`` module defines ``.detrend_none``, ``.detrend_mean``, and ``.detrend_linear``, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls ``.detrend_none``. 'mean' calls ``.detrend_mean``. 'linear' calls ``.detrend_linear``.

`scale_by_freq` : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

`noverlap` : int, default: 0 (no overlap)

The number of points of overlap between segments.

`Fc` : int, default: 0

The center frequency of `*x*`, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

`return_line` : bool, default: False

Whether to include the line object plotted in the returned values.

## Returns

-----

`Pxy` : 1-D array

The values for the cross spectrum  $P_{xy}$  before scaling (complex valued).

`freqs` : 1-D array

The frequencies corresponding to the elements in `*Pxy*`.

`line` : `~matplotlib.lines.Line2D``

The line created by this function.

Only returned if `*return_line*` is True.

## Other Parameters

-----

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*y*`

`**kwargs`

Keyword arguments control the ``.Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

antialiased or aa: bool  
 clip\_box: `~matplotlib.transforms.BboxBase`` or None  
 clip\_on: bool  
 clip\_path: Patch or (Path, Transform) or None  
 color or c: `:mpltype:`color``  
 dash\_capstyle: ``.CapStyle`` or {'butt', 'projecting', 'round'}  
 dash\_joinstyle: ``.JoinStyle`` or {'miter', 'round', 'bevel'}  
 dashes: sequence of floats (on/off ink in points) or (None, None)  
 data: (2, N) array or two 1D arrays  
 drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'  
 figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``  
 fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}  
 gapcolor: `:mpltype:`color`` or None  
 gid: str  
 in\_layout: bool  
 label: object  
 linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}  
 linewidth or lw: float  
 marker: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``  
 markeredgewidth or mec: `:mpltype:`color``  
 markeredgewidth or mew: float  
 markerfacecolor or mfc: `:mpltype:`color``  
 markerfacecoloralt or mfcalt: `:mpltype:`color``  
 markersize or ms: float  
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]  
 mouseover: bool  
 path\_effects: list of ``.AbstractPathEffect``  
 picker: float or callable[[Artist, Event], tuple[bool, dict]]  
 pickradius: float  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 solid\_capstyle: ``.CapStyle`` or {'butt', 'projecting', 'round'}  
 solid\_joinstyle: ``.JoinStyle`` or {'miter', 'round', 'bevel'}  
 transform: unknown  
 url: str  
 visible: bool  
 xdata: 1D array  
 ydata: 1D array  
 zorder: float

#### See Also

-----

psd : is equivalent to setting ```y = x```.

#### Notes

-----

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``.axes.Axes.csd``.

For plotting, the power is plotted as

`:math:`10 \log_{10}(P_{xy})`` for decibels, though `:math:`P_{xy}`` itself is returned.

## References

-----  
Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,  
John Wiley & Sons (1986)

## matplotlib.pyplot.delaxes

```
delaxes(ax: 'matplotlib.axes.Axes | None' = None) -> 'None'
```

Remove an `~.axes.Axes`` (defaulting to the current Axes) from its figure.

## matplotlib.pyplot.disconnect

```
disconnect(cid: 'int') -> 'None'
```

Disconnect the callback with id `*cid*`.

### Notes

-----

.. note::

This is the `:ref:`pyplot` wrapper <pyplot_interface> for ~.FigureCanvasBase.mpl_disconnect`.`

### Examples

-----

::

```
cid = canvas.mpl_connect('button_press_event', on_press)
... later
canvas.mpl_disconnect(cid)
```

## matplotlib.pyplot.draw

```
draw() -> 'None'
```

Redraw the current figure.

This is used to update a figure that has been altered, but not automatically re-drawn. If interactive mode is on (via `~.ion()``), this should be only rarely needed, but there may be ways to modify the state of a figure without marking it as "stale". Please report these cases as bugs.

This is equivalent to calling ```fig.canvas.draw_idle()```, where ```fig``` is the current figure.

### See Also

-----

`.FigureCanvasBase.draw_idle`  
`.FigureCanvasBase.draw`

## matplotlib.pyplot.ecdf

```
ecdf(x: 'ArrayLike', weights: 'ArrayLike | None' = None, *, complementary: 'bool' =
False, orientation: "Literal['vertical', 'horizontal']" = 'vertical', compress: 'bool'
= False, data=None, **kwargs) -> 'Line2D'
```

Compute and plot the empirical cumulative distribution function of `*x*`.

.. versionadded:: 3.8

#### Parameters

-----

`x` : 1d array-like

The input data. Infinite entries are kept (and move the relevant end of the ecdf from 0/1), but NaNs and masked values are errors.

`weights` : 1d array-like or None, default: None

The weights of the entries; must have the same shape as `*x*`.

Weights corresponding to NaN data points are dropped, and then the remaining weights are normalized to sum to 1. If unset, all entries have the same weight.

`complementary` : bool, default: False

Whether to plot a cumulative distribution function, which increases from 0 to 1 (the default), or a complementary cumulative distribution function, which decreases from 1 to 0.

`orientation` : {"vertical", "horizontal"}, default: "vertical"

Whether the entries are plotted along the x-axis ("vertical", the default) or the y-axis ("horizontal"). This parameter takes the same values as in `~.Axes.hist``.

`compress` : bool, default: False

Whether multiple entries with the same values are grouped together (with a summed weight) before plotting. This is mainly useful if `*x*` contains many identical data points, to decrease the rendering complexity of the plot. If `*x*` contains no duplicate points, this has no effect and just uses some time and memory.

#### Other Parameters

-----

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*weights*`

`**kwargs`

Keyword arguments control the `~.Line2D`` properties:

#### Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

clip\_path: Patch or (Path, Transform) or None  
 color or c: :mpltype:`color`  
 dash\_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}  
 dash\_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}  
 dashes: sequence of floats (on/off ink in points) or (None, None)  
 data: (2, N) array or two 1D arrays  
 drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'  
 figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`  
 fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}  
 gapcolor: :mpltype:`color` or None  
 gid: str  
 in\_layout: bool  
 label: object  
 linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}  
 linewidth or lw: float  
 marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`  
 markeredgewidth or mec: :mpltype:`color`  
 markeredgewidth or mew: float  
 markerfacecolor or mfc: :mpltype:`color`  
 markerfacecoloralt or mfcalt: :mpltype:`color`  
 markersize or ms: float  
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]  
 mouseover: bool  
 path\_effects: list of `~.AbstractPathEffect`  
 picker: float or callable[[Artist, Event], tuple[bool, dict]]  
 pickradius: float  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 solid\_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}  
 solid\_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}  
 transform: unknown  
 url: str  
 visible: bool  
 xdata: 1D array  
 ydata: 1D array  
 zorder: float

## Returns

-----  
 `~.Line2D`

## Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.ecdf`.

The ecdf plot can be thought of as a cumulative histogram with one bin per data entry; i.e. it reports on the entire dataset without any arbitrary binning.

If *\*x\** contains NaNs or masked entries, either remove them first from the array (if they should not taken into account), or replace them by

-inf or +inf (if they should be sorted at the beginning or the end of the array).

## matplotlib.pyplot.errorbar

```
errorbar(x: 'float | ArrayLike', y: 'float | ArrayLike', yerr: 'float | ArrayLike | None' = None, xerr: 'float | ArrayLike | None' = None, fmt: 'str' = '', *, ecolor: 'ColorType | None' = None, elinewidth: 'float | None' = None, capsize: 'float | None' = None, barsabove: 'bool' = False, lolims: 'bool | ArrayLike' = False, uplims: 'bool | ArrayLike' = False, xlolims: 'bool | ArrayLike' = False, xuplims: 'bool | ArrayLike' = False, errorevery: 'int | tuple[int, int]' = 1, capthick: 'float | None' = None, data=None, **kwargs) -> 'ErrorbarContainer'
```

Plot y versus x as lines and/or markers with attached errorbars.

\*x\*, \*y\* define the data locations, \*xerr\*, \*yerr\* define the errorbar sizes. By default, this draws the data markers/lines as well as the errorbars. Use `fmt='none'` to draw errorbars without any data markers.

.. versionadded:: 3.7

Caps and error lines are drawn in polar coordinates on polar plots.

### Parameters

-----  
x, y : float or array-like  
The data positions.

xerr, yerr : float or array-like, shape(N,) or shape(2, N), optional  
The errorbar sizes:

- scalar: Symmetric +/- values for all data points.
- shape(N,): Symmetric +/-values for each data point.
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- \*None\*: No errorbar.

All values must be  $\geq 0$ .

See `:doc:`gallery/statistics/errorbar_features``  
for an example on the usage of ``xerr`` and ``yerr``.

fmt : str, default: "  
The format for the data points / data lines. See ``.plot`` for details.

Use 'none' (case-insensitive) to plot errorbars without any data markers.

ecolor : :mpltype:`color`, default: None  
The color of the errorbar lines. If None, use the color of the line connecting the markers.

elinewidth : float, default: None

The linewidth of the errorbar lines. If None, the linewidth of the current style is used.

capsize : float, default: :rc:`errorbar.capsize`  
The length of the error bar caps in points.

capthick : float, default: None  
An alias to the keyword argument `*markededgewidth*` (a.k.a. `*mew*`). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if `*mew*` or `*markededgewidth*` are given, then they will over-ride `*capthick*`. This may change in future releases.

barsabove : bool, default: False  
If True, will plot the errorbars above the plot symbols. Default is below.

lolims, uplims, xlolims, xuplims : bool or array-like, default: False  
These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. `*lims*`-arguments may be scalars, or array-likes of the same length as `*xerr*` and `*yerr*`. To use limits with inverted axes, `~.Axes.set_xlim`` or `~.Axes.set_ylim`` must be called before `:meth:`errorbar``. Note the tricky parameter names: setting e.g. `*lolims*` to True means that the y-value is a `*lower*` limit of the True value, so, only an `*upward*`-pointing arrow will be drawn!

errorevery : int or (int, int), default: 1  
draws error bars on a subset of the data. `*errorevery* = N` draws error bars on the points `(x[::N], y[::N])`.  
`*errorevery* = (start, N)` draws error bars on the points `(x[start::N], y[start::N])`. e.g. `errorevery=(6, 3)` adds error bars to the data at `(x[6], x[9], x[12], x[15], ...)`.  
Used to avoid overlapping error bars when two series share x-axis values.

## Returns

-----  
``ErrorbarContainer``  
The container contains:

- `data_line` : A ``matplotlib.lines.Line2D`` instance of x, y plot markers and/or line.
- `caplines` : A tuple of ``matplotlib.lines.Line2D`` instances of the error bar caps.
- `barlinecols` : A tuple of ``LineCollection`` with the horizontal and vertical error ranges.

## Other Parameters

-----  
`data` : indexable object, optional  
If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*, *y*, *xerr*, *yerr*`

`**kwargs`

All other keyword arguments are passed on to the `~.Axes.plot`` call drawing the markers. For example, this code makes big red squares with thick green edges::

```
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s', mfc='red',
mec='green', ms=20, mew=4)
```

where `*mfc*`, `*mec*`, `*ms*` and `*mew*` are aliases for the longer property names, `*markerfacecolor*`, `*markeredgecolor*`, `*markersize*` and `*markeredgewidth*`.

Valid kwargs for the marker properties are:

- `*dashes*`
- `*dash_capstyle*`
- `*dash_joinstyle*`
- `*drawstyle*`
- `*fillstyle*`
- `*linestyle*`
- `*marker*`
- `*markeredgecolor*`
- `*markeredgewidth*`
- `*markerfacecolor*`
- `*markerfacecoloralt*`
- `*markersize*`
- `*markevery*`
- `*solid_capstyle*`
- `*solid_joinstyle*`

Refer to the corresponding `~.Line2D`` property for more details:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: `~.CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: `~.JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

label: object  
 linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}  
 linewidth or lw: float  
 marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`  
 markeredgewidth or mec: :mpltype:`color`  
 markeredgewidth or mew: float  
 markerfacecolor or mfc: :mpltype:`color`  
 markerfacecoloralt or mfcalt: :mpltype:`color`  
 markersize or ms: float  
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]  
 mouseover: bool  
 path\_effects: list of `~.AbstractPathEffect`  
 picker: float or callable[[Artist, Event], tuple[bool, dict]]  
 pickradius: float  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 solid\_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}  
 solid\_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}  
 transform: unknown  
 url: str  
 visible: bool  
 xdata: 1D array  
 ydata: 1D array  
 zorder: float

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.errorbar`.

## matplotlib.pyplot.eventplot

```

eventplot(positions: 'ArrayLike | Sequence[ArrayLike]', *, orientation:
 "Literal['horizontal', 'vertical']" = 'horizontal', lineoffsets: 'float |
 Sequence[float]' = 1, linelengths: 'float | Sequence[float]' = 1, linewidths: 'float |
 Sequence[float] | None' = None, colors: 'ColorType | Sequence[ColorType] | None' =
 None, alpha: 'float | Sequence[float] | None' = None, linestyle: 'LineStyleType |
 Sequence[LineStyleType]' = 'solid', data=None, **kwargs) -> 'EventCollection'

```

Plot identical parallel lines at the given positions.

This type of plot is commonly used in neuroscience for representing neural events, where it is usually called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

Parameters

-----

positions : array-like or list of array-like  
A 1D array-like defines the positions of one sequence of events.

Multiple groups of events may be passed as a list of array-likes.  
Each group can be styled independently by passing lists of values to `*lineoffsets*`, `*linelengths*`, `*linewidths*`, `*colors*` and `*linestyles*`.

Note that `*positions*` can be a 2D array, but in practice different event groups usually have different counts so that one will use a list of different-length arrays rather than a 2D array.

orientation : {'horizontal', 'vertical'}, default: 'horizontal'  
The direction of the event sequence:

- 'horizontal': the events are arranged horizontally.  
The indicator lines are vertical.
- 'vertical': the events are arranged vertically.  
The indicator lines are horizontal.

lineoffsets : float or array-like, default: 1  
The offset of the center of the lines from the origin, in the direction orthogonal to `*orientation*`.

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

linelengths : float or array-like, default: 1  
The total height of the lines (i.e. the lines stretches from ```lineoffset - linelength/2``` to ```lineoffset + linelength/2```).

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

linewidths : float or array-like, default: `:rc:`lines.linewidth``  
The line width(s) of the event lines, in points.

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

colors : `:mplttype:`color`` or list of color, default: `:rc:`lines.color``  
The color(s) of the event lines.

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

alpha : float or array-like, default: 1  
The alpha blending value(s), between 0 (transparent) and 1 (opaque).

If `*positions*` is 2D, this can be a sequence with length matching the length of `*positions*`.

linestyles : str or tuple or list of such values, default: 'solid'  
Default is 'solid'. Valid strings are ['solid', 'dashed',

'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form::

(offset, onoffseq),

where *\*onoffseq\** is an even length tuple of on and off ink in points.

If *\*positions\** is 2D, this can be a sequence with length matching the length of *\*positions\**.

*data* : indexable object, optional

If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`:

*\*positions\**, *\*lineoffsets\**, *\*linelengths\**, *\*linewidths\**, *\*colors\**, *\*linestyles\**

**\*\*kwargs**

Other keyword arguments are line collection properties. See `.LineCollection`` for a list of the valid properties.

Returns

-----

list of `.EventCollection``

The `.EventCollection`` that were added.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.eventplot``.

For *\*linelengths\**, *\*linewidths\**, *\*colors\**, *\*alpha\** and *\*linestyles\**, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as *\*positions\**, and each value will be applied to the corresponding row of the array.

Examples

-----

.. plot:: gallery/lines\_bars\_and\_markers/eventplot\_demo.py

## matplotlib.pyplot.figspect

`figspect(arg)`

Calculate the width and height for a figure with a specified aspect ratio.

While the height is taken from `:rc:`figure.figsize``, the width is adjusted to match the desired aspect ratio. Additionally, it is ensured that the width is in the range [4., 16.] and the height is in the range [2., 16.]. If necessary, the default height is adjusted to ensure this.

Parameters

-----

arg : float or 2D array

If a float, this defines the aspect ratio (i.e. the ratio height / width).

In case of an array the aspect ratio is number of rows / number of columns, so that the array could be fitted in the figure undistorted.

Returns

-----

size : (2,) array

The width and height of the figure in inches.

Notes

-----

If you want to create an Axes within the figure, that still preserves the aspect ratio, be sure to create it with equal width and height. See examples below.

Thanks to Fernando Perez for this function.

Examples

-----

Make a figure twice as tall as it is wide::

```
w, h = figaspect(2.)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

Make a figure with the proper aspect for an array::

```
A = rand(5, 3)
w, h = figaspect(A)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

## matplotlib.pyplot.figureimage

```
figureimage(X: 'ArrayLike', xo: 'int' = 0, yo: 'int' = 0, alpha: 'float | None' = None,
norm: 'str | Normalize | None' = None, cmap: 'str | Colormap | None' = None, vmin:
'float | None' = None, vmax: 'float | None' = None, origin: "Literal['upper', 'lower']
| None" = None, resize: 'bool' = False, *, colorizer: 'Colorizer | None' = None,
**kwargs) -> 'FigureImage'
```

Add a non-resampled image to the figure.

The image is attached to the lower or upper left corner depending on \*origin\*.

Parameters

-----

X

The image data. This is an array of one of the following shapes:

- (M, N): an image with scalar data. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

`xo, yo` : int

The `*x*/y*` image offset in pixels.

`alpha` : None or float

The alpha blending value.

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*X*` is RGB(A).

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*X*` is RGB(A).

`vmin, vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

This parameter is ignored if `*X*` is RGB(A).

`origin` : {'upper', 'lower'}, default: `:rc:`image.origin``

Indicates where the [0, 0] index of the array is in the upper left or lower left corner of the Axes.

`resize` : bool

If `*True*`, resize the figure to match the given image size.

`colorizer` : `~matplotlib.colorbar.Colorizer`` or None, default: None

The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

This parameter is ignored if `*X*` is RGB(A).

## Returns

-----  
``matplotlib.image.FigureImage``

## Other Parameters

-----  
`**kwargs`

Additional kwargs are ``Artist`` kwargs passed on to ``FigureImage``.

## Notes

-----  
.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``Figure.figimage``.

`figimage` complements the Axes image (``~matplotlib.axes.Axes.imshow``) which will be resampled to fit the current Axes. If you want a resampled image to fill the entire figure, you can define an ``~matplotlib.axes.Axes`` with extent `[0, 0, 1, 1]`.

## Examples

-----  
::

```
f = plt.figure()
nx = int(f.get_figwidth() * f.dpi)
ny = int(f.get_figheight() * f.dpi)
data = np.random.random((ny, nx))
f.figimage(data)
plt.show()
```

## matplotlib.pyplot.figlegend

```
figlegend(*args, **kwargs) -> 'Legend'
```

Place a legend on the figure.

Call signatures::

```
figlegend()
figlegend(handles, labels)
figlegend(handles=handles)
figlegend(labels)
```

The call signatures correspond to the following different ways to use this method:

**\*\*1. Automatic detection of elements to be shown in the legend\*\***

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the :meth:`~Artist.set\_label` method on the artist::

```
plt.plot([1, 2, 3], label='Inline label')
plt.figlegend()
```

or::

```
line, = plt.plot([1, 2, 3])
line.set_label('Label via method')
plt.figlegend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling `Figure.legend` without any arguments and without setting the labels manually will result in no legend being drawn.

#### **\*\*2. Explicitly listing the artists and labels in the legend\*\***

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively::

```
plt.figlegend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

#### **\*\*3. Explicitly listing the artists in the legend\*\***

This is similar to 2, but the labels are taken from the artists' label properties. Example::

```
line1, = ax1.plot([1, 2, 3], label='label1')
line2, = ax2.plot([1, 2, 3], label='label2')
plt.figlegend(handles=[line1, line2])
```

#### **\*\*4. Labeling existing plot elements\*\***

.. admonition:: Discouraged

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

To make a legend for all artists on all Axes, call this function with an iterable of strings, one for each legend item. For example::

```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot([1, 3, 5], color='blue')
ax2.plot([2, 4, 6], color='red')
plt.figlegend(['the blues', 'the reds'])
```

## Parameters

-----

`handles` : list of ``Artist``, optional

A list of Artists (lines, patches) to be added to the legend.

Use this together with `*labels*`, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

`labels` : list of str, optional

A list of labels to show next to the artists.

Use this together with `*handles*`, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

## Returns

-----

``~matplotlib.legend.Legend``

## Other Parameters

-----

`loc` : str or pair of floats, default: 'upper right'

The location of the legend.

The strings ```'upper left'```, ```'upper right'```, ```'lower left'```, ```'lower right'``` place the legend at the corresponding corner of the figure.

The strings ```'upper center'```, ```'lower center'```, ```'center left'```, ```'center right'``` place the legend at the center of the corresponding edge of the figure.

The string ```'center'``` places the legend at the center of the figure.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in figure coordinates (in which case `*bbox_to_anchor*` will be ignored).

For back-compatibility, ```'center right'``` (but no other location) can also be spelled ```'right'```, and each "string" location can also be given as a numeric value:

=====

Location String Location Code

=====

'best' (Axes only) 0

'upper right' 1

'upper left' 2

'lower left' 3

'lower right' 4

'right' 5

'center left' 6  
'center right' 7  
'lower center' 8  
'upper center' 9  
'center' 10

=====

If a figure is using the constrained layout manager, the string codes of the `*loc*` keyword argument can get better layout behaviour using the prefix 'outside'. There is ambiguity at the corners, so 'outside upper right' will make space for the legend above the rest of the axes in the layout, and 'outside right upper' will make space on the right side of the layout. In addition to the values of `*loc*` listed above, we have 'outside right upper', 'outside right lower', 'outside left upper', and 'outside left lower'. See :ref:`legend\_guide` for more details.

`bbox_to_anchor` : `.BboxBase``, 2-tuple, or 4-tuple of floats  
Box that is used to position the legend in conjunction with `*loc*`. Defaults to ```axes.bbox``` (if called as a method to `.Axes.legend``) or ```figure.bbox``` (if ```figure.legend```). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by `*bbox_transform*`, with the default transform Axes or Figure coordinates, depending on which ```legend``` is called.

If a 4-tuple or `.BboxBase`` is given, then it specifies the bbox ```(x, y, width, height)``` that the legend is placed in.  
To put the legend in the best location in the bottom right quadrant of the Axes (or figure)::

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple ```(x, y)``` places the corner of the legend specified by `*loc*` at `x, y`. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used::

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

`ncols` : int, default: 1  
The number of columns that the legend has.

For backward compatibility, the spelling `*ncol*` is also supported but it is discouraged. If both are given, `*ncols*` takes precedence.

`prop` : None or `~matplotlib.font_manager.FontProperties`` or dict  
The font properties of the legend. If None (default), the current :data:`matplotlib.rcParams` will be used.

`fontsize` : int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}  
The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if `*prop*` is not specified.

`labelcolor` : str or list, default: `:rc:`legend.labelcolor``  
The color of the text in the legend. Either a valid color string (for example, 'red'), or a list of color strings. The `labelcolor` can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

`Labelcolor` can be set globally using `:rc:`legend.labelcolor``. If None, use `:rc:`text.color``.

`numpoints` : int, default: `:rc:`legend.numpoints``  
The number of marker points in the legend when creating a legend entry for a `.Line2D`` (line).

`scatterpoints` : int, default: `:rc:`legend.scatterpoints``  
The number of marker points in the legend when creating a legend entry for a `.PathCollection`` (scatter plot).

`scatteryoffsets` : iterable of floats, default: ```[0.375, 0.5, 0.3125]```  
The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to ```[0.5]```.

`markerscale` : float, default: `:rc:`legend.markerscale``  
The relative size of legend markers compared to the originally drawn ones.

`markerfirst` : bool, default: True  
If `*True*`, legend marker is placed to the left of the legend label.  
If `*False*`, legend marker is placed to the right of the legend label.

`reverse` : bool, default: False  
If `*True*`, the legend labels are displayed in reverse order from the input.  
If `*False*`, the legend labels are displayed in the same order as the input.

.. versionadded:: 3.7

`frameon` : bool, default: `:rc:`legend.frameon``  
Whether the legend should be drawn on a patch (frame).

`fancybox` : bool, default: `:rc:`legend.fancybox``  
Whether round edges should be enabled around the `.FancyBboxPatch`` which makes up the legend's background.

`shadow` : None, bool or dict, default: `:rc:`legend.shadow``  
Whether to draw a shadow behind the legend.  
The shadow can be configured using `.Patch`` keywords.  
Customization via `:rc:`legend.shadow`` is currently not supported.

`framealpha` : float, default: `:rc:`legend.framealpha``  
The alpha transparency of the legend's background.  
If `*shadow*` is activated and `*framealpha*` is ```None```, the default value is ignored.

`facecolor` : "inherit" or color, default: `:rc:`legend.facecolor``  
The legend's background color.

If `inherit`, use `:rc:axes.facecolor`.

`edgecolor` : "inherit" or color, default: `:rc:legend.edgecolor`

The legend's background patch edge color.

If `inherit`, use `:rc:axes.edgecolor`.

`mode` : {"expand", None}

If `*mode*` is set to `"expand"` the legend will be horizontally expanded to fill the Axes area (or `*bbox_to_anchor*` if defines the legend's size).

`bbox_transform` : None or `~matplotlib.transforms.Transform`

The transform for the bounding box (`*bbox_to_anchor*`). For a value of `None` (default) the Axes

`:data:~matplotlib.axes.Axes.transAxes` transform will be used.

`title` : str or None

The legend's title. Default is no title (`None`).

`title_fontproperties` : None or `~matplotlib.font_manager.FontProperties` or dict

The font properties of the legend's title. If None (default), the `*title_fontsize*` argument will be used if present; if `*title_fontsize*` is also None, the current `:rc:legend.title_fontsize` will be used.

`title_fontsize` : int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default:

`:rc:legend.title_fontsize`

The font size of the legend's title.

Note: This cannot be combined with `*title_fontproperties*`. If you want to set the fontsize alongside other font properties, use the `*size*` parameter in `*title_fontproperties*`.

`alignment` : {'center', 'left', 'right'}, default: 'center'

The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

`borderpad` : float, default: `:rc:legend.borderpad`

The fractional whitespace inside the legend border, in font-size units.

`labelspacing` : float, default: `:rc:legend.labelspacing`

The vertical space between the legend entries, in font-size units.

`handlelength` : float, default: `:rc:legend.handlelength`

The length of the legend handles, in font-size units.

`handleheight` : float, default: `:rc:legend.handleheight`

The height of the legend handles, in font-size units.

`handletextpad` : float, default: `:rc:legend.handletextpad`

The pad between the legend handle and text, in font-size units.

`borderaxespad` : float, default: `:rc:legend.borderaxespad`

The pad between the Axes and legend border, in font-size units.

`columnspacing` : float, default: `:rc:legend.columnspacing`

The spacing between columns, in font-size units.

handler\_map : dict or None

The custom dictionary mapping instances or types to a legend handler. This `*handler_map*` updates the default handler map found at ``matplotlib.legend.Legend.get_legend_handler_map``.

draggable : bool, default: False

Whether the legend can be dragged with the mouse.

See Also

-----  
`.Axes.legend`

Notes

-----  
Some artists are not supported by this function. See `:ref:`legend_guide`` for details.

## matplotlib.pyplot.figure\_exists

```
figure_exists(num: 'int | str') -> 'bool'
```

Return whether the figure with the given id exists.

Parameters

-----  
num : int or str  
A figure identifier.

Returns

-----  
bool  
Whether or not a figure with id `*num*` exists.

## matplotlib.pyplot.figtext

```
figtext(x: 'float', y: 'float', s: 'str', fontdict: 'dict[str, Any] | None' = None, **kwargs) -> 'Text'
```

Add text to figure.

Parameters

-----  
x, y : float  
The position to place the text. By default, this is in figure coordinates, floats in [0, 1]. The coordinate system can be changed using the `*transform*` keyword.

s : str  
The text string.

fontdict : dict, optional  
A dictionary to override the default text properties. If not given, the defaults are determined by `:rc:`font.*``. Properties passed as

\*kwargs\* override the corresponding ones given in \*fontdict\*.

## Returns

-----

``~.text.Text``

## Other Parameters

-----

**\*\*kwargs** : ``~matplotlib.text.Text`` properties

Other miscellaneous text parameters.

## Properties:

**agg\_filter**: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

**alpha**: float or None

**animated**: bool

**antialiased**: bool

**backgroundcolor**: `:mpltype:`color``

**bbox**: dict with properties for ``~patches.FancyBboxPatch``

**clip\_box**: unknown

**clip\_on**: unknown

**clip\_path**: unknown

**color** or **c**: `:mpltype:`color``

**figure**: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

**fontfamily** or **family** or **fontname**: {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}

**fontproperties** or **font** or **font\_properties**: ``~font_manager.FontProperties`` or ``str`` or ``pathlib.Path``

**fontsize** or **size**: float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}

**fontstretch** or **stretch**: {a numeric value in range 0-1000, 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'}

**fontstyle** or **style**: {'normal', 'italic', 'oblique'}

**fontvariant** or **variant**: {'normal', 'small-caps'}

**fontweight** or **weight**: {a numeric value in range 0-1000, 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}

**gid**: str

**horizontalalignment** or **ha**: {'left', 'center', 'right'}

**in\_layout**: bool

**label**: object

**linespacing**: float (multiple of font size)

**math\_fontfamily**: str

**mouseover**: bool

**multialignment** or **ma**: {'left', 'right', 'center'}

**parse\_math**: bool

**path\_effects**: list of ``~AbstractPathEffect``

**picker**: None or bool or float or callable

**position**: (float, float)

**rasterized**: bool

**rotation**: float or {'vertical', 'horizontal'}

**rotation\_mode**: {None, 'default', 'anchor'}

**sketch\_params**: (scale: float, length: float, randomness: float)

**snap**: bool or None

**text**: object

**transform**: ``~matplotlib.transforms.Transform``

**transform\_rotates\_text**: bool

**url**: str

**usetex**: bool, default: `:rc:`text.usetex``

verticalalignment or va: {'baseline', 'bottom', 'center', 'center\_baseline', 'top'}  
visible: bool  
wrap: bool  
x: float  
y: float  
zorder: float

See Also

-----

.Axes.text  
.pyplot.text

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.Figure.text``.

## matplotlib.pyplot.figure

```
figure(num: 'int | str | Figure | SubFigure | None' = None, figsize: 'ArrayLike | None' = None, dpi: 'float | None' = None, *, facecolor: 'ColorType | None' = None, edgecolor: 'ColorType | None' = None, frameon: 'bool' = True, FigureClass: 'type[Figure]' = , clear: 'bool' = False, **kwargs) -> 'Figure'
```

Create a new figure, or activate an existing figure.

Parameters

-----

num : int or str or `.Figure`` or `.SubFigure``, optional  
A unique identifier for the figure.

If a figure with that identifier already exists, this figure is made active and returned. An integer refers to the `Figure.number`` attribute, a string refers to the figure label.

If there is no figure with the identifier or `*num`` is not given, a new figure is created, made active and returned. If `*num`` is an int, it will be used for the `Figure.number`` attribute, otherwise, an auto-generated integer value is used (starting at 1 and incremented for each new figure). If `*num`` is a string, the figure label and the window title is set to this value. If num is a `SubFigure``, its parent `Figure`` is activated.

figsize : (float, float), default: `:rc:`figure.figsize``  
Width, height in inches.

dpi : float, default: `:rc:`figure.dpi``  
The resolution of the figure in dots-per-inch.

facecolor : `mpltype:`color``, default: `:rc:`figure.facecolor``  
The background color.

edgecolor : `mpltype:`color``, default: `:rc:`figure.edgecolor``

The border color.

frameon : bool, default: True

If False, suppress drawing the figure frame.

FigureClass : subclass of `~matplotlib.figure.Figure``

If set, an instance of this subclass will be created, rather than a plain `~.Figure``.

clear : bool, default: False

If True and the figure already exists, then it is cleared.

layout : {'constrained', 'compressed', 'tight', 'none', `~.LayoutEngine``, None}, default: None

The layout mechanism for positioning of plot elements to avoid overlapping Axes decorations (labels, ticks, etc). Note that layout managers can measurably slow down figure display.

- 'constrained': The constrained layout solver adjusts Axes sizes to avoid overlapping Axes decorations. Can handle complex plot layouts and colorbars, and is thus recommended.

See :ref:`constrainedlayout\_guide` for examples.

- 'compressed': uses the same algorithm as 'constrained', but removes extra space between fixed-aspect-ratio Axes. Best for simple grids of Axes.

- 'tight': Use the tight layout mechanism. This is a relatively simple algorithm that adjusts the subplot parameters so that decorations do not overlap. See `~.Figure.set_tight_layout`` for further details.

- 'none': Do not use a layout engine.

- A `~.LayoutEngine`` instance. Builtin layout classes are `~.ConstrainedLayoutEngine`` and `~.TightLayoutEngine``, more easily accessible by 'constrained' and 'tight'. Passing an instance allows third parties to provide their own layout engine.

If not given, fall back to using the parameters `*tight_layout*` and `*constrained_layout*`, including their config defaults :rc:`figure.autolayout` and :rc:`figure.constrained\_layout.use`.

**\*\*kwargs**

Additional keyword arguments are passed to the `~.Figure`` constructor.

Returns

-----  
`~matplotlib.figure.Figure``

Notes

-----  
A newly created figure is passed to the `~.FigureCanvasBase.new_manager`` method or the `new_figure_manager`` function provided by the current

backend, which install a canvas and a manager on the figure.

Once this is done, `figure.hooks`` are called, one at a time, on the figure; these hooks allow arbitrary customization of the figure (e.g., attaching callbacks) or of associated elements (e.g., modifying the toolbar). See `:doc:/gallery/user_interfaces/mplcvd`` for an example of toolbar customization.

If you are creating many figures, make sure you explicitly call `.pyplot.close`` on the figures you are not using, because this will enable pyplot to properly clean up the memory.

`~matplotlib.rcParams`` defines the default values, which can be modified in the `matplotlibrc` file.

## matplotlib.pyplot.fill

```
fill(*args, data=None, **kwargs) -> 'list[Polygon]'
```

Plot filled polygons.

### Parameters

`*args` : sequence of `x`, `y`, [`color`]

Each polygon is defined by the lists of `*x*` and `*y*` positions of its nodes, optionally followed by a `*color*` specifier. See `:mod:`matplotlib.colors`` for supported color specifiers. The standard color cycle is used for polygons without a color specifier.

You can plot multiple polygons by providing multiple `*x*`, `*y*`, `*[color]*` groups.

For example, each of the following is legal::

```
ax.fill(x, y) # a polygon with default color
ax.fill(x, y, "b") # a blue polygon
ax.fill(x, y, x2, y2) # two polygons
ax.fill(x, y, "b", x2, y2, "r") # a blue and a red polygon
```

`data` : indexable object, optional

An object with labelled data. If given, provide the label names to plot in `*x*` and `*y*`, e.g.::

```
ax.fill("time", "signal",
data={"time": [0, 1, 2], "signal": [0, 1, 0]})
```

### Returns

list of `~matplotlib.patches.Polygon``

### Other Parameters

`**kwargs` : `~matplotlib.patches.Polygon`` properties

## Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.fill``.

Use :meth:`fill\_between` if you would like to fill the region between two curves.

## matplotlib.pyplot.fill\_between

```
fill_between(x: 'ArrayLike', y1: 'ArrayLike | float', y2: 'ArrayLike | float' = 0,
where: 'Sequence[bool] | None' = None, interpolate: 'bool' = False, step:
'Literal['pre', 'post', 'mid'] | None' = None, *, data=None, **kwargs) ->
'FillBetweenPolyCollection'
```

Fill the area between two horizontal curves.

The curves are defined by the points `(*x*, *y1*)` and `(*x*, *y2*)`. This creates one or multiple polygons describing the filled area.

You may exclude some horizontal sections from filling using `*where*`.

By default, the edges connect the given points directly. Use `*step*` if the filling should be a step function, i.e. constant in between `*x*`.

### Parameters

-----

`x` : array-like

The x coordinates of the nodes defining the curves.

`y1` : array-like or float

The y coordinates of the nodes defining the first curve.

`y2` : array-like or float, default: 0

The y coordinates of the nodes defining the second curve.

`where` : array-like of bool, optional

Define `*where*` to exclude some horizontal regions from being filled.

The filled regions are defined by the coordinates ```x[where]```.

More precisely, fill between ```x[i]``` and ```x[i+1]``` if

```where[i] and where[i+1]```. Note that this definition implies

that an isolated `*True*` value between two `*False*` values in `*where*` will not result in filling. Both sides of the `*True*` position remain unfilled due to the adjacent `*False*` values.

`interpolate` : bool, default: False

This option is only relevant if `*where*` is used and the two curves are crossing each other.

Semantically, `*where*` is often used for `*y1* > *y2*` or

similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the `*x*` array. Such a polygon cannot describe the above semantics close to the intersection. The x-sections containing the intersection are simply clipped.

Setting `*interpolate*` to `*True*` will calculate the actual intersection point and extend the filled region up to this point.

`step` : {'pre', 'post', 'mid'}, optional
Define `*step*` if the filling should be a step function, i.e. constant in between `*x*`. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every `*x*` position, i.e. the interval `[(x[i-1], x[i])]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every `*x*` position, i.e. the interval `[(x[i], x[i+1])]` has the value `y[i]`.
- 'mid': Steps occur half-way between the `*x*` positions.

Returns

``.FillBetweenPolyCollection``
A ``.FillBetweenPolyCollection`` containing the plotted polygons.

Other Parameters

`data` : indexable object, optional
If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*y1*`, `*y2*`, `*where*`

`**kwargs`

All other keyword arguments are passed on to ``.FillBetweenPolyCollection``. They control the ``.Polygon`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
`alpha`: array-like or float or None
`animated`: bool
`antialiased` or `aa` or `antialiaseds`: bool or list of bools
`array`: array-like or None
`capstyle`: ``.CapStyle`` or {'butt', 'projecting', 'round'}
`clim`: (vmin: float, vmax: float)
`clip_box`: `~matplotlib.transforms.BboxBase`` or None
`clip_on`: bool
`clip_path`: Patch or (Path, Transform) or None
`cmap`: ``.Colormap`` or str or None
`color`: `:mpltype:`color`` or list of RGBA tuples
`data`: array-like
`edgecolor` or `ec` or `edgecolors`: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'

facecolor or facecolors or fc: :mpltype:`color` or list of :mpltype:`color`
 figure: ~matplotlib.figure.Figure` or ~matplotlib.figure.SubFigure`
 gid: str
 hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
 hatch_linewidth: unknown
 in_layout: bool
 joinstyle: ~.JoinStyle` or {'miter', 'round', 'bevel'}
 label: object
 linestyle or dashes or linestyles or ls: str or tuple or list thereof
 linewidth or linewidths or lw: float or list of floats
 mouseover: bool
 norm: ~.Normalize` or str or None
 offset_transform or transOffset: ~.Transform`
 offsets: (N, 2) or (2,) array-like
 path_effects: list of ~.AbstractPathEffect`
 paths: list of array-like
 picker: None or bool or float or callable
 pickradius: float
 rasterized: bool
 sizes: ~numpy.ndarray` or None
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 transform: ~matplotlib.transforms.Transform`
 url: str
 urls: list of str or None
 verts: list of array-like
 verts_and_codes: unknown
 visible: bool
 zorder: float

See Also

fill_between : Fill between two sets of y-values.

fill_betweenx : Fill between two sets of x-values.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ~.axes.Axes.fill_between`.

matplotlib.pyplot.fill_betweenx

```

fill_betweenx(y: 'ArrayLike', x1: 'ArrayLike | float', x2: 'ArrayLike | float' = 0,
where: 'Sequence[bool] | None' = None, step: "Literal['pre', 'post', 'mid'] | None" =
None, interpolate: 'bool' = False, *, data=None, **kwargs) ->
'FillBetweenPolyCollection'

```

Fill the area between two vertical curves.

The curves are defined by the points (*y*, *x1*) and (*y*, *x2*). This creates one or multiple polygons describing the filled area.

You may exclude some vertical sections from filling using **where**.

By default, the edges connect the given points directly. Use **step** if the filling should be a step function, i.e. constant in between **y**.

Parameters

y : array-like

The y coordinates of the nodes defining the curves.

x1 : array-like or float

The x coordinates of the nodes defining the first curve.

x2 : array-like or float, default: 0

The x coordinates of the nodes defining the second curve.

where : array-like of bool, optional

Define **where** to exclude some vertical regions from being filled.

The filled regions are defined by the coordinates `y[where]`.

More precisely, fill between `y[i]` and `y[i+1]` if

`where[i]` and `where[i+1]`. Note that this definition implies

that an isolated **True** value between two **False** values in **where**

will not result in filling. Both sides of the **True** position

remain unfilled due to the adjacent **False** values.

interpolate : bool, default: False

This option is only relevant if **where** is used and the two curves are crossing each other.

Semantically, **where** is often used for **x1* > *x2** or

similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the **y** array.

Such a polygon cannot describe the above semantics close to the intersection. The y-sections containing the intersection are simply clipped.

Setting **interpolate** to **True** will calculate the actual intersection point and extend the filled region up to this point.

step : {'pre', 'post', 'mid'}, optional

Define **step** if the filling should be a step function,

i.e. constant in between **y**. The value determines where the step will occur:

- 'pre': The x value is continued constantly to the left from every **y** position, i.e. the interval `(y[i-1], y[i]]` has the value `x[i]`.

- 'post': The y value is continued constantly to the right from every **y** position, i.e. the interval `[y[i], y[i+1))` has the value `x[i]`.

- 'mid': Steps occur half-way between the **y** positions.

Returns

``FillBetweenPolyCollection``

A ``FillBetweenPolyCollection`` containing the plotted polygons.

Other Parameters

data : indexable object, optional

If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

`*y*, *x1*, *x2*, *where*`

`**kwargs`

All other keyword arguments are passed on to

``FillBetweenPolyCollection``. They control the ``Polygon`` properties:

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: array-like or float or None

animated: bool

antialiased or aa or antialiaseds: bool or list of bools

array: array-like or None

capstyle: ``CapStyle`` or {'butt', 'projecting', 'round'}

clim: (vmin: float, vmax: float)

clip_box: ``~matplotlib.transforms.BboxBase`` or None

clip_on: bool

clip_path: Patch or (Path, Transform) or None

cmap: ``Colormap`` or str or None

color: `:mpltype:`color`` or list of RGBA tuples

data: array-like

edgecolor or ec or edgecolors: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'

facecolor or facecolors or fc: `:mpltype:`color`` or list of `:mpltype:`color``

figure: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

gid: str

hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}

hatch_linewidth: unknown

in_layout: bool

joinstyle: ``JoinStyle`` or {'miter', 'round', 'bevel'}

label: object

linestyle or dashes or linestyles or ls: str or tuple or list thereof

linewidth or linewidths or lw: float or list of floats

mouseover: bool

norm: ``Normalize`` or str or None

offset_transform or transOffset: ``Transform``

offsets: (N, 2) or (2,) array-like

path_effects: list of ``AbstractPathEffect``

paths: list of array-like

picker: None or bool or float or callable

pickradius: float

rasterized: bool

sizes: ``numpy.ndarray`` or None

sketch_params: (scale: float, length: float, randomness: float)

snap: bool or None

transform: ``~matplotlib.transforms.Transform``

url: str

urls: list of str or None
verts: list of array-like
verts_and_codes: unknown
visible: bool
zorder: float

See Also

fill_between : Fill between two sets of y-values.
fill_betweenx : Fill between two sets of x-values.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.fill_betweenx``.

matplotlib.pyplot.findobj

```
findobj(o: 'Artist | None' = None, match: 'Callable[[Artist], bool] | type[Artist] | None' = None, include_self: 'bool' = True) -> 'list[Artist]'
```

Find artist objects.

Recursively find all `.Artist`` instances contained in the artist.

Parameters

match

A filter criterion for the matches. This can be

- `*None*`: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool``. The result will only contain artists for which the function returns `*True*`.
- A class instance: e.g., `.Line2D``. The result will only contain artists of this class or its subclasses (`isinstance`` check).

include_self : bool

Include `*self*` in the list to be checked for a match.

Returns

list of `.Artist``

matplotlib.pyplot.flag

```
flag() -> 'None'
```

Set the colormap to 'flag'.

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)`` for more information.

matplotlib.pyplot.gca

```
gca() -> 'Axes'
```

Get the current Axes.

If there is currently no Axes on this Figure, a new one is created using `.Figure.add_subplot`. (To test whether there is currently an Axes on a Figure, check whether `figure.axes` is empty. To test whether there is currently a Figure on the pyplot figure stack, check whether `.pyplot.get_fignums()` is empty.)

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.Figure.gca`.

matplotlib.pyplot.gcf

```
gcf() -> 'Figure'
```

Get the current figure.

If there is currently no figure on the pyplot figure stack, a new one is created using `~.pyplot.figure()`. (To test whether there is currently a figure on the pyplot figure stack, check whether `~.pyplot.get_fignums()` is empty.)

matplotlib.pyplot.gci

```
gci() -> 'ColorizingArtist | None'
```

Get the current colorable artist.

Specifically, returns the current `.ScalarMappable` instance (`.Image` created by `imshow` or `figimage`, `.Collection` created by `pcolor` or `scatter`, etc.), or `*None*` if no such instance has been defined.

The current image is an attribute of the current Axes, or the nearest earlier Axes in the current figure that contains an image.

Notes

Historically, the only colorable artists were images; hence the name `gci` (get current image).

matplotlib.pyplot.get

```
get(obj, *args, **kwargs)
```

Return the value of an `.Artist`'s `*property*`, or print all of them.

Parameters

obj : `~matplotlib.artist.Artist``
The queried artist; e.g., a `~.Line2D``, a `~.Text``, or an `~.axes.Axes``.

property : str or None, default: None
If `*property*` is 'somename', this function returns
```obj.get_somename()```.

If it's None (or unset), it *\*prints\** all gettable properties from  
`*obj*`. Many properties have aliases for shorter typing, e.g. 'lw' is  
an alias for 'linewidth'. In the output, aliases and full property  
names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

See Also

-----  
`setp`

Notes

-----

.. note::

This is equivalent to `~matplotlib.artist.getp``.

## matplotlib.pyplot.get\_backend

```
get_backend(*, auto_select=True)
```

Return the name of the current backend.

Parameters

-----

auto\_select : bool, default: True

Whether to trigger backend resolution if no backend has been  
selected so far. If True, this ensures that a valid backend  
is returned. If False, this returns None if no backend has been  
selected so far.

.. versionadded:: 3.10

.. admonition:: Provisional

The `*auto_select*` flag is provisional. It may be changed or removed  
without prior warning.

See Also

-----

`matplotlib.use`

## matplotlib.pyplot.get\_cmap

```
get_cmap(name: 'Colormap | str | None' = None, lut: 'int | None' = None) -> 'Colormap'
```

Get a colormap instance, defaulting to rc values if *\*name\** is None.

### Parameters

-----

name : ~matplotlib.colors.Colormap` or str or None, default: None

If a ~matplotlib.colors.Colormap` instance, it will be returned. Otherwise, the name of a colormap known to Matplotlib, which will be resampled by *\*lut\**. The default, None, means :rc:`image.cmap`.

lut : int or None, default: None

If *\*name\** is not already a Colormap instance and *\*lut\** is not None, the colormap will be resampled to have *\*lut\** entries in the lookup table.

### Returns

-----

Colormap

## matplotlib.pyplot.get\_current\_fig\_manager

```
get_current_fig_manager() -> 'FigureManagerBase | None'
```

Return the figure manager of the current figure.

The figure manager is a container for the actual backend-dependent window that displays the figure on screen.

If no current figure exists, a new one is created, and its figure manager is returned.

### Returns

-----

~matplotlib.figure.FigureManagerBase` or backend-dependent subclass thereof

## matplotlib.pyplot.get\_figlabels

```
get_figlabels() -> 'list[Any]'
```

Return a list of existing figure labels.

## matplotlib.pyplot.get\_fignums

```
get_fignums() -> 'list[int]'
```

Return a list of existing figure numbers.

## matplotlib.pyplot.get\_plot\_commands

```
get_plot_commands() -> 'list[str]'
```

[\*Deprecated\*] Get a sorted list of all of the plotting commands.

### Notes

-----  
.. deprecated:: 3.7

## matplotlib.pyplot.get\_scale\_names

```
get_scale_names()
```

Return the names of the available scales.

## matplotlib.pyplot.getp

```
getp(obj, *args, **kwargs)
```

Return the value of an `Artist`'s `property`, or print all of them.

Parameters

-----

`obj` : `~matplotlib.artist.Artist`

The queried artist; e.g., a `Line2D`, a `Text`, or an `axes.Axes`.

`property` : str or None, default: None

If `*property*` is 'somename', this function returns

`obj.get_somename()`.

If it's None (or unset), it `*prints*` all gettable properties from `*obj*`. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

See Also

-----

setp

Notes

-----

.. note::

This is equivalent to `matplotlib.artist.getp`.

## matplotlib.pyplot.ginput

```
ginput(n: 'int' = 1, timeout: 'float' = 30, show_clicks: 'bool' = True, mouse_add: 'MouseButton' = , mouse_pop: 'MouseButton' = , mouse_stop: 'MouseButton' =) -> 'list[tuple[int, int]]'
```

Blocking call to interact with a figure.

Wait until the user clicks `*n*` times on the figure, and return the

coordinates of each click in a list.

There are three possible interactions:

- Add a point.
- Remove the most recently added point.
- Stop the interaction and return the points added so far.

The actions are assigned to mouse buttons via the arguments `*mouse_add*`, `*mouse_pop*` and `*mouse_stop*`.

#### Parameters

-----

`n` : int, default: 1

Number of mouse clicks to accumulate. If negative, accumulate clicks until the input is terminated manually.

`timeout` : float, default: 30 seconds

Number of seconds to wait before timing out. If zero or negative will never time out.

`show_clicks` : bool, default: True

If True, show a red cross at the location of each click.

`mouse_add` : ``MouseButton`` or None, default: ``MouseButton.LEFT``  
Mouse button used to add points.

`mouse_pop` : ``MouseButton`` or None, default: ``MouseButton.RIGHT``  
Mouse button used to remove the most recently added point.

`mouse_stop` : ``MouseButton`` or None, default: ``MouseButton.MIDDLE``  
Mouse button used to stop input.

#### Returns

-----

list of tuples

A list of the clicked (x, y) coordinates.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``Figure.ginput``.

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right-clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

## matplotlib.pyplot.gray

```
gray() -> 'None'
```

Set the colormap to 'gray'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.grid

```
grid(visible: 'bool | None' = None, which: "Literal['major', 'minor', 'both']" = 'major', axis: "Literal['both', 'x', 'y']" = 'both', **kwargs) -> 'None'
```

Configure the grid lines.

### Parameters

visible : bool or None, optional

Whether to show the grid lines. If any *\*kwargs\** are supplied, it is assumed you want the grid on and *\*visible\** will be set to True.

If *\*visible\** is *\*None\** and there are no *\*kwargs\**, this toggles the visibility of the lines.

which : {'major', 'minor', 'both'}, optional  
The grid lines to apply the changes on.

axis : {'both', 'x', 'y'}, optional  
The axis to apply the changes on.

*\*\*kwargs* : `~matplotlib.lines.Line2D`` properties  
Define the line properties of the grid, e.g.::

```
grid(color='r', linestyle='-', linewidth=2)
```

Valid keyword arguments are:

### Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None

animated: bool

antialiased or aa: bool

clip\_box: `~matplotlib.transforms.BboxBase`` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

color or c: `:mpltype:`color``

dash\_capstyle: `~.CapStyle`` or {'butt', 'projecting', 'round'}

dash\_joinstyle: `~.JoinStyle`` or {'miter', 'round', 'bevel'}

dashes: sequence of floats (on/off ink in points) or (None, None)

data: (2, N) array or two 1D arrays

drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}

gapcolor: `:mpltype:`color`` or None

gid: str

in\_layout: bool

label: object

linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

linewidth or lw: float

marker: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``

markeredgecolor or mec: `:mpltype:`color``

markeredgewidth or mew: float

markerfacecolor or mfc: :mpltype:`color`  
 markerfacecoloralt or mfcalt: :mpltype:`color`  
 markersize or ms: float  
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]  
 mouseover: bool  
 path\_effects: list of ``AbstractPathEffect``  
 picker: float or callable[[Artist, Event], tuple[bool, dict]]  
 pickradius: float  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 solid\_capstyle: ``CapStyle`` or {'butt', 'projecting', 'round'}  
 solid\_joinstyle: ``JoinStyle`` or {'miter', 'round', 'bevel'}  
 transform: unknown  
 url: str  
 visible: bool  
 xdata: 1D array  
 ydata: 1D array  
 zorder: float

## Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``axes.Axes.grid``.

The axis is drawn as a unit, so the effective zorder for drawing the grid is determined by the zorder of each axis, not by the zorder of the ``Line2D`` objects comprising the grid. Therefore, to set grid zorder, use ``set_axisbelow`` or, for more control, call the ``~.Artist.set_zorder`` method of each axis.

## matplotlib.pyplot.hexbin

```

hexbin(x: 'ArrayLike', y: 'ArrayLike', C: 'ArrayLike | None' = None, *, gridsize: 'int
| tuple[int, int]' = 100, bins: "Literal['log'] | int | Sequence[float] | None" =
None, xscale: "Literal['linear', 'log']" = 'linear', yscale: "Literal['linear',
'log']" = 'linear', extent: 'tuple[float, float, float, float] | None' = None, cmap:
'str | Colormap | None' = None, norm: 'str | Normalize | None' = None, vmin: 'float |
None' = None, vmax: 'float | None' = None, alpha: 'float | None' = None, linewidths:
'float | None' = None, edgecolors: "Literal['face', 'none'] | ColorType" = 'face',
reduce_C_function: 'Callable[[np.ndarray | list[float]], float]' = , mincnt: 'int |
None' = None, marginals: 'bool' = False, colorizer: 'Colorizer | None' = None,
data=None, **kwargs) -> 'PolyCollection'

```

Make a 2D hexagonal binning plot of points `*x*`, `*y*`.

If `*C*` is `*None*`, the value of the hexagon is determined by the number of points in the hexagon. Otherwise, `*C*` specifies values at the coordinate `(x[i], y[i])`. For each hexagon, these values are reduced using `*reduce_C_function*`.

## Parameters

-----

`x`, `y` : array-like

The data positions. *\*x\** and *\*y\** must be of the same length.

*C* : array-like, optional

If given, these values are accumulated in the bins. Otherwise, every point has a value of 1. Must be of the same length as *\*x\** and *\*y\**.

*gridsize* : int or (int, int), default: 100

If a single int, the number of hexagons in the *\*x\**-direction.

The number of hexagons in the *\*y\**-direction is chosen such that the hexagons are approximately regular.

Alternatively, if a tuple (*\*nx\**, *\*ny\**), the number of hexagons in the *\*x\**-direction and the *\*y\**-direction. In the *\*y\**-direction, counting is done along vertically aligned hexagons, not along the zig-zag chains of hexagons; see the following illustration.

.. plot::

```
import numpy
import matplotlib.pyplot as plt

np.random.seed(19680801)
n= 300
x = np.random.standard_normal(n)
y = np.random.standard_normal(n)

fig, ax = plt.subplots(figsize=(4, 4))
h = ax.hexbin(x, y, gridsize=(5, 3))
hx, hy = h.get_offsets().T
ax.plot(hx[24::3], hy[24::3], 'ro-')
ax.plot(hx[-3:], hy[-3:], 'ro-')
ax.set_title('gridsize=(5, 3)')
ax.axis('off')
```

To get approximately regular hexagons, choose

$n_x = \sqrt{3}, n_y$ .

*bins* : 'log' or int or sequence, default: None

Discretization of the hexagon values.

- If *\*None\**, no binning is applied; the color of each hexagon directly corresponds to its count value.

- If 'log', use a logarithmic scale for the colormap.

Internally,  $\log_{10}(i+1)$  is used to determine the hexagon color. This is equivalent to `norm=LogNorm()`.

- If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.

- If a sequence of values, the values of the lower bound of the bins to be used.

*xscale* : {'linear', 'log'}, default: 'linear'

Use a linear or log10 scale on the horizontal axis.

`yscale` : {'linear', 'log'}, default: 'linear'  
Use a linear or log10 scale on the vertical axis.

`mincnt` : int >= 0, default: \*None\*  
If not \*None\*, only display cells with at least \*mincnt\* number of points in the cell.

`marginals` : bool, default: \*False\*  
If `marginals` is \*True\*, plot the marginal density as colormapped rectangles along the bottom of the x-axis and left of the y-axis.

`extent` : 4-tuple of float, default: \*None\*  
The limits of the bins (`xmin`, `xmax`, `ymin`, `ymax`).  
The default assigns the limits based on \*gridsize\*, \*x\*, \*y\*, \*xscale\* and \*yscale\*.

If \*xscale\* or \*yscale\* is set to 'log', the limits are expected to be the exponent for a power of 10. E.g. for x-limits of 1 and 50 in 'linear' scale and y-limits of 10 and 1000 in 'log' scale, enter (1, 50, 1, 3).

#### Returns

-----  
`~matplotlib.collections.PolyCollection`  
A ~.PolyCollection` defining the hexagonal bins.`

- `~.PolyCollection.get_offsets`` contains a Mx2 array containing the x, y positions of the M hexagon centers in data coordinates.  
- `~.PolyCollection.get_array`` contains the values of the M hexagons.

If \*marginals\* is \*True\*, horizontal bar and vertical bar (both `PolyCollections`) will be attached to the return collection as attributes \*hbar\* and \*vbar\*.

#### Other Parameters

-----  
`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:~image.cmap``  
The `Colormap` instance or registered colormap name used to map scalar data to colors.

`norm` : str or `~matplotlib.colors.Normalize``, optional  
The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using \*cmap\*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `~.Normalize`` or one of its subclasses (see :ref:`colormapnorms`).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `~.Normalize`` subclass is dynamically generated and instantiated.

`vmin, vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/*vmax*` when a `*norm*` instance is given (but using a ``str`` `*norm*` name together with `*vmin*/*vmax*` is acceptable).

`alpha` : float between 0 and 1, optional

The alpha blending value, between 0 (transparent) and 1 (opaque).

`linewidths` : float, default: `*None*`

If `*None*`, defaults to `:rc:`patch.linewidth``.

`edgecolors` : `{'face', 'none', *None*}` or color, default: `'face'`

The color of the hexagon edges. Possible values are:

- `'face'`: Draw the edges in the same color as the fill color.
- `'none'`: No edges are drawn. This can sometimes lead to unsightly unpainted pixels between the hexagons.
- `*None*`: Draw outlines in the default color.
- An explicit color.

`reduce_C_function` : callable, default: ``numpy.mean``

The function to aggregate `*C*` within the bins. It is ignored if `*C*` is not given. This must have the signature::

```
def reduce_C_function(C: array) -> float
```

Commonly used functions are:

- ``numpy.mean``: average of the points
- ``numpy.sum``: integral of the point values
- ``numpy.amax``: value taken from the largest point

By default will only reduce cells with at least 1 point because some reduction functions (such as ``numpy.amax``) will error/warn with empty input. Changing `*mincnt*` will adjust the cutoff, and if set to 0 will pass empty input to the reduction function.

`colorizer` : ``~matplotlib.colorbar.Colorizer`` or `None`, default: `None`

The Colorizer object used to map color to data. If `None`, a Colorizer object is created from a `*norm*` and `*cmap*`.

`data` : indexable object, optional

If given, the following parameters also accept a string ``s``, which is interpreted as ``data[s]`` if ``s`` is a key in ``data``:

`*x*`, `*y*`, `*C*`

`**kwargs` : ``~matplotlib.collections.PolyCollection`` properties

All other keyword arguments are passed on to ``PolyCollection``:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array

and two offsets from the bottom left corner of the image

alpha: array-like or float or None

animated: bool

antialiased or aa or antialiaseds: bool or list of bools

array: array-like or None

capstyle: ``~matplotlib.transforms.BboxBase`` or `{'butt', 'projecting', 'round'}`

clim: (vmin: float, vmax: float)

clip\_box: ``~matplotlib.transforms.BboxBase`` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

cmap: ``~matplotlib.colors.Colormap`` or str or None

color: :mpltype:``color`` or list of RGBA tuples

edgecolor or ec or edgecolors: :mpltype:``color`` or list of :mpltype:``color`` or 'face'

facecolor or facecolors or fc: :mpltype:``color`` or list of :mpltype:``color``

figure: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

gid: str

hatch: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`

hatch\_linewidth: unknown

in\_layout: bool

joinstyle: ``~matplotlib.transforms.Transform`` or `{'miter', 'round', 'bevel'}`

label: object

linestyle or dashes or linestyles or ls: str or tuple or list thereof

linewidth or linewidths or lw: float or list of floats

mouseover: bool

norm: ``~matplotlib.colors.Normalize`` or str or None

offset\_transform or transOffset: ``~matplotlib.transforms.Transform``

offsets: (N, 2) or (2,) array-like

path\_effects: list of ``~matplotlib.transforms.Transform``

paths: list of array-like

picker: None or bool or float or callable

pickradius: float

rasterized: bool

sizes: ``numpy.ndarray`` or None

sketch\_params: (scale: float, length: float, randomness: float)

snap: bool or None

transform: ``~matplotlib.transforms.Transform``

url: str

urls: list of str or None

verts: list of array-like

verts\_and\_codes: unknown

visible: bool

zorder: float

#### See Also

-----

hist2d : 2D histogram rectangular bins

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``~matplotlib.axes.Axes.hexbin``.

```
hist(x: 'ArrayLike | Sequence[ArrayLike]', bins: 'int | Sequence[float] | str | None'
 = None, *, range: 'tuple[float, float] | None' = None, density: 'bool' = False,
weights: 'ArrayLike | None' = None, cumulative: 'bool | float' = False, bottom:
'ArrayLike | float | None' = None, histtype: "Literal['bar', 'barstacked', 'step',
'stepfilled']" = 'bar', align: "Literal['left', 'mid', 'right']" = 'mid', orientation:
"Literal['vertical', 'horizontal']" = 'vertical', rwidth: 'float | None' = None, log:
'bool' = False, color: 'ColorType | Sequence[ColorType] | None' = None, label: 'str |
Sequence[str] | None' = None, stacked: 'bool' = False, data=None, **kwargs) ->
'tuple[np.ndarray | list[np.ndarray], np.ndarray, BarContainer | Polygon |
list[BarContainer | Polygon]]'
```

Compute and plot a histogram.

This method uses `numpy.histogram` to bin the data in `*x*` and count the number of values in each bin, then draws the distribution either as a `.BarContainer` or `.Polygon`. The `*bins*`, `*range*`, `*density*`, and `*weights*` parameters are forwarded to `numpy.histogram`.

If the data has already been binned and counted, use `~.bar` or `~.stairs` to plot the distribution::

```
counts, bins = np.histogram(x)
plt.stairs(counts, bins)
```

Alternatively, plot pre-computed bins and counts using `hist()` by treating each bin as a single point with a weight equal to its count::

```
plt.hist(bins[:-1], bins, weights=counts)
```

The data input `*x*` can be a singular array, a list of datasets of potentially different lengths (`[*x0*, *x1*, ...]`), or a 2D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form. If the input is an array, then the return value is a tuple (`*n*`, `*bins*`, `*patches*`); if the input is a sequence of arrays, then the return value is a tuple (`[*n0*, *n1*, ...]`, `*bins*`, `[*patches0*, *patches1*, ...]`).

Masked arrays are not supported.

#### Parameters

`x` : (n,) array or sequence of (n,) arrays  
Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.

`bins` : int or sequence or str, default: `:rc:hist.bins`  
If `*bins*` is an integer, it defines the number of equal-width bins in the range.

If `*bins*` is a sequence, it defines the bin edges, including the left edge of the first bin and the right edge of the last bin; in this case, bins may be unequally spaced. All but the last (righthand-most) bin is half-open. In other words, if `*bins*` is::

```
[1, 2, 3, 4]
```

then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

If *bins* is a string, it is one of the binning strategies supported by `numpy.histogram_bin_edges`: 'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'.

*range* : tuple or None, default: None

The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, *range* is `(x.min(), x.max())`.

Range has no effect if *bins* is a sequence.

If *bins* is a sequence or *range* is specified, autoscaling is based on the specified bin range instead of the range of *x*.

*density* : bool, default: False

If `True`, draw and return a probability density: each bin will display the bin's raw count divided by the total number of counts *and* the bin width

`(`density = counts / (sum(counts) * np.diff(bins))`),`  
so that the area under the histogram integrates to 1  
`(`np.sum(density * np.diff(bins)) == 1`).`

If *stacked* is also `True`, the sum of the histograms is normalized to 1.

*weights* : (n,) array-like or None, default: None

An array of weights, of the same shape as *x*. Each value in *x* only contributes its associated weight towards the bin count (instead of 1). If *density* is `True`, the weights are normalized, so that the integral of the density over the range remains 1.

*cumulative* : bool or -1, default: False

If `True`, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints.

If *density* is also `True` then the histogram is normalized such that the last bin equals 1.

If *cumulative* is a number less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if *density* is also `True`, then the histogram is normalized such that the first bin equals 1.

*bottom* : array-like or float, default: 0

Location of the bottom of each bin, i.e. bins are drawn from `bottom` to `bottom + hist(x, bins)`. If a scalar, the bottom of each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

histtype : {'bar', 'barstacked', 'step', 'stepfilled'}, default: 'bar'  
The type of histogram to draw.

- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

align : {'left', 'mid', 'right'}, default: 'mid'  
The horizontal alignment of the histogram bars.

- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

orientation : {'vertical', 'horizontal'}, default: 'vertical'  
If 'horizontal', `~.Axes.barh`` will be used for bar-type histograms and the `*bottom*` kwarg will be the left edges.

rwidth : float or None, default: None  
The relative width of the bars as a fraction of the bin width. If ```None```, automatically compute the width.

Ignored if `*histtype*` is 'step' or 'stepfilled'.

log : bool, default: False  
If ```True```, the histogram axis will be set to a log scale.

color : :mpltype:`color` or list of :mpltype:`color` or None, default: None  
Color or sequence of colors, one per dataset. Default (```None```) uses the standard line color sequence.

label : str or list of str, optional  
String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that `~.Axes.legend`` will work as expected.

stacked : bool, default: False  
If ```True```, multiple data are stacked on top of each other. If ```False``` multiple data are arranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

## Returns

-----  
n : array or list of arrays  
The values of the histogram bins. See `*density*` and `*weights*` for a description of the possible semantics. If input `*x*` is an array, then this is an array of length `*nbins*`. If input is a sequence of arrays ```[data1, data2, ...]```, then this is a list of arrays with the values of the histograms for each of the arrays in the same order. The dtype of the array `*n*` (or of its element arrays) will always be float even if no weighting or normalization is used.

bins : array

The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

patches : `~.BarContainer`` or list of a single `~.Polygon`` or list of such objects  
Container of individual artists used to create the histogram  
or list of such containers if there are multiple input datasets.

#### Other Parameters

-----  
data : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*weights*`

`**kwargs`

`~matplotlib.patches.Patch`` properties. The following properties additionally accept a sequence of values corresponding to the datasets in `*x*`:

`*edgecolor*`, `*facecolor*`, `*linewidth*`, `*linestyle*`, `*hatch*`.

.. versionadded:: 3.10

Allowing sequences of values in above listed Patch properties.

#### See Also

-----  
hist2d : 2D histogram with rectangular bins

hexbin : 2D histogram with hexagonal bins

stairs : Plot a pre-computed histogram

bar : Plot a pre-computed histogram

#### Notes

-----  
.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.hist``.

For large numbers of bins (>1000), plotting can be significantly accelerated by using `~.Axes.stairs`` to plot a pre-computed histogram (`~plt.stairs(*np.histogram(data))```), or by setting `*histtype*` to `'step'` or `'stepfilled'` rather than `'bar'` or `'barstacked'`.

## matplotlib.pyplot.hist2d

```
hist2d(x: 'ArrayLike', y: 'ArrayLike', bins: 'None | int | tuple[int, int] | ArrayLike | tuple[ArrayLike, ArrayLike]' = 10, *, range: 'ArrayLike | None' = None, density: 'bool' = False, weights: 'ArrayLike | None' = None, cmin: 'float | None' = None, cmax: 'float | None' = None, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray, np.ndarray, QuadMesh]'
```

Make a 2D histogram plot.

## Parameters

-----

x, y : array-like, shape (n, )

Input values

bins : None or int or [int, int] or array-like or [array, array]

The bin specification:

- If int, the number of bins for the two dimensions  
(`nx = ny = bins`).
- If `[int, int]`, the number of bins in each dimension  
(`nx, ny = bins`).
- If array-like, the bin edges for the two dimensions  
(`x_edges = y_edges = bins`).
- If `[array, array]`, the bin edges in each dimension  
(`x_edges, y_edges = bins`).

The default value is 10.

range : array-like shape(2, 2), optional

The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): `[[xmin, xmax], [ymin, ymax]]`. All values outside of this range will be considered outliers and not tallied in the histogram.

density : bool, default: False

Normalize histogram. See the documentation for the `*density*` parameter of `~.Axes.hist` for more details.

weights : array-like, shape (n, ), optional

An array of values `w_i` weighing each sample `(x_i, y_i)`.

cmin, cmax : float, default: None

All bins that has count less than `*cmin*` or more than `*cmax*` will not be displayed (set to NaN before passing to `~.Axes.pcolormesh`) and these count values in the return value count histogram will also be set to nan upon return.

## Returns

-----

h : 2D array

The bi-dimensional histogram of samples x and y. Values in x are histogrammed along the first dimension and values in y are histogrammed along the second dimension.

xedges : 1D array

The bin edges along the x-axis.

yedges : 1D array

The bin edges along the y-axis.

image : `~.matplotlib.collections.QuadMesh`

## Other Parameters

-----

cmap : str or `~matplotlib.colors.Colormap`, default: `:rc:~image.cmap`

The Colormap instance or registered colormap name used to map scalar data

to colors.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `~matplotlib.colors.Normalize`` or one of its subclasses (see :ref:`colormapnorms`).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `~matplotlib.colors.Normalize`` subclass is dynamically generated and instantiated.

`vmin, vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

`colorizer` : `~matplotlib.colorbar.Colorizer`` or None, default: None

The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

`alpha` : ``0 <= scalar <= 1`` or ``None``, optional

The alpha blending value.

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*y*`, `*weights*`

`**kwargs`

Additional parameters are passed along to the `~.Axes.pcolormesh`` method and `~matplotlib.collections.QuadMesh`` constructor.

See Also

-----

`hist` : 1D histogram plotting

`hexbin` : 2D histogram with hexagonal bins

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.hist2d``.

- Currently ```hist2d``` calculates its own axis limits, and any limits previously set are ignored.

- Rendering the histogram with a logarithmic color scale is accomplished by passing a `.colors.LogNorm` instance to the `*norm*` keyword argument. Likewise, power-law normalization (similar in effect to gamma correction) can be accomplished with `.colors.PowerNorm`.

## matplotlib.pyplot.hlines

```
hlines(y: 'float | ArrayLike', xmin: 'float | ArrayLike', xmax: 'float | ArrayLike',
 colors: 'ColorType | Sequence[ColorType] | None' = None, linestyle: 'LineStyleType' =
 'solid', *, label: 'str' = '', data=None, **kwargs) -> 'LineCollection'
```

Plot horizontal lines at each `*y*` from `*xmin*` to `*xmax*`.

### Parameters

-----

`y` : float or array-like  
y-indexes where to plot the lines.

`xmin, xmax` : float or array-like  
Respective beginning and end of each line. If scalars are provided, all lines will have the same length.

`colors` : `:mpltype:color` or list of color , default: `:rc:lines.color`

`linestyle` : {'solid', 'dashed', 'dashdot', 'dotted'}, default: 'solid'

`label` : str, default: ''

### Returns

-----

`~matplotlib.collections.LineCollection`

### Other Parameters

-----

`data` : indexable object, optional  
If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*y*, *xmin*, *xmax*, *colors*`  
`**kwargs` : `~matplotlib.collections.LineCollection` properties.

### See Also

-----

`vlines` : vertical lines  
`axhline` : horizontal line across the Axes

### Notes

-----

.. note::

This is the `:ref:~pyplot.wrapper <pyplot_interface>` for `~axes.Axes.hlines`.

## matplotlib.pyplot.hot

```
hot() -> 'None'
```

Set the colormap to 'hot'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

## matplotlib.pyplot.hsv

```
hsv() -> 'None'
```

Set the colormap to 'hsv'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

## matplotlib.pyplot.imread

```
imread(fname: 'str | pathlib.Path | BinaryIO', format: 'str | None' = None) -> 'np.ndarray'
```

Read an image from a file into an array.

.. note::

This function exists for historical reasons. It is recommended to use ``PIL.Image.open`` instead for loading images.

### Parameters

-----

fname : str or file-like

The image file to read: a filename, a URL or a file-like object opened in read-binary mode.

Passing a URL is deprecated. Please open the URL for reading and pass the result to Pillow, e.g. with ``np.array(PIL.Image.open(urllib.request.urlopen(url)))``.

format : str, optional

The image file format assumed for reading the data. The image is loaded as a PNG file if \*format\* is set to "png", if \*fname\* is a path or opened file with a ".png" extension, or if it is a URL. In all other cases, \*format\* is ignored and the format is auto-detected by ``PIL.Image.open``.

### Returns

-----

numpy.array

The image data. The returned array has shape

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

PNG images are returned as float arrays (0-1). All other formats are

returned as int arrays, with a bit depth determined by the file's contents.

#### Notes

-----

.. note::

This is equivalent to ``matplotlib.image.imread``.

## matplotlib.pyplot.imsave

```
imsave(fname: 'str | os.PathLike | BinaryIO', arr: 'ArrayLike', **kwargs) -> 'None'
```

Colormap and save an array as an image file.

RGB(A) images are passed through. Single channel images will be colormapped according to `*cmap*` and `*norm*`.

.. note::

If you want to save a single channel image as gray scale please use an image I/O library (such as pillow, tiff file, or imageio) directly.

#### Parameters

-----

`fname` : str or path-like or file-like

A path or a file-like object to store the image in.

If `*format*` is not set, then the output format is inferred from the extension of `*fname*`, if any, and from `:rc:`savefig.format`` otherwise.

If `*format*` is set, it determines the output format.

`arr` : array-like

The image data. Accepts NumPy arrays or sequences

(e.g., lists or tuples). The shape can be one of

MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).

`vmin`, `vmax` : float, optional

`*vmin*` and `*vmax*` set the color scaling for the image by fixing the values that map to the colormap color limits. If either `*vmin*` or `*vmax*` is None, that limit is determined from the `*arr*` min/max value.

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

A Colormap instance or registered colormap name. The colormap maps scalar data to colors. It is ignored for RGB(A) data.

`format` : str, optional

The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under `*fname*`.

`origin` : {'upper', 'lower'}, default: `:rc:`image.origin``

Indicates whether the `((0, 0))`` index of the array is in the upper left or lower left corner of the Axes.

`dpi` : float

The DPI to store in the metadata of the file. This does not affect the resolution of the output image. Depending on file format, this may be rounded to the nearest integer.

`metadata` : dict, optional

Metadata in the image file. The supported keys depend on the output format, see the documentation of the respective backends for more information.

Currently only supported for "png", "pdf", "ps", "eps", and "svg".

`pil_kwargs` : dict, optional

Keyword arguments passed to `PIL.Image.Image.save``. If the 'pnginfo' key is present, it completely overrides `*metadata*`, including the default 'Software' key.

Notes

-----

.. note::

This is equivalent to ``matplotlib.image.imsave``.

## matplotlib.pyplot.imshow

```
imshow(X: 'ArrayLike | PIL.Image.Image', cmap: 'str | Colormap | None' = None, norm:
'str | Normalize | None' = None, *, aspect: "Literal['equal', 'auto'] | float | None"
= None, interpolation: 'str | None' = None, alpha: 'float | ArrayLike | None' = None,
vmin: 'float | None' = None, vmax: 'float | None' = None, colorizer: 'Colorizer |
None' = None, origin: "Literal['upper', 'lower'] | None" = None, extent: 'tuple[float,
float, float, float] | None' = None, interpolation_stage: "Literal['data', 'rgba',
'auto'] | None" = None, filternorm: 'bool' = True, filterrad: 'float' = 4.0, resample:
'bool | None' = None, url: 'str | None' = None, data=None, **kwargs) -> 'AxesImage'
```

Display data as an image, i.e., on a 2D regular raster.

The input may either be actual RGB(A) data, or 2D scalar data, which will be rendered as a pseudocolor image. For displaying a grayscale image, set up the colormapping using the parameters ``cmap='gray', vmin=0, vmax=255``.

The number of pixels used to render an image is set by the Axes size and the figure `*dpi*`. This can lead to aliasing artifacts when the image is resampled, because the displayed image size will usually not match the size of `*X*` (see `:doc:`gallery/images_contours_and_fields/image_antialiasing``). The resampling can be controlled via the `*interpolation*` parameter and/or `:rc:`image.interpolation``.

Parameters

-----

`X` : array-like or PIL image

The image data. Supported array shapes are:

- (M, N): an image with scalar data. The values are mapped to colors using normalization and a colormap. See parameters `*norm*`, `*cmap*`, `*vmin*`, `*vmax*`.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the image.

Out-of-range RGB(A) values are clipped.

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``  
The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*X*` is RGB(A).

`norm` : str or `~matplotlib.colors.Normalize``, optional  
The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*X*` is RGB(A).

`vmin`, `vmax` : float, optional  
When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

This parameter is ignored if `*X*` is RGB(A).

`colorizer` : `~matplotlib.colorbar.Colorizer`` or None, default: None  
The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

This parameter is ignored if `*X*` is RGB(A).

`aspect` : {'equal', 'auto'} or float or None, default: None  
The aspect ratio of the Axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling `~Axes.set_aspect``. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square (unless pixel sizes are explicitly made non-square in data coordinates using `*extent*`).
- 'auto': The Axes is kept fixed and the aspect is adjusted so that the data fit in the Axes. In general, this will result in non-square pixels.

Normally, `None` (the default) means to use `:rc:`image.aspect``. However, if the image uses a transform that does not contain the axes data transform, then `None` means to not modify the axes aspect at all (in that case, directly call `.Axes.set_aspect`` if desired).

`interpolation` : str, default: `:rc:`image.interpolation``  
The interpolation method used.

Supported values are 'none', 'auto', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'.

The data `*X*` is resampled to the pixel size of the image on the figure canvas, using the interpolation method to either up- or downsample the data.

If `*interpolation*` is 'none', then for the ps, pdf, and svg backends no down- or upsampling occurs, and the image data is passed to the backend as a native image. Note that different ps, pdf, and svg viewers may display these raw pixels differently. On other backends, 'none' is the same as 'nearest'.

If `*interpolation*` is the default 'auto', then 'nearest' interpolation is used if the image is upsampled by more than a factor of three (i.e. the number of display pixels is at least three times the size of the data array). If the upsampling rate is smaller than 3, or the image is downsampled, then 'hanning' interpolation is used to act as an anti-aliasing filter, unless the image happens to be upsampled by exactly a factor of two or one.

See `:doc:`gallery/images_contours_and_fields/interpolation_methods`` for an overview of the supported interpolation methods, and `:doc:`gallery/images_contours_and_fields/image_antialiasing`` for a discussion of image antialiasing.

Some interpolation methods require an additional radius parameter, which can be set by `*filterrad*`. Additionally, the antigrain image resize filter is controlled by the parameter `*filternorm*`.

`interpolation_stage` : {'auto', 'data', 'rgba'}, default: 'auto'  
Supported values:

- 'data': Interpolation is carried out on the data provided by the user. This is useful if interpolating between pixels during upsampling.
- 'rgba': The interpolation is carried out in RGBA-space after the color-mapping has been applied. This is useful if downsampling and combining pixels visually.
- 'auto': Select a suitable interpolation stage automatically. This uses 'rgba' when downsampling, or upsampling at a rate less than 3, and 'data' when upsampling at a higher rate.

See `:doc:`gallery/images_contours_and_fields/image_antialiasing`` for a discussion of image antialiasing.

alpha : float or array-like, optional

The alpha blending value, between 0 (transparent) and 1 (opaque).

If *\*alpha\** is an array, the alpha blending values are applied pixel by pixel, and *\*alpha\** must have the same shape as *\*X\**.

origin : {'upper', 'lower'}, default: :rc:`image.origin`

Place the [0, 0] index of the array in the upper left or lower left corner of the Axes. The convention (the default) 'upper' is typically used for matrices and images.

Note that the vertical axis points upward for 'lower' but downward for 'upper'.

See the :ref:`imshow\_extent` tutorial for examples and a more detailed description.

extent : floats (left, right, bottom, top), optional

The bounding box in data coordinates that the image will fill. These values may be unitful and match the units of the Axes. The image is stretched individually along x and y to fill the box.

The default extent is determined by the following conditions. Pixels have unit size in data coordinates. Their centers are on integer coordinates, and their center coordinates range from 0 to columns-1 horizontally and from 0 to rows-1 vertically.

Note that the direction of the vertical axis and thus the default values for top and bottom depend on *\*origin\**:

- For ``origin == 'upper'`` the default is ``(-0.5, numcols-0.5, numrows-0.5, -0.5)``.
- For ``origin == 'lower'`` the default is ``(-0.5, numcols-0.5, -0.5, numrows-0.5)``.

See the :ref:`imshow\_extent` tutorial for examples and a more detailed description.

filternorm : bool, default: True

A parameter for the antigrain image resize filter (see the antigrain documentation). If *\*filternorm\** is set, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

filterrad : float > 0, default: 4.0

The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

resample : bool, default: :rc:`image.resample`

When *\*True\**, use a full resampling method. When *\*False\**, only resample when the output image is larger than the input image.

url : str, optional  
Set the url of the created `.AxesImage`. See `.Artist.set_url`.

#### Returns

-----  
`~matplotlib.image.AxesImage`

#### Other Parameters

-----  
data : indexable object, optional  
If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

**\*\*kwargs** : `~matplotlib.artist.Artist` properties  
These parameters are passed on to the constructor of the `.AxesImage` artist.

#### See Also

-----  
matshow : Plot a matrix or an array as an image.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.imshow`.

Unless `*extent*` is used, pixel centers will be located at integer coordinates. In other words: the origin will coincide with the center of pixel (0, 0).

There are two common representations for RGB images with an alpha channel:

- Straight (unassociated) alpha: R, G, and B channels represent the color of the pixel, disregarding its opacity.
- Premultiplied (associated) alpha: R, G, and B channels represent the color of the pixel, adjusted for its opacity by multiplication.

`~matplotlib.pyplot.imshow` expects RGB images adopting the straight (unassociated) alpha representation.

## matplotlib.pyplot.inferno

```
inferno() -> 'None'
```

Set the colormap to 'inferno'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.install\_repl\_displayhook

---

```
install_repl_displayhook() -> 'None'
```

Connect to the display hook of the current shell.

The display hook gets called when the read-evaluate-print-loop (REPL) of the shell has finished the execution of a command. We use this callback to be able to automatically update a figure in interactive mode.

This works both with IPython and with vanilla python shells.

## matplotlib.pyplot.interactive

```
interactive(b)
```

Set whether to redraw after every plotting command (e.g. ``pyplot.xlabel``).

## matplotlib.pyplot.ioff

```
ioff() -> 'AbstractContextManager'
```

Disable interactive mode.

See ``pyplot.isinteractive`` for more details.

See Also

-----

`ion` : Enable interactive mode.

`isinteractive` : Whether interactive mode is enabled.

`show` : Show all figures (and maybe block).

`pause` : Show all figures, and block for a time.

Notes

-----

For a temporary change, this can be used as a context manager::

```
if interactive mode is on
```

```
then figures will be shown on creation
```

```
plt.ion()
```

```
This figure will be shown immediately
```

```
fig = plt.figure()
```

```
with plt.ioff():
```

```
interactive mode will be off
```

```
figures will not automatically be shown
```

```
fig2 = plt.figure()
```

```
...
```

To enable optional usage as a context manager, this function returns a context manager object, which is not intended to be stored or accessed by the user.

## matplotlib.pyplot.ion

```
ion() -> 'AbstractContextManager'
```

Enable interactive mode.

See ``pyplot.isinteractive`` for more details.

See Also

-----

`ioff` : Disable interactive mode.

`isinteractive` : Whether interactive mode is enabled.

`show` : Show all figures (and maybe block).

`pause` : Show all figures, and block for a time.

Notes

-----

For a temporary change, this can be used as a context manager::

```
if interactive mode is off
then figures will not be shown on creation
plt.ioff()
This figure will not be shown immediately
fig = plt.figure()
```

```
with plt.ion():
interactive mode will be on
figures will automatically be shown
fig2 = plt.figure()
...
```

To enable optional usage as a context manager, this function returns a context manager object, which is not intended to be stored or accessed by the user.

## matplotlib.pyplot.isinteractive

```
isinteractive() -> 'bool'
```

Return whether plots are updated after every plotting command.

The interactive mode is mainly useful if you build plots from the command line and want to see the effect of each command while you are building the figure.

In interactive mode:

- newly created figures will be shown immediately;
- figures will automatically redraw on change;
- ``pyplot.show`` will not block by default.

In non-interactive mode:

- newly created figures and changes to figures will not be reflected until explicitly asked to be;
- ``pyplot.show`` will block by default.

See Also

-----

ion : Enable interactive mode.  
ioff : Disable interactive mode.  
show : Show all figures (and maybe block).  
pause : Show all figures, and block for a time.

## matplotlib.pyplot.jet

```
jet() -> 'None'
```

Set the colormap to 'jet'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``help(colormaps)`` for more information.

## matplotlib.pyplot.legend

```
legend(*args, **kwargs) -> 'Legend'
```

Place a legend on the Axes.

Call signatures::

```
legend()
legend(handles, labels)
legend(handles=handles)
legend(labels)
```

The call signatures correspond to the following different ways to use this method:

**\*\*1. Automatic detection of elements to be shown in the legend\*\***

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the :meth:`~Artist.set\_label` method on the artist::

```
ax.plot([1, 2, 3], label='Inline label')
ax.legend()
```

or::

```
line, = ax.plot([1, 2, 3])
line.set_label('Label via method')
ax.legend()
```

.. note::

Specific artists can be excluded from the automatic legend element selection by using a label starting with an underscore, "". A string starting with an underscore is the default label for all artists, so calling ``Axes.legend`` without any arguments and without setting the labels manually will result in a ``UserWarning``

and an empty legend being drawn.

#### **\*\*2. Explicitly listing the artists and labels in the legend\*\***

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively::

```
ax.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

#### **\*\*3. Explicitly listing the artists in the legend\*\***

This is similar to 2, but the labels are taken from the artists' label properties. Example::

```
line1, = ax.plot([1, 2, 3], label='label1')
line2, = ax.plot([1, 2, 3], label='label2')
ax.legend(handles=[line1, line2])
```

#### **\*\*4. Labeling existing plot elements\*\***

.. admonition:: Discouraged

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

To make a legend for all artists on an Axes, call this function with an iterable of strings, one for each legend item. For example::

```
ax.plot([1, 2, 3])
ax.plot([5, 6, 7])
ax.legend(['First line', 'Second line'])
```

#### **Parameters**

handles : list of (``Artist`` or tuple of ``Artist``), optional  
A list of Artists (lines, patches) to be added to the legend.  
Use this together with `*labels*`, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

If an entry contains a tuple, then the legend handler for all Artists in the tuple will be placed alongside a single label.

labels : list of str, optional  
A list of labels to show next to the artists.  
Use this together with `*handles*`, if you need full control on what

is shown in the legend and the automatic mechanism described above is not sufficient.

#### Returns

-----  
`~matplotlib.legend.Legend`

#### Other Parameters

-----

`loc` : str or pair of floats, default: `:rc:~legend.loc``  
The location of the legend.

The strings ```upper left```, ```upper right```, ```lower left```, ```lower right``` place the legend at the corresponding corner of the axes.

The strings ```upper center```, ```lower center```, ```center left```, ```center right``` place the legend at the center of the corresponding edge of the axes.

The string ```center``` places the legend at the center of the axes.

The string ```best``` places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes coordinates (in which case `*bbox_to_anchor*` will be ignored).

For back-compatibility, ```center right``` (but no other location) can also be spelled ```right```, and each "string" location can also be given as a numeric value:

```
=====
Location String Location Code
=====
'best' (Axes only) 0
'upper right' 1
'upper left' 2
'lower left' 3
'lower right' 4
'right' 5
'center left' 6
'center right' 7
'lower center' 8
'upper center' 9
'center' 10
=====
```

`bbox_to_anchor` : ``~.BboxBase``, 2-tuple, or 4-tuple of floats  
Box that is used to position the legend in conjunction with `*loc*`.  
Defaults to ```axes.bbox``` (if called as a method to ``~.Axes.legend``) or

`figure.bbox` (if figure.legend`). This argument allows arbitrary placement of the legend.`

Bbox coordinates are interpreted in the coordinate system given by `*bbox_transform*`, with the default transform Axes or Figure coordinates, depending on which `legend`` is called.

If a 4-tuple or `.BboxBase`` is given, then it specifies the bbox ```(x, y, width, height)``` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the Axes (or figure)::

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple ```(x, y)``` places the corner of the legend specified by `*loc*` at x, y. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used::

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

`ncols : int, default: 1`  
The number of columns that the legend has.

For backward compatibility, the spelling `*ncol*` is also supported but it is discouraged. If both are given, `*ncols*` takes precedence.

`prop : None or ~matplotlib.font_manager.FontProperties` or dict`  
The font properties of the legend. If None (default), the current `:data:matplotlib.rcParams`` will be used.

`fontsize : int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}`  
The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if `*prop*` is not specified.

`labelcolor : str or list, default: :rc:`legend.labelcolor``  
The color of the text in the legend. Either a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using `:rc:`legend.labelcolor``. If None, use `:rc:`text.color``.

`numpoints : int, default: :rc:`legend.numpoints``  
The number of marker points in the legend when creating a legend entry for a `.Line2D`` (line).

`scatterpoints : int, default: :rc:`legend.scatterpoints``  
The number of marker points in the legend when creating a legend entry for a `.PathCollection`` (scatter plot).

`scatteryoffsets : iterable of floats, default: ``[0.375, 0.5, 0.3125]```  
The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the

legend text, and 1.0 is at the top. To draw all markers at the same height, set to ``[0.5]``.

markerscale : float, default: :rc:`legend.markerscale`  
The relative size of legend markers compared to the originally drawn ones.

markerfirst : bool, default: True  
If \*True\*, legend marker is placed to the left of the legend label.  
If \*False\*, legend marker is placed to the right of the legend label.

reverse : bool, default: False  
If \*True\*, the legend labels are displayed in reverse order from the input.  
If \*False\*, the legend labels are displayed in the same order as the input.

.. versionadded:: 3.7

frameon : bool, default: :rc:`legend.frameon`  
Whether the legend should be drawn on a patch (frame).

fancybox : bool, default: :rc:`legend.fancybox`  
Whether round edges should be enabled around the ``.FancyBboxPatch`` which makes up the legend's background.

shadow : None, bool or dict, default: :rc:`legend.shadow`  
Whether to draw a shadow behind the legend.  
The shadow can be configured using ``.Patch`` keywords.  
Customization via :rc:`legend.shadow` is currently not supported.

framealpha : float, default: :rc:`legend.framealpha`  
The alpha transparency of the legend's background.  
If \*shadow\* is activated and \*framealpha\* is ``None``, the default value is ignored.

facecolor : "inherit" or color, default: :rc:`legend.facecolor`  
The legend's background color.  
If ``"inherit"`` , use :rc:`axes.facecolor`.

edgecolor : "inherit" or color, default: :rc:`legend.edgecolor`  
The legend's background patch edge color.  
If ``"inherit"`` , use :rc:`axes.edgecolor`.

mode : {"expand", None}  
If \*mode\* is set to ``"expand"`` the legend will be horizontally expanded to fill the Axes area (or \*bbox\_to\_anchor\* if defines the legend's size).

bbox\_transform : None or ~matplotlib.transforms.Transform`  
The transform for the bounding box (\*bbox\_to\_anchor\*). For a value of ``None`` (default) the Axes' :data:`~matplotlib.axes.Axes.transAxes` transform will be used.

title : str or None  
The legend's title. Default is no title (``None``).

title\_fontproperties : None or ~matplotlib.font\_manager.FontProperties` or dict

The font properties of the legend's title. If None (default), the `*title_fontsize*` argument will be used if present; if `*title_fontsize*` is also None, the current `:rc:`legend.title_fontsize`` will be used.

`title_fontsize` : int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: `:rc:`legend.title_fontsize``

The font size of the legend's title.

Note: This cannot be combined with `*title_fontproperties*`. If you want to set the fontsize alongside other font properties, use the `*size*` parameter in `*title_fontproperties*`.

`alignment` : {'center', 'left', 'right'}, default: 'center'

The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

`borderpad` : float, default: `:rc:`legend.borderpad``

The fractional whitespace inside the legend border, in font-size units.

`labelspacing` : float, default: `:rc:`legend.labelspacing``

The vertical space between the legend entries, in font-size units.

`handlelength` : float, default: `:rc:`legend.handlelength``

The length of the legend handles, in font-size units.

`handleheight` : float, default: `:rc:`legend.handleheight``

The height of the legend handles, in font-size units.

`handletextpad` : float, default: `:rc:`legend.handletextpad``

The pad between the legend handle and text, in font-size units.

`borderaxespad` : float, default: `:rc:`legend.borderaxespad``

The pad between the Axes and legend border, in font-size units.

`columnspacing` : float, default: `:rc:`legend.columnspacing``

The spacing between columns, in font-size units.

`handler_map` : dict or None

The custom dictionary mapping instances or types to a legend handler. This `*handler_map*` updates the default handler map found at ``matplotlib.legend.Legend.get_legend_handler_map``.

`draggable` : bool, default: False

Whether the legend can be dragged with the mouse.

See Also

-----  
`.Figure.legend`

Notes

-----

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for ``axes.Axes.legend``.

Some artists are not supported by this function. See `:ref:`legend_guide`` for details.

#### Examples

-----

.. plot:: gallery/text\_labels\_and\_annotations/legend.py

## matplotlib.pyplot.locator\_params

```
locator_params(axis: "Literal['both', 'x', 'y']" = 'both', tight: 'bool | None' = None, **kwargs) -> 'None'
```

Control behavior of major tick locators.

Because the locator is involved in autoscaling, `~.Axes.autoscale_view`` is called automatically after the parameters are changed.

#### Parameters

-----

axis : {'both', 'x', 'y'}, default: 'both'

The axis on which to operate. (For 3D Axes, \*axis\* can also be set to 'z', and 'both' refers to all three axes.)

tight : bool or None, optional

Parameter passed to `~.Axes.autoscale_view``.

Default is None, for no change.

#### Other Parameters

-----

**\*\*kwargs**

Remaining keyword arguments are passed to directly to the ```set_params()``` method of the locator. Supported keywords depend on the type of the locator. See for example `~.ticker.MaxNLocator.set_params`` for the `~.ticker.MaxNLocator`` used by default for linear.

#### Notes

-----

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>`` for `~.axes.Axes.locator_params``.

#### Examples

-----

When plotting small subplots, one might want to reduce the maximum number of ticks and use tight bounds, for example::

```
ax.locator_params(tight=True, nbins=4)
```

## matplotlib.pyplot.loglog

```
loglog(*args, **kwargs) -> 'list[Line2D]'
```

Make a plot with log scaling on both the x- and y-axis.

Call signatures::

```
loglog([x], y, [fmt], data=None, **kwargs)
loglog([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around `.plot` which additionally changes both the x-axis and the y-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters `*base*`, `*subs*` and `*nonpositive*` control the x/y-axis properties. They are just forwarded to `.Axes.set_xscale` and `.Axes.set_yscale`. To use different properties on the x-axis and the y-axis, use e.g.  
`ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`.

Parameters

-----  
`base` : float, default: 10  
Base of the logarithm.

`subs` : sequence, optional  
The location of the minor ticks. If `*None*`, reasonable locations are automatically chosen depending on the number of decades in the plot. See `.Axes.set_xscale` / `.Axes.set_yscale` for details.

`nonpositive` : {'mask', 'clip'}, default: 'clip'  
Non-positive values can be masked as invalid, or clipped to a very small positive number.

`**kwargs`  
All parameters supported by `.plot`.

Returns

-----  
list of `.Line2D`  
Objects representing the plotted data.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.loglog`.

## matplotlib.pyplot.magma

```
magma() -> 'None'
```

Set the colormap to 'magma'.

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)` for more information.

## matplotlib.pyplot.magnitude\_spectrum

```
magnitude_spectrum(x: 'ArrayLike', *, Fs: 'float | None' = None, Fc: 'int | None' = None, window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None, scale: "Literal['default', 'linear', 'dB'] | None" = None, data=None, **kwargs)
-> 'tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the magnitude spectrum.

Compute the magnitude spectrum of *x*. Data is padded to a length of *pad\_to* and the windowing function *window* is applied to the signal.

### Parameters

-----

*x* : 1-D array or sequence

Array or sequence containing the data.

*Fs* : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

*window* : callable or ndarray, default: ``window_hanning``

A function or a vector of length *NFFT*. To create window vectors see ``window_hanning``, ``window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

*sides* : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

*pad\_to* : int, optional

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to ``~numpy.fft.fft``. The default is None, which sets *pad\_to* equal to the length of the input signal (i.e. no padding).

*scale* : {'default', 'linear', 'dB'}

The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale, i.e., the dB amplitude ( $20 * \log_{10}$ ). 'default' is 'linear'.

*Fc* : int, default: 0

The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

### Returns

-----

*spectrum* : 1-D array

The values for the magnitude spectrum before scaling (real valued).

freqs : 1-D array

The frequencies corresponding to the elements in `*spectrum*`.

line : `~matplotlib.lines.Line2D``

The line created by this function.

#### Other Parameters

-----  
data : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*X*`

`**kwargs`

Keyword arguments control the ``.Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: ``.CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``.JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

`markersize` or `ms`: float

`markevery`: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

`mouseover`: bool

`path_effects`: list of ``.AbstractPathEffect``

`picker`: float or callable[[Artist, Event], tuple[bool, dict]]

`pickradius`: float

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

solid\_capstyle: ``CapStyle`` or `{'butt', 'projecting', 'round'}`  
solid\_joinstyle: ``JoinStyle`` or `{'miter', 'round', 'bevel'}`  
transform: unknown  
url: str  
visible: bool  
xdata: 1D array  
ydata: 1D array  
zorder: float

See Also

-----

psd

Plots the power spectral density.

angle\_spectrum

Plots the angles of the corresponding frequencies.

phase\_spectrum

Plots the phase (unwrapped angle) of the corresponding frequencies.

specgram

Can plot the magnitude spectrum of segments within the signal in a colormap.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``axes.Axes.magnitude_spectrum``.

## matplotlib.pyplot.margins

```
margins(*margins: 'float', x: 'float | None' = None, y: 'float | None' = None, tight: 'bool | None' = True) -> 'tuple[float, float] | None'
```

Set or retrieve margins around the data for autoscaling axis limits.

This allows to configure the padding around the data without having to set explicit limits using ``~.Axes.set_xlim`` / ``~.Axes.set_ylim``.

Autoscaling determines the axis limits by adding `*margin*` times the data interval as padding around the data. See the following illustration:

.. plot:: \_embedded\_plots/axes\_margins.py

All input parameters must be floats greater than -0.5. Passing both positional and keyword arguments is invalid and will raise a `TypeError`. If no arguments (positional or otherwise) are provided, the current margins will remain unchanged and simply be returned.

The default margins are :rc:`axes.xmargin` and :rc:`axes.ymargin`.

Parameters

-----

`*margins` : float, optional

If a single positional argument is provided, it specifies both margins of the x-axis and y-axis limits. If two

positional arguments are provided, they will be interpreted as `*xmargin*`, `*ymargin*`. If setting the margin on a single axis is desired, use the keyword arguments described below.

`x, y` : float, optional  
Specific margin values for the x-axis and y-axis, respectively. These cannot be used with positional arguments, but can be used individually to alter on e.g., only the y-axis.

`tight` : bool or None, default: True  
The `*tight*` parameter is passed to `~.axes.Axes.autoscale_view``, which is executed after a margin is changed; the default here is `*True*`, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting `*tight*` to `*None*` preserves the previous setting.

Returns

-----  
`xmargin, ymargin` : float

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.margins``.

If a previously used Axes method such as :meth:`pcolor` has set `~.Axes.use_sticky_edges`` to ``True``, only the limits not set by the "sticky artists" will be modified. To force all margins to be set, set `~.Axes.use_sticky_edges`` to ``False`` before calling :meth:`margins`.

See Also

-----  
`.Axes.set_xmargin`, `.Axes.set_ymargin`

## matplotlib.pyplot.matshow

```
matshow(A: 'ArrayLike', fignum: 'None | int' = None, **kwargs) -> 'AxesImage'
```

Display a 2D array as a matrix in a new figure window.

The origin is set at the upper left hand corner.  
The indexing is ```(row, column)``` so that the first index runs vertically and the second index runs horizontally in the figure:

.. code-block:: none

```
A[0, 0] ■ A[0, M-1]
■ ■
A[N-1, 0] ■ A[N-1, M-1]
```

The aspect ratio of the figure window is that of the array, unless this would make an excessively short or narrow figure.

Tick labels for the xaxis are placed on top.

#### Parameters

-----

A : 2D array-like

The matrix to be displayed.

fignum : None or int

If \*None\*, create a new, appropriately sized figure window.

If 0, use the current Axes (creating one if there is none, without ever adjusting the figure size).

Otherwise, create a new Axes on the figure with the given number (creating it at the appropriate size if it does not exist, but not adjusting the figure size otherwise). Note that this will be drawn on top of any preexisting Axes on the figure.

#### Returns

-----

`~matplotlib.image.AxesImage`

#### Other Parameters

-----

\*\*kwargs : `~matplotlib.axes.Axes.imshow` arguments

## matplotlib.pyplot.minorticks\_off

```
minorticks_off() -> 'None'
```

Remove minor ticks from the Axes.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.minorticks_off`.

## matplotlib.pyplot.minorticks\_on

```
minorticks_on() -> 'None'
```

Display minor ticks on the Axes.

Displaying minor ticks may reduce performance; you may turn them off using `~minorticks_off()` if drawing speed is a problem.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.minorticks_on``.

## matplotlib.pyplot.new\_figure\_manager

```
new_figure_manager(num, *args, FigureClass=, **kwargs)
```

Create a new figure manager instance.

## matplotlib.pyplot.nipy\_spectral

```
nipy_spectral() -> 'None'
```

Set the colormap to 'nipy\_spectral'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.pause

```
pause(interval: 'float') -> 'None'
```

Run the GUI event loop for `*interval*` seconds.

If there is an active figure, it will be updated and displayed before the pause, and the GUI event loop (if any) will run during the pause.

This can be used for crude animation. For more complex animation use :mod:`matplotlib.animation`.

If there is no active figure, sleep for `*interval*` seconds instead.

See Also

-----

matplotlib.animation : Proper animations

show : Show all figures and optional block until all figures are closed.

## matplotlib.pyplot.pcolor

```
pcolor(*args: 'ArrayLike', shading: "Literal['flat', 'nearest', 'auto'] | None" = None, alpha: 'float | None' = None, norm: 'str | Normalize | None' = None, cmap: 'str | Colormap | None' = None, vmin: 'float | None' = None, vmax: 'float | None' = None, colorizer: 'Colorizer | None' = None, data=None, **kwargs) -> 'Collection'
```

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature::

```
pcolor([X, Y,] C, /, **kwargs)
```

`*X*` and `*Y*` can be used to specify the corners of the quadrilaterals.

The arguments `*X*`, `*Y*`, `*C*` are positional-only.


.. hint::

`pcolor()` can be very slow for large arrays. In most cases you should use the similar but much faster `~.Axes.pcolormesh` instead. See [:ref: Differences between pcolor\(\) and pcolormesh\(\)](#) [<differences-pcolor-pcolormesh>](#) for a discussion of the differences.

#### Parameters

`C` : 2D array-like  
The color-mapped values. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

`X`, `Y` : array-like, optional  
The coordinates of the corners of quadrilaterals of a `pcolormesh`:

`(X[i+1, j], Y[i+1, j]) (X[i+1, j+1], Y[i+1, j+1])`  
  
`(X[i, j], Y[i, j]) (X[i, j+1], Y[i, j+1])`

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the [:ref: Notes <axes-pcolormesh-grid-orientation>](#) section below.

If `shading='flat'` the dimensions of `*X*` and `*Y*` should be one greater than those of `*C*`, and the quadrilateral is colored due to the value at `C[i, j]`. If `*X*`, `*Y*` and `*C*` have equal dimensions, a warning will be raised and the last row and column of `*C*` will be ignored.

If `shading='nearest'`, the dimensions of `*X*` and `*Y*` should be the same as those of `*C*` (if not, a `ValueError` will be raised). The color `C[i, j]` will be centered on `(X[i, j], Y[i, j])`.

If `*X*` and/or `*Y*` are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2D arrays, making a rectangular grid.

`shading` : {'flat', 'nearest', 'auto'}, default: `rc:~pcolor.shading`  
The fill style for the quadrilateral. Possible values:

- 'flat': A solid color is used for each quad. The color of the quad `(i, j)`, `(i+1, j)`, `(i, j+1)`, `(i+1, j+1)` is given by `C[i, j]`. The dimensions of `*X*` and `*Y*` should be one greater than those of `*C*`; if they are the same as `*C*`, then a deprecation warning is raised, and the last row and column of `*C*` are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of `*X*` and `*Y*` must be the same as `*C*`.

- 'auto': Choose 'flat' if dimensions of *\*X\** and *\*Y\** are one larger than *\*C\**. Choose 'nearest' if dimensions are the same.

See :doc:`gallery/images\_contours\_and\_fields/pcolormesh\_grids` for more description.

cmap : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``  
The Colormap instance or registered colormap name used to map scalar data to colors.

norm : str or `~matplotlib.colors.Normalize``, optional  
The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *\*cmap\**. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `~matplotlib.colors.Normalize`` or one of its subclasses (see :ref:`colormapnorms`).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `~matplotlib.colors.Normalize`` subclass is dynamically generated and instantiated.

vmin, vmax : float, optional  
When using scalar data and no explicit *\*norm\**, *\*vmin\** and *\*vmax\** define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *\*vmin\*/vmax\** when a *\*norm\** instance is given (but using a `~matplotlib.colors.Normalize`` name together with *\*vmin\*/vmax\** is acceptable).

colorizer : `~matplotlib.colorbar.Colorizer`` or None, default: None  
The Colorizer object used to map color to data. If None, a Colorizer object is created from a *\*norm\** and *\*cmap\**.

edgecolors : {'none', None, 'face', color, color sequence}, optional  
The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.
- *\*None\**: `:rc:`patch.edgecolor`` will be used. Note that currently `:rc:`patch.force_edgecolor`` has to be True for this to work.
- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form *\*edgecolor\** works as an alias.

alpha : float, default: None  
The alpha blending value of the face color, between 0 (transparent) and 1 (opaque). Note: The edgecolor is currently not affected by this.

snap : bool, default: False  
Whether to snap the mesh to pixel boundaries.

Returns

-----  
`matplotlib.collections.PolyQuadMesh`

## Other Parameters

-----  
antialiaseds : bool, default: False

The default `*antialiaseds*` is False if the default `*edgecolors*` is not "none" is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If `*edgecolors*` is not "none", then the default `*antialiaseds*` is taken from `:rc:`patch.antialiased``. Stroking the edges may be preferred if `*alpha*` is 1, but will cause artifacts otherwise.

data : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

**\*\*kwargs**

Additionally, the following arguments are allowed. They are passed along to the `~matplotlib.collections.PolyQuadMesh`` constructor:

Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: array-like or float or None

animated: bool

antialiased or aa or antialiaseds: bool or list of bools

array: array-like or None

capstyle: ``.CapStyle`` or {'butt', 'projecting', 'round'}

clim: (vmin: float, vmax: float)

clip\_box: `~matplotlib.transforms.BboxBase`` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

cmap: ``.Colormap`` or str or None

color: `:mptype:`color`` or list of RGBA tuples

edgecolor or ec or edgecolors: `:mptype:`color`` or list of `:mptype:`color`` or 'face'

facecolor or facecolors or fc: `:mptype:`color`` or list of `:mptype:`color``

figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

gid: str

hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '\*'}

hatch\_linewidth: unknown

in\_layout: bool

joinstyle: ``.JoinStyle`` or {'miter', 'round', 'bevel'}

label: object

linestyle or dashes or linestyles or ls: str or tuple or list thereof

linewidth or linewidths or lw: float or list of floats

mouseover: bool

norm: ``.Normalize`` or str or None

offset\_transform or transOffset: ``.Transform``

offsets: (N, 2) or (2,) array-like

path\_effects: list of ``.AbstractPathEffect``

paths: list of array-like

picker: None or bool or float or callable

pickradius: float

rasterized: bool  
sizes: `numpy.ndarray` or None  
sketch\_params: (scale: float, length: float, randomness: float)  
snap: bool or None  
transform: `~matplotlib.transforms.Transform`  
url: str  
urls: list of str or None  
verts: list of array-like  
verts\_and\_codes: unknown  
visible: bool  
zorder: float

See Also

-----  
pcolormesh : for an explanation of the differences between  
pcolor and pcolormesh.  
imshow : If \*X\* and \*Y\* are each equidistant, `~.Axes.imshow` can be a  
faster alternative.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `.axes.Axes.pcolor`.`

**\*\*Masked arrays\*\***

\*X\*, \*Y\* and \*C\* may be masked arrays. If either `C[i, j]`, or one  
of the vertices surrounding `C[i, j]` (\*X\* or \*Y\* at  
`[i, j]`, `[i+1, j]`, `[i, j+1]`, `[i+1, j+1]`) is masked, nothing is  
plotted.

.. \_axes-pcolor-grid-orientation:

**\*\*Grid orientation\*\***

The grid orientation follows the standard matrix convention: An array  
\*C\* with shape (nrows, ncolumns) is plotted with the column number as  
\*X\* and the row number as \*Y\*.

## matplotlib.pyplot.pcolormesh

```
pcolormesh(*args: 'ArrayLike', alpha: 'float | None' = None, norm: 'str | Normalize |
None' = None, cmap: 'str | Colormap | None' = None, vmin: 'float | None' = None, vmax:
'float | None' = None, colorizer: 'Colorizer | None' = None, shading: "Literal['flat',
'nearest', 'gouraud', 'auto'] | None" = None, antialiased: 'bool' = False, data=None,
**kwargs) -> 'QuadMesh'
```

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature::

pcolormesh([X, Y,] C, /, \*\*kwargs)

`*X*` and `*Y*` can be used to specify the corners of the quadrilaterals.

The arguments `*X*`, `*Y*`, `*C*` are positional-only.

.. hint::

`~.Axes.pcolormesh`` is similar to `~.Axes.pcolor``. It is much faster and preferred in most cases. For a detailed discussion on the differences see :ref:`Differences between pcolor() and pcolormesh() <differences-pcolor-pcolormesh>`.

#### Parameters

-----

`C` : array-like

The mesh data. Supported array shapes are:

- (M, N) or M\*N: a mesh with scalar data. The values are mapped to colors using normalization and a colormap. See parameters `*norm*`, `*cmap*`, `*vmin*`, `*vmax*`.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the mesh data.

`X`, `Y` : array-like, optional

The coordinates of the corners of quadrilaterals of a `pcolormesh`::

(`X[i+1, j]`, `Y[i+1, j]`) (`X[i+1, j+1]`, `Y[i+1, j+1]`)

●■■■■■●

■ ■

●■■■■■●

(`X[i, j]`, `Y[i, j]`) (`X[i, j+1]`, `Y[i, j+1]`)

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the :ref:`Notes <axes-pcolormesh-grid-orientation>` section below.

If ```shading='flat'``` the dimensions of `*X*` and `*Y*` should be one greater than those of `*C*`, and the quadrilateral is colored due to the value at ```C[i, j]```. If `*X*`, `*Y*` and `*C*` have equal dimensions, a warning will be raised and the last row and column of `*C*` will be ignored.

If ```shading='nearest'``` or ```'gouraud'```, the dimensions of `*X*` and `*Y*` should be the same as those of `*C*` (if not, a `ValueError` will be raised). For ```nearest'``` the color ```C[i, j]``` is centered on ```(X[i, j], Y[i, j])```. For ```gouraud'```, a smooth interpolation is carried out between the quadrilateral corners.

If `*X*` and/or `*Y*` are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2D arrays, making a rectangular grid.

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``  
The Colormap instance or registered colormap name used to map scalar data to colors.

`norm` : str or `~matplotlib.colors.Normalize``, optional  
The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

`vmin, vmax` : float, optional  
When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

`colorizer` : `~matplotlib.colorbar.Colorizer`` or None, default: None  
The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

`edgecolors` : {'none', None, 'face', color, color sequence}, optional  
The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.
- `*None*`: `:rc:`patch.edgecolor`` will be used. Note that currently `:rc:`patch.force_edgecolor`` has to be True for this to work.
- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form `*edgecolor*` works as an alias.

`alpha` : float, default: None  
The alpha blending value, between 0 (transparent) and 1 (opaque).

`shading` : {'flat', 'nearest', 'gouraud', 'auto'}, optional  
The fill style for the quadrilateral; defaults to `:rc:`pcolor.shading``. Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by ```C[i, j]```. The dimensions of `*X*` and `*Y*` should be one greater than those of `*C*`; if they are the same as `*C*`, then a deprecation warning is raised, and the last row and column of `*C*` are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The

dimensions of `*X*` and `*Y*` must be the same as `*C*`.

- 'gouraud': Each quad will be Gouraud shaded: The color of the corners (`i`, `j`) are given by `C[i, j]`. The color values of the area in between is interpolated from the corner values. The dimensions of `*X*` and `*Y*` must be the same as `*C*`. When Gouraud shading is used, `*edgecolors*` is ignored.
- 'auto': Choose 'flat' if dimensions of `*X*` and `*Y*` are one larger than `*C*`. Choose 'nearest' if dimensions are the same.

See `:doc:/gallery/images_contours_and_fields/pcolormesh_grids` for more description.

`snap` : bool, default: False  
Whether to snap the mesh to pixel boundaries.

`rasterized` : bool, optional  
Rasterize the pcolormesh when drawing vector graphics. This can speed up rendering and produce smaller files for large data sets. See also `:doc:/gallery/misc/rasterization_demo`.

## Returns

-----  
`~matplotlib.collections.QuadMesh`

## Other Parameters

-----  
`data` : indexable object, optional  
If given, all parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`.

## \*\*kwargs

Additionally, the following arguments are allowed. They are passed along to the `~matplotlib.collections.QuadMesh` constructor:

## Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image  
`alpha`: array-like or float or None  
`animated`: bool  
`antialiased` or `aa` or `antialiaseds`: bool or list of bools  
`array`: array-like  
`capstyle`: `~matplotlib.collections.QuadMesh` or {'butt', 'projecting', 'round'}  
`clim`: (vmin: float, vmax: float)  
`clip_box`: `~matplotlib.transforms.BboxBase` or None  
`clip_on`: bool  
`clip_path`: Patch or (Path, Transform) or None  
`cmap`: `~matplotlib.colors.Colormap` or str or None  
`color`: `~matplotlib.colors.mpltype:color` or list of RGBA tuples  
`edgecolor` or `ec` or `edgecolors`: `~matplotlib.colors.mpltype:color` or list of `~matplotlib.colors.mpltype:color` or 'face'  
`facecolor` or `facecolors` or `fc`: `~matplotlib.colors.mpltype:color` or list of `~matplotlib.colors.mpltype:color`  
`figure`: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`  
`gid`: str  
`hatch`: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '\*'}  
`hatch_linewidth`: unknown  
`in_layout`: bool

joinstyle: ``JoinStyle`` or `{'miter', 'round', 'bevel'}`  
 label: object  
 linestyle or dashes or linestyles or ls: str or tuple or list thereof  
 linewidth or linewidths or lw: float or list of floats  
 mouseover: bool  
 norm: ``Normalize`` or str or None  
 offset\_transform or transOffset: ``Transform``  
 offsets: (N, 2) or (2,) array-like  
 path\_effects: list of ``AbstractPathEffect``  
 picker: None or bool or float or callable  
 pickradius: float  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 transform: ``~matplotlib.transforms.Transform``  
 url: str  
 urls: list of str or None  
 visible: bool  
 zorder: float

#### See Also

-----  
 pcolor : An alternative implementation with slightly different features. For a detailed discussion on the differences see :ref:`Differences between pcolor() and pcolormesh()` < differences-pcolor-pcolormesh >`.  
 imshow : If `*X*` and `*Y*` are each equidistant, ``~.Axes.imshow`` can be a faster alternative.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``~.axes.Axes.pcolormesh``.

**\*\*Masked arrays\*\***

`*C*` may be a masked array. If ``C[i, j]`` is masked, the corresponding quadrilateral will be transparent. Masking of `*X*` and `*Y*` is not supported. Use ``~.Axes.pcolor`` if you need this functionality.

.. \_axes-pcolormesh-grid-orientation:

**\*\*Grid orientation\*\***

The grid orientation follows the standard matrix convention: An array `*C*` with shape (nrows, ncolumns) is plotted with the column number as `*X*` and the row number as `*Y*`.

.. \_differences-pcolor-pcolormesh:

**\*\*Differences between pcolor() and pcolormesh()\*\***

Both methods are used to create a pseudocolor plot of a 2D array

using quadrilaterals.

The main difference lies in the created object and internal data handling:

While `~.Axes.pcolor`` returns a `~.PolyQuadMesh``, `~.Axes.pcolormesh`` returns a `~.QuadMesh``. The latter is more specialized for the given purpose and thus is faster. It should almost always be preferred.

There is also a slight difference in the handling of masked arrays. Both `~.Axes.pcolor`` and `~.Axes.pcolormesh`` support masked arrays for `*C*`. However, only `~.Axes.pcolor`` supports masked arrays for `*X*` and `*Y*`. The reason lies in the internal handling of the masked values. `~.Axes.pcolor`` leaves out the respective polygons from the `PolyQuadMesh`. `~.Axes.pcolormesh`` sets the facecolor of the masked elements to transparent. You can see the difference when using edgecolors. While all edges are drawn irrespective of masking in a `QuadMesh`, the edge between two adjacent masked quadrilaterals in `~.Axes.pcolor`` is not drawn as the corresponding polygons do not exist in the `PolyQuadMesh`. Because `PolyQuadMesh` draws each individual polygon, it also supports applying hatches and linestyle to the collection.

Another difference is the support of Gouraud shading in `~.Axes.pcolormesh``, which is not available with `~.Axes.pcolor``.

## matplotlib.pyplot.phase\_spectrum

```
phase_spectrum(x: 'ArrayLike', *, Fs: 'float | None' = None, Fc: 'int | None' = None,
window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, pad_to: 'int |
None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None,
data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the phase spectrum.

Compute the phase spectrum (unwrapped angle spectrum) of `*x*`. Data is padded to a length of `*pad_to*` and the windowing function `*window*` is applied to the signal.

### Parameters

-----

`x` : 1-D array or sequence

Array or sequence containing the data

`Fs` : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

`window` : callable or ndarray, default: `~.window_hanning``

A function or a vector of length `*NFFT*`. To create window vectors see `~.window_hanning``, `~.window_none``, `numpy.blackman``, `numpy.hamming``, `numpy.bartlett``, `scipy.signal``, `scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real

data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to `~numpy.fft.fft`. The default is None, which sets `*pad_to*` equal to the length of the input signal (i.e. no padding).

`Fc` : int, default: 0

The center frequency of `*x*`, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns

-----

`spectrum` : 1-D array

The values for the phase spectrum in radians (real valued).

`freqs` : 1-D array

The frequencies corresponding to the elements in `*spectrum*`.

`line` : `~matplotlib.lines.Line2D``

The line created by this function.

Other Parameters

-----

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`

`**kwargs`

Keyword arguments control the ``.Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mptype:~color``

`dash_capstyle`: ``.CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``.JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

gapcolor: :mpltype:`color` or None  
 gid: str  
 in\_layout: bool  
 label: object  
 linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}  
 linewidth or lw: float  
 marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`  
 markeredgcolor or mec: :mpltype:`color`  
 markeredgewidth or mew: float  
 markerfacecolor or mfc: :mpltype:`color`  
 markerfacecoloralt or mfcalt: :mpltype:`color`  
 markersize or ms: float  
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]  
 mouseover: bool  
 path\_effects: list of `~.AbstractPathEffect`  
 picker: float or callable[[Artist, Event], tuple[bool, dict]]  
 pickradius: float  
 rasterized: bool  
 sketch\_params: (scale: float, length: float, randomness: float)  
 snap: bool or None  
 solid\_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}  
 solid\_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}  
 transform: unknown  
 url: str  
 visible: bool  
 xdata: 1D array  
 ydata: 1D array  
 zorder: float

#### See Also

-----

`magnitude_spectrum`

Plots the magnitudes of the corresponding frequencies.

`angle_spectrum`

Plots the wrapped version of this function.

`specgram`

Can plot the phase spectrum of segments within the signal in a colormap.

#### Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `~.axes.Axes.phase_spectrum``.

**matplotlib.pyplot.pie**

---

```
pie(x: 'ArrayLike', *, explode: 'ArrayLike | None' = None, labels: 'Sequence[str] | None' = None, colors: 'ColorType | Sequence[ColorType] | None' = None, autopct: 'str | Callable[[float], str] | None' = None, pctdistance: 'float' = 0.6, shadow: 'bool' = False, labeldistance: 'float | None' = 1.1, startangle: 'float' = 0, radius: 'float' = 1, counterclock: 'bool' = True, wedgeprops: 'dict[str, Any] | None' = None, textprops: 'dict[str, Any] | None' = None, center: 'tuple[float, float]' = (0, 0), frame: 'bool' = False, rotatelabels: 'bool' = False, normalize: 'bool' = True, hatch: 'str | Sequence[str] | None' = None, data=None) -> 'tuple[list[Wedge], list[Text]] | tuple[list[Wedge], list[Text], list[Text]]'
```

Plot a pie chart.

Make a pie chart of array *x*. The fractional area of each wedge is given by `x/sum(x)`.

The wedges are plotted counterclockwise, by default starting from the x-axis.

Parameters

*x* : 1D array-like  
The wedge sizes.

*explode* : array-like, default: None  
If not *None*, is a `len(x)` array which specifies the fraction of the radius with which to offset each wedge.

*labels* : list, default: None  
A sequence of strings providing the labels for each wedge

*colors* : `mpltype:color` or list of `mpltype:color`, default: None  
A sequence of colors through which the pie chart will cycle. If *None*, will use the colors in the currently active cycle.

*hatch* : str or list, default: None  
Hatching pattern applied to all pie wedges or sequence of patterns through which the chart will cycle. For a list of valid patterns, see `:doc:/gallery/shapes_and_collections/hatch_style_reference`.

.. versionadded:: 3.7

*autopct* : None or str or callable, default: None  
If not *None*, *autopct* is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If *autopct* is a format string, the label will be `fmt % pct`. If *autopct* is a function, then it will be called.

*pctdistance* : float, default: 0.6  
The relative distance along the radius at which the text generated by *autopct* is drawn. To draw the text outside the pie, set *pctdistance* > 1. This parameter is ignored if *autopct* is *None*.

*labeldistance* : float or None, default: 1.1  
The relative distance along the radius at which the labels are drawn. To draw the labels inside the pie, set *labeldistance* < 1. If set to *None*, labels are not drawn but are still stored for

use in ``.legend``.

`shadow` : bool or dict, default: False

If bool, whether to draw a shadow beneath the pie. If dict, draw a shadow passing the properties in the dict to ``.Shadow``.

.. versionadded:: 3.8

`*shadow*` can be a dict.

`startangle` : float, default: 0 degrees

The angle by which the start of the pie is rotated, counterclockwise from the x-axis.

`radius` : float, default: 1

The radius of the pie.

`counterclock` : bool, default: True

Specify fractions direction, clockwise or counterclockwise.

`wedgeprops` : dict, default: None

Dict of arguments passed to each ``.patches.Wedge`` of the pie.

For example, ```wedgeprops = {'linewidth': 3}``` sets the width of the wedge border lines equal to 3. By default, ```clip_on=False```.

When there is a conflict between these properties and other keywords, properties passed to `*wedgeprops*` take precedence.

`textprops` : dict, default: None

Dict of arguments to pass to the text objects.

`center` : (float, float), default: (0, 0)

The coordinates of the center of the chart.

`frame` : bool, default: False

Plot Axes frame with the chart if true.

`rotatelabels` : bool, default: False

Rotate each label to the angle of the corresponding slice if true.

`normalize` : bool, default: True

When `*True*`, always make a full pie by normalizing `x` so that ```sum(x) == 1```. `*False*` makes a partial pie if ```sum(x) <= 1``` and raises a ``ValueError`` for ```sum(x) > 1```.

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*explode*`, `*labels*`, `*colors*`

Returns

-----

`patches` : list

A sequence of ``.matplotlib.patches.Wedge`` instances

`texts` : list

A list of the label `Text`` instances.

`autotexts` : list

A list of `Text`` instances for the numeric labels. This will only be returned if the parameter `*autopct*` is not `*None*`.

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for `axes.Axes.pie``.

The pie chart will probably look best if the figure and Axes are square, or the Axes aspect is equal.

This method sets the aspect ratio of the axis to "equal".

The Axes aspect ratio can be controlled with `axes.set_aspect``.

## matplotlib.pyplot.pink

```
pink() -> 'None'
```

Set the colormap to 'pink'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.plasma

```
plasma() -> 'None'
```

Set the colormap to 'plasma'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

## matplotlib.pyplot.plot

```
plot(*args: 'float | ArrayLike | str', scalex: 'bool' = True, scaley: 'bool' = True, data=None, **kwargs) -> 'list[Line2D]'
```

Plot y versus x as lines and/or markers.

Call signatures::

```
plot([x], y, [fmt], *, data=None, **kwargs)
```

```
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by `*x*`, `*y*`.

The optional parameter `*fmt*` is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the `*Notes*` section below.

```
>>> plot(x, y) # plot x and y using default line style and color
```

```
>>> plot(x, y, 'bo') # plot x and y using blue circle markers
>>> plot(y) # plot y using x as index array 0..N-1
>>> plot(y, 'r+') # ditto, but with red plusses
```

You can use `.Line2D`` properties as keyword arguments for more control on the appearance. Line properties and `*fmt*` can be mixed. The following two calls yield identical results:

```
>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
... linewidth=2, markersize=12)
```

When conflicting with `*fmt*`, keyword arguments take precedence.

## **\*\*Plotting labelled data\*\***

There's a convenient way for plotting objects with labelled data (i.e. data that can be accessed by index `obj['y']`). Instead of giving the data in `*x*` and `*y*`, you can provide the object in the `*data*` parameter and just give the labels for `*x*` and `*y*`:

```
>>> plot('xlabel', 'ylabel', data=obj)
```

All indexable objects are supported. This could e.g. be a ``dict``, a ``pandas.DataFrame`` or a structured numpy array.

## **\*\*Plotting multiple sets of data\*\***

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call ``plot`` multiple times.  
Example:

```
>>> plot(x1, y1, 'bo')
>>> plot(x2, y2, 'go')
```

- If `*x*` and/or `*y*` are 2D arrays, a separate data set will be drawn for every column. If both `*x*` and `*y*` are 2D, they must have the same shape. If only one of them is 2D with shape (N, m) the other must have length N and will be used for every data set m.

Example:

```
>>> x = [1, 2, 3]
>>> y = np.array([[1, 2], [3, 4], [5, 6]])
>>> plot(x, y)
```

is equivalent to:

```
>>> for col in range(y.shape[1]):
... plot(x, y[:, col])
```

- The third way is to specify multiple sets of `*[x]*`, `*y*`, `*[fmt]*`

groups::

```
>>> plot(x1, y1, 'g^', x2, y2, 'g-')
```

In this case, any additional keyword argument applies to all datasets. Also, this syntax cannot be combined with the *\*data\** parameter.

By default, each line is assigned a different style specified by a 'style cycle'. The *\*fmt\** and line property parameters are only necessary if you want explicit deviations from these defaults. Alternatively, you can also change the style cycle using `:rc:`axes.prop_cycle``.

## Parameters

-----

*x*, *y* : array-like or float

The horizontal / vertical coordinates of the data points.

*\*x\** values are optional and default to ```range(len(y))```.

Commonly, these parameters are 1D arrays.

They can also be scalars, or two-dimensional (in that case, the columns represent separate data sets).

These arguments cannot be passed as keywords.

*fmt* : str, optional

A format string, e.g. 'ro' for red circles. See the *\*Notes\** section for a full description of the format strings.

Format strings are just an abbreviation for quickly setting basic line properties. All of these and more can also be controlled by keyword arguments.

This argument cannot be passed as keyword.

*data* : indexable object, optional

An object with labelled data. If given, provide the label names to plot in *\*x\** and *\*y\**.

.. note::

Technically there's a slight ambiguity in calls where the second label is a valid *\*fmt\**. ```plot('n', 'o', data=obj)``` could be ```plt(x, y)``` or ```plt(y, fmt)```. In such cases, the former interpretation is chosen, but a warning is issued. You may suppress the warning by adding an empty format string ```plot('n', 'o', "", data=obj)```.

## Returns

-----

list of `.Line2D``

A list of lines representing the plotted data.

## Other Parameters

-----  
scalex, scaley : bool, default: True

These parameters determine if the view limits are adapted to the data limits. The values are passed on to  
`~.axes.Axes.autoscale\_view`.

**\*\*kwargs** : `~matplotlib.lines.Line2D` properties, optional  
**\*kwargs\*** are used to specify properties like a line label (for auto legends), linewidth, antialiasing, marker face color.  
Example::

```
>>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
>>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')
```

If you specify multiple lines with one plot call, the kwargs apply to all those lines. In case the label object is iterable, each element is used as labels for each set of data.

Here is a list of available `~.Line2D` properties:

Properties:

agg\_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None

animated: bool

antialiased or aa: bool

clip\_box: `~matplotlib.transforms.BboxBase` or None

clip\_on: bool

clip\_path: Patch or (Path, Transform) or None

color or c: :mpltype:`color`

dash\_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}

dash\_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}

dashes: sequence of floats (on/off ink in points) or (None, None)

data: (2, N) array or two 1D arrays

drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`

fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}

gapcolor: :mpltype:`color` or None

gid: str

in\_layout: bool

label: object

linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

linewidth or lw: float

marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`

markeredgecolor or mec: :mpltype:`color`

markeredgewidth or mew: float

markerfacecolor or mfc: :mpltype:`color`

markerfacecoloralt or mfcalt: :mpltype:`color`

markersize or ms: float

markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

mouseover: bool

path\_effects: list of `~.AbstractPathEffect`

picker: float or callable[[Artist, Event], tuple[bool, dict]]

pickradius: float

rasterized: bool  
sketch\_params: (scale: float, length: float, randomness: float)  
snap: bool or None  
solid\_capstyle: ``.CapStyle`` or `{'butt', 'projecting', 'round'}`  
solid\_joinstyle: ``.JoinStyle`` or `{'miter', 'round', 'bevel'}`  
transform: unknown  
url: str  
visible: bool  
xdata: 1D array  
ydata: 1D array  
zorder: float

See Also

-----

`scatter` : XY scatter plot with markers of varying size and/or color (sometimes also called bubble chart).

Notes

-----

.. note::

This is the :ref:`pyplot wrapper <pyplot\_interface>` for ``.axes.Axes.plot``.

**\*\*Format Strings\*\***

A format string consists of a part for color, marker and line::

`fmt = '[marker][line][color]'`

Each of them is optional. If not provided, the value from the style cycle is used. Exception: If ```line``` is given, but no ```marker```, the data will be a line without markers.

Other combinations such as ```[color][marker][line]``` are also supported, but note that their parsing may be ambiguous.

**\*\*Markers\*\***

=====

character description

=====

```.```` point marker  
```.```` pixel marker  
```o``` circle marker  
```v``` triangle\_down marker  
```^``` triangle\_up marker  
```<``` triangle\_left marker  
```>``` triangle\_right marker  
```1``` tri\_down marker  
```2``` tri\_up marker  
```3``` tri\_left marker  
```4``` tri\_right marker  
```8``` octagon marker  
```s``` square marker

```

`p` pentagon marker
`P` plus (filled) marker
`*` star marker
`h` hexagon1 marker
`H` hexagon2 marker
`+` plus marker
`x` x marker
`X` x (filled) marker
`D` diamond marker
`d` thin_diamond marker
`|` vline marker
`_` hline marker
=====

```

****Line Styles****

```

=====
character description
=====
`-` solid line style
`--` dashed line style
`-.` dash-dot line style
`:` dotted line style
=====

```

Example format strings::

```

'b' # blue markers with default shape
'or' # red circles
'g' # green solid line
'--' # dashed line with default color
'^k:' # black triangle_up markers connected by a dotted line

```

****Colors****

The supported color abbreviations are the single letter codes

```

=====
character color
=====
`b` blue
`g` green
`r` red
`c` cyan
`m` magenta
`y` yellow
`k` black
`w` white
=====

```

and the ``CN`` colors that index into the default property cycle.

If the color is the only part of the format string, you can additionally use any `matplotlib.colors` spec, e.g. full names` (``green``) or hex strings (``#008000``).

matplotlib.pyplot.plot_date

```
plot_date(x: 'ArrayLike', y: 'ArrayLike', fmt: 'str' = 'o', tz: 'str | datetime.tzinfo  
| None' = None, xdate: 'bool' = True, ydate: 'bool' = False, *, data=None, **kwargs)  
-> 'list[Line2D]'
```

[*Deprecated*] Plot coercing the axis to treat floats as dates.

.. deprecated:: 3.9

This method exists for historic reasons and will be removed in version 3.11.

- ``datetime``-like data should directly be plotted using
`~.Axes.plot`.
- If you need to plot plain numeric data as :ref:`date-format` or
need to set a timezone, call ``ax.xaxis.axis_date`` /
``ax.yaxis.axis_date`` before `~.Axes.plot`. See
`.Axis.axis_date`.

Similar to `.plot`, this plots *y* vs. *x* as lines or markers.
However, the axis labels are formatted as dates depending on *xdate*
and *ydate*. Note that `.plot` will work with `datetime` and
`numpy.datetime64` objects without resorting to this method.

Parameters

x, y : array-like

The coordinates of the data points. If *xdate* or *ydate* is
True, the respective values *x* or *y* are interpreted as
:ref:`Matplotlib dates <date-format>`.

fmt : str, optional

The plot format string. For details, see the corresponding
parameter in `.plot`.

tz : timezone string or `datetime.tzinfo`, default: :rc:`timezone`
The time zone to use in labeling dates.

xdate : bool, default: True

If *True*, the *x*-axis will be interpreted as Matplotlib dates.

ydate : bool, default: False

If *True*, the *y*-axis will be interpreted as Matplotlib dates.

Returns

list of `Line2D`

Objects representing the plotted data.

Other Parameters

data : indexable object, optional

If given, the following parameters also accept a string ``s``, which is
interpreted as ``data[s]`` if ``s`` is a key in ``data``:

`*x*, *y*`

`**kwargs`

Keyword arguments control the ``Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: ``~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`color` or `c`: `:mpltype:`color``

`dash_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`dash_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`dashes`: sequence of floats (on/off ink in points) or (None, None)

`data`: (2, N) array or two 1D arrays

`drawstyle` or `ds`: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'

`figure`: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``

`fillstyle`: {'full', 'left', 'right', 'bottom', 'top', 'none'}

`gapcolor`: `:mpltype:`color`` or None

`gid`: str

`in_layout`: bool

`label`: object

`linestyle` or `ls`: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}

`linewidth` or `lw`: float

`marker`: marker style string, ``~.path.Path`` or ``~.markers.MarkerStyle``

`markeredgecolor` or `mec`: `:mpltype:`color``

`markeredgewidth` or `mew`: float

`markerfacecolor` or `mfc`: `:mpltype:`color``

`markerfacecoloralt` or `mfcalt`: `:mpltype:`color``

`markersize` or `ms`: float

`markevery`: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]

`mouseover`: bool

`path_effects`: list of ``AbstractPathEffect``

`picker`: float or callable[[Artist, Event], tuple[bool, dict]]

`pickradius`: float

`rasterized`: bool

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`solid_capstyle`: ``CapStyle`` or {'butt', 'projecting', 'round'}

`solid_joinstyle`: ``JoinStyle`` or {'miter', 'round', 'bevel'}

`transform`: unknown

`url`: str

`visible`: bool

`xdata`: 1D array

`ydata`: 1D array

`zorder`: float

See Also

`matplotlib.dates` : Helper functions on dates.

`matplotlib.dates.date2num` : Convert dates to num.

`matplotlib.dates.num2date` : Convert num to dates.

`matplotlib.dates.drange` : Create an equally spaced sequence of dates.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.plot_date``.

If you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to `.plot_date``. `.plot_date`` will set the default tick locator to `.AutoDateLocator`` (if the tick locator is not already set to a `.DateLocator`` instance) and the default tick formatter to `.AutoDateFormatter`` (if the tick formatter is not already set to a `.DateFormatter`` instance).

.. deprecated:: 3.9
Use `plot` instead.

matplotlib.pyplot.polar

```
polar(*args, **kwargs) -> 'list[Line2D]'
```

Make a polar plot.

call signature::

`polar(theta, r, [fmt], **kwargs)`

This is a convenience wrapper around `.pyplot.plot``. It ensures that the current Axes is polar (or creates one if needed) and then passes all parameters to ``.pyplot.plot``.

.. note::

When making polar plots using the :ref:`pyplot API <pyplot_interface>`, ``.polar()`` should typically be the first command because that makes sure a polar Axes is created. Using other commands such as ``.plt.title()`` before this can lead to the implicit creation of a rectangular Axes, in which case a subsequent ``.polar()`` call will fail.

matplotlib.pyplot.prism

```
prism() -> 'None'
```

Set the colormap to 'prism'.

This changes the default colormap as well as the colormap of the current image if there is one. See ``.help(colormaps)`` for more information.

matplotlib.pyplot.psd

```
psd(x: 'ArrayLike', *, NFFT: 'int | None' = None, Fs: 'float | None' = None, Fc: 'int | None' = None, detrend: "Literal['none', 'mean', 'linear'] | Callable[[ArrayLike], ArrayLike] | None" = None, window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, noverlap: 'int | None' = None, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None, scale_by_freq: 'bool | None' = None, return_line: 'bool | None' = None, data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray] | tuple[np.ndarray, np.ndarray, Line2D]'
```

Plot the power spectral density.

The power spectral density :math:`P_{xx}` by Welch's average periodogram method. The vector x is divided into $NFFT$ length segments. Each segment is detrended by function `detrend` and windowed by function `window`. `noverlap` gives the length of the overlap between segments. The :math:`|\mathrm{fft}(i)|^2` of each segment :math:`i` are averaged to compute :math:`P_{xx}`, with a scaling to correct for power loss due to windowing.

If $\text{len}(x) < NFFT$, it will be zero padded to $NFFT$.

Parameters

`x` : 1-D array or sequence

Array or sequence containing the data

`Fs` : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

`window` : callable or ndarray, default: `~.window_hanning`

A function or a vector of length $NFFT$. To create window vectors see `~.window_hanning`, `~.window_none`, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

`sides` : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

`pad_to` : int, optional

The number of points to which the data segment is padded when performing the FFT. This can be different from $NFFT$, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to `~numpy.fft.fft`. The default is None, which sets `*pad_to*` equal to $NFFT$.

`NFFT` : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use `*pad_to*` for this instead.

`detrend` : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove

the mean or linear trend. Unlike in MATLAB, where the `*detrend*` parameter is a vector, in Matplotlib it is a function. The `:mod:`~matplotlib.mlab`` module defines `.detrend_none`, `.detrend_mean`, and `.detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: `'none'` calls `.detrend_none`, `'mean'` calls `.detrend_mean`, `'linear'` calls `.detrend_linear`.

`scale_by_freq` : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

`noverlap` : int, default: 0 (no overlap)

The number of points of overlap between segments.

`Fc` : int, default: 0

The center frequency of `*x*`, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

`return_line` : bool, default: False

Whether to include the line object plotted in the returned values.

Returns

`Pxx` : 1-D array

The values for the power spectrum `:math:P_{xx}` before scaling (real valued).

`freqs` : 1-D array

The frequencies corresponding to the elements in `*Pxx*`.

`line` : `~matplotlib.lines.Line2D`

The line created by this function.

Only returned if `*return_line*` is True.

Other Parameters

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`

`**kwargs`

Keyword arguments control the `.Line2D`` properties:

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased` or `aa`: bool

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

clip_on: bool
 clip_path: Patch or (Path, Transform) or None
 color or c: :mpltype:`color`
 dash_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}
 dash_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}
 dashes: sequence of floats (on/off ink in points) or (None, None)
 data: (2, N) array or two 1D arrays
 drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
 figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`
 fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
 gapcolor: :mpltype:`color` or None
 gid: str
 in_layout: bool
 label: object
 linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
 linewidth or lw: float
 marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
 markeredgewidth or mec: :mpltype:`color`
 markeredgewidth or mew: float
 markerfacecolor or mfc: :mpltype:`color`
 markerfacecoloralt or mfcalt: :mpltype:`color`
 markersize or ms: float
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]
 mouseover: bool
 path_effects: list of `~.AbstractPathEffect`
 picker: float or callable[[Artist, Event], tuple[bool, dict]]
 pickradius: float
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 solid_capstyle: `~.CapStyle` or {'butt', 'projecting', 'round'}
 solid_joinstyle: `~.JoinStyle` or {'miter', 'round', 'bevel'}
 transform: unknown
 url: str
 visible: bool
 xdata: 1D array
 ydata: 1D array
 zorder: float

See Also

specgram

Differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

magnitude_spectrum

Plots the magnitude spectrum.

csd

Plots the spectral density between two signals.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.psd`.

For plotting, the power is plotted as $10\log_{10}(P_{xx})$ for decibels, though `*Pxx*` itself is returned.

References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

matplotlib.pyplot.quiver

```
quiver(*args, data=None, **kwargs) -> 'Quiver'
```

Plot a 2D field of arrows.

Call signature::

`quiver([X, Y], U, V, [C], /, **kwargs)`

`*X*`, `*Y*` define the arrow locations, `*U*`, `*V*` define the arrow directions, and `*C*` optionally sets the color. The arguments `*X*`, `*Y*`, `*U*`, `*V*`, `*C*` are positional-only.

****Arrow length****

The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the `*scale*` and `*scale_units*` parameters.

****Arrow shape****

The arrow shape is determined by `*width*`, `*headwidth*`, `*headlength*` and `*headaxislength*`. See the notes below.

****Arrow styling****

Each arrow is internally represented by a filled polygon with a default edge linewidth of 0. As a result, an arrow is rather a filled area, not a line with a head, and `.PolyCollection` properties like `*linewidth*`, `*edgecolor*`, `*facecolor*`, etc. act accordingly.

Parameters

`X, Y` : 1D or 2D array-like, optional
The x and y coordinates of the arrow locations.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of `*U*` and `*V*`.

If `*X*` and `*Y*` are 1D but `*U*`, `*V*` are 2D, `*X*`, `*Y*` are expanded to 2D using ```X, Y = np.meshgrid(X, Y)```. In this case ```len(X)``` and ```len(Y)``` must match the column and row dimensions of `*U*` and `*V*`.

U, V : 1D or 2D array-like

The x and y direction components of the arrow vectors. The interpretation of these components (in data or in screen space) depends on `*angles*`.

`*U*` and `*V*` must have the same number of elements, matching the number of arrow locations in `*X*`, `*Y*`. `*U*` and `*V*` may be masked. Locations masked in any of `*U*`, `*V*`, and `*C*` will not be drawn.

C : 1D or 2D array-like, optional

Numeric data that defines the arrow colors by colormapping via `*norm*` and `*cmap*`.

This does not support explicit colors. If you want to set colors directly, use `*color*` instead. The size of `*C*` must match the number of arrow locations.

angles : {'uv', 'xy'} or array-like, default: 'uv'

Method for determining the angle of the arrows.

- 'uv': Arrow directions are based on

:ref:`display coordinates <coordinate-systems>`; i.e. a 45° angle will always show up as diagonal on the screen, irrespective of figure or Axes aspect ratio or Axes data ranges. This is useful when the arrows represent a quantity whose direction is not tied to the x and y data coordinates.

If `*U* == *V*` the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).

- 'xy': Arrow direction in data coordinates, i.e. the arrows point from

(x, y) to (x+u, y+v). This is ideal for vector fields or gradient plots where the arrows should directly represent movements or gradients in the x and y directions.

- Arbitrary angles may be specified explicitly as an array of values in degrees, counter-clockwise from the horizontal axis.

In this case `*U*`, `*V*` is only used to determine the length of the arrows.

For example, ```angles=[30, 60, 90]``` will orient the arrows at 30, 60, and 90 degrees respectively, regardless of the `*U*` and `*V*` components.

Note: inverting a data axis will correspondingly invert the arrows only with ```angles='xy'```.

pivot : {'tail', 'mid', 'middle', 'tip'}, default: 'tail'

The part of the arrow that is anchored to the `*X*`, `*Y*` grid. The arrow rotates about this point.

'mid' is a synonym for 'middle'.

scale : float, optional

Scales the length of the arrow inversely.

Number of data values represented by one unit of arrow length on the plot.
 For example, if the data represents velocity in meters per second (m/s), the scale parameter determines how many meters per second correspond to one unit of arrow length relative to the width of the plot.
 Smaller scale parameter makes the arrow longer.

By default, an autoscaling algorithm is used to scale the arrow length to a reasonable size, which is based on the average vector length and the number of vectors.

The arrow length unit is given by the `*scale_units*` parameter.

`scale_units` : {'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, default: 'width'

The physical image unit, which is used for rendering the scaled arrow data U^* , V^* .

The rendered arrow length is given by

length in x direction = $\frac{u}{\mathrm{scale}} \mathrm{scale_unit}$

length in y direction = $\frac{v}{\mathrm{scale}} \mathrm{scale_unit}$

For example, `((u, v) = (0.5, 0))` with `scale=10`, `scale_units="width"` results in a horizontal arrow with a length of $0.5 / 10 * \text{"width"}$, i.e. 0.05 times the Axes width.

Supported values are:

- 'width' or 'height': The arrow length is scaled relative to the width or height of the Axes.

For example, `((u, v) = (1, 0))` with `scale_units='width'`, `scale=1.0`, will result in an arrow length of width of the Axes.

- 'dots': The arrow length of the arrows is measured in display dots (pixels).

- 'inches': Arrow lengths are scaled based on the DPI (dots per inch) of the figure. This ensures that the arrows have a consistent physical size on the figure, in inches, regardless of data values or plot scaling.

For example, `((u, v) = (1, 0))` with `scale_units='inches'`, `scale=2` results in a 0.5 inch-long arrow.

- 'x' or 'y': The arrow length is scaled relative to the x or y axis units.

For example, `((u, v) = (0, 1))` with `scale_units='x'`, `scale=1` results in a vertical arrow with the length of 1 x-axis unit.

- 'xy': Arrow length will be same as 'x' or 'y' units.

This is useful for creating vectors in the x-y plane where u and v have the same units as x and y. To plot vectors in the x-y plane with u and v having the same units as x and y, use `angles='xy'`, `scale_units='xy'`, `scale=1`.

Note: Setting `*scale_units*` without setting `scale` does not have any effect because the scale units only differ by a constant factor and that is rescaled through autoscaling.

`units` : {'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, default: 'width'

Affects the arrow size (except for the length). In particular, the shaft **width** is measured in multiples of this unit.

Supported values are:

- 'width', 'height': The width or height of the Axes.
- 'dots', 'inches': Pixels or inches based on the figure dpi.
- 'x', 'y', 'xy': **X**, **Y** or `:math:\sqrt{X^2 + Y^2}` in data units.

The following table summarizes how these values affect the visible arrow size under zooming and figure size changes:

units	zoom	figure size change
'x', 'y', 'xy'	arrow size scales —	
'width', 'height'	—	arrow size scales
'dots', 'inches'	—	—

width : float, optional

Shaft width in arrow units. All head parameters are relative to **width**.

The default depends on choice of **units** above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

headwidth : float, default: 3

Head width as multiple of shaft **width**. See the notes below.

headlength : float, default: 5

Head length as multiple of shaft **width**. See the notes below.

headaxislength : float, default: 4.5

Head length at shaft intersection as multiple of shaft **width**. See the notes below.

minshaft : float, default: 1

Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible!

minlength : float, default: 1

Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead.

color : `:mplttype:'color'` or list `:mplttype:'color'`, optional

Explicit color(s) for the arrows. If **C** has been set, **color** has no effect.

This is a synonym for the ``.PolyCollection`` **facecolor** parameter.

Other Parameters

data : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

****kwargs** : `~matplotlib.collections.PolyCollection`` properties, optional
All other keyword arguments are passed on to `~.PolyCollection``:

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
alpha: array-like or float or None
animated: bool
antialiased or **aa** or **antialiaseds**: bool or list of bools
array: array-like or None
capstyle: `~.CapStyle`` or {'butt', 'projecting', 'round'}
clim: (vmin: float, vmax: float)
clip_box: `~matplotlib.transforms.BboxBase`` or None
clip_on: bool
clip_path: Patch or (Path, Transform) or None
cmap: `~.Colormap`` or str or None
color: `:mpltype:`color`` or list of RGBA tuples
edgecolor or **ec** or **edgecolors**: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'
facecolor or **facecolors** or **fc**: `:mpltype:`color`` or list of `:mpltype:`color``
figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``
gid: str
hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
hatch_linewidth: unknown
in_layout: bool
joinstyle: `~.JoinStyle`` or {'miter', 'round', 'bevel'}
label: object
linestyle or **dashes** or **linestyles** or **ls**: str or tuple or list thereof
linewidth or **linewidths** or **lw**: float or list of floats
mouseover: bool
norm: `~.Normalize`` or str or None
offset_transform or **transOffset**: `~.Transform``
offsets: (N, 2) or (2,) array-like
path_effects: list of `~.AbstractPathEffect``
paths: list of array-like
picker: None or bool or float or callable
pickradius: float
rasterized: bool
sizes: `~numpy.ndarray`` or None
sketch_params: (scale: float, length: float, randomness: float)
snap: bool or None
transform: `~matplotlib.transforms.Transform``
url: str
urls: list of str or None
verts: list of array-like
verts_and_codes: unknown
visible: bool
zorder: float

Returns

`~matplotlib.quiver.Quiver``

See Also

`.Axes.quiverkey` : Add a key to a quiver plot.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.quiver``.

****Arrow shape****

The arrow is drawn as a polygon using the nodes as shown below. The values `*headwidth*`, `*headlength*`, and `*headaxislength*` are in units of `*width*`.

.. image:: /_static/quiver_sizes.svg
:width: 500px

The defaults give a slightly swept-back arrow. Here are some guidelines how to get other head shapes:

- To make the head a triangle, make `*headaxislength*` the same as `*headlength*`.
- To make the arrow more pointed, reduce `*headwidth*` or increase `*headlength*` and `*headaxislength*`.
- To make the head smaller relative to the shaft, scale down all the head parameters proportionally.
- To remove the head completely, set all `*head*` parameters to 0.
- To get a diamond-shaped head, make `*headaxislength*` larger than `*headlength*`.
- Warning: For `*headaxislength*` < (`*headlength*` / `*headwidth*`), the "headaxis" nodes (i.e. the ones connecting the head with the shaft) will protrude out of the head in forward direction so that the arrow head looks broken.

matplotlib.pyplot.quiverkey

```
quiverkey(Q: 'Quiver', X: 'float', Y: 'float', U: 'float', label: 'str', **kwargs) -> 'QuiverKey'
```

Add a key to a quiver plot.

The positioning of the key depends on `*X*`, `*Y*`, `*coordinates*`, and `*labelpos*`. If `*labelpos*` is 'N' or 'S', `*X*`, `*Y*` give the position of the middle of the key arrow. If `*labelpos*` is 'E', `*X*`, `*Y*` positions the head, and if `*labelpos*` is 'W', `*X*`, `*Y*` positions the tail; in either of these two cases, `*X*`, `*Y*` is somewhere in the middle of the arrow+label key object.

Parameters

Q : `~matplotlib.quiver.Quiver``

A `~.Quiver`` object as returned by a call to `~.Axes.quiver()`.

X, Y : float

The location of the key.

U : float

The length of the key.

label : str

The key label (e.g., length and units of the key).

angle : float, default: 0

The angle of the key arrow, in degrees anti-clockwise from the horizontal axis.

coordinates : {'axes', 'figure', 'data', 'inches'}, default: 'axes'

Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with (0, 0) in the lower left and (1, 1) in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with (0, 0) at the lower left corner.

color : :mpltype:`color`

Overrides face and edge colors from *Q*.

labelpos : {'N', 'S', 'E', 'W'}

Position the label above, below, to the right, to the left of the arrow, respectively.

labelsep : float, default: 0.1

Distance in inches between the arrow and the label.

labelcolor : :mpltype:`color`, default: :rc:`text.color`

Label color.

fontproperties : dict, optional

A dictionary with keyword arguments accepted by the `~matplotlib.font_manager.FontProperties` initializer: *family*, *style*, *variant*, *size*, *weight*.`

zorder : float

The zorder of the key. The default is 0.1 above *Q*.

****kwargs**

Any additional keyword arguments are used to override vector properties taken from *Q*.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.quiverkey`.`

matplotlib.pyplot.rc

```
rc(group: 'str', **kwargs) -> 'None'
```

Set the current `.rcParams`.` *group* is the grouping for the rc, e.g., for ```lines.linewidth``` the group is ```lines```, for ```axes.facecolor```, the group is ```axes```, and so on. Group may also be a list or tuple of group names, e.g., `(*xtick*, *ytick*)`. *kwargs* is a dictionary attribute name/value pairs, e.g.,::

```
rc('lines', linewidth=2, color='r')
```

sets the current `.rcParams`` and is equivalent to::

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

```
=====
Alias Property
=====
'lw' 'linewidth'
'ls' 'linestyle'
'c' 'color'
'fc' 'facecolor'
'ec' 'edgecolor'
'mew' 'markeredgewidth'
'aa' 'antialiased'
=====
```

Thus you could abbreviate the above call as::

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows::

```
font = {'family' : 'monospace',
'weight' : 'bold',
'size' : 'larger'}
rc('font', **font) # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use ```matplotlib.style.use('default')``` or `:func:`~matplotlib.rcdefaults`` to restore the default ``rcParams`` after changes.

Notes

.. note::

This is equivalent to ``matplotlib.rc``.

Similar functionality is available by using the normal dict interface, i.e. ```rcParams.update({"lines.linewidth": 2, ...})``` (but ```rcParams.update``` does not support abbreviations or grouping).

matplotlib.pyplot.rc_context

```
rc_context(rc: 'dict[str, Any] | None' = None, fname: 'str | pathlib.Path |
os.PathLike | None' = None) -> 'AbstractContextManager[None]'
```

Return a context manager for temporarily changing rcParams.

The `:rc:`backend`` will not be reset by the context manager.

rcParams changed both through the context manager invocation and in the body of the context will be reset on context exit.

Parameters

rc : dict

The rcParams to temporarily set.

fname : str or path-like

A file with Matplotlib rc settings. If both *fname* and *rc* are given, settings from *rc* take precedence.

See Also

:ref:`customizing-with-matplotlibrc-files`

Notes

.. note::

This is equivalent to `matplotlib.rc_context``.

Examples

Passing explicit values via a dict::

```
with mpl.rc_context({'interactive': False}):
    fig, ax = plt.subplots()
    ax.plot(range(3), range(3))
    fig.savefig('example.png')
    plt.close(fig)
```

Loading settings from a file::

```
with mpl.rc_context(fname='print.rc'):
    plt.plot(x, y) # uses 'print.rc'
```

Setting in the context body::

```
with mpl.rc_context():
    # will be reset
    mpl.rcParams['lines.linewidth'] = 5
    plt.plot(x, y)
```

matplotlib.pyplot.rcdefaults

```
rcdefaults() -> 'None'
```

Restore the `matplotlib.rcParams`` from Matplotlib's internal default style.

Style-blacklisted `matplotlib.rcParams`` (defined in `matplotlib.style.core.STYLE_BLACKLIST``) are not updated.

See Also

`matplotlib.rc_file_defaults`

Restore the `matplotlib.rcParams`` from the rc file originally loaded by Matplotlib.

matplotlib.style.use
Use a specific style file. Call ``style.use('default')`` to restore the default style.

Notes

.. note::

This is equivalent to `matplotlib.rcdefaults`.

matplotlib.pyplot.rgrids

```
rgrids(radii: 'ArrayLike | None' = None, labels: 'Sequence[str | Text] | None' = None,
angle: 'float | None' = None, fmt: 'str | None' = None, **kwargs) ->
'tuple[list[Line2D], list[Text]]'
```

Get or set the radial gridlines on the current polar plot.

Call signatures::

lines, labels = rgrids()

lines, labels = rgrids(radii, labels=None, angle=22.5, fmt=None, **kwargs)

When called with no arguments, `rgrids` simply returns the tuple (*lines*, *labels*). When called with arguments, the labels will appear at the specified radial distances and angle.

Parameters

radii : tuple with floats

The radii for the radial gridlines

labels : tuple with strings or None

The labels to use at each radial gridline. The `matplotlib.ticker.ScalarFormatter` will be used if None.

angle : float

The angular position of the radius labels in degrees.

fmt : str or None

Format string used in `matplotlib.ticker.FormatStrFormatter`. For example '%f'.

Returns

lines : list of `matplotlib.lines.Line2D`

The radial gridlines.

labels : list of `matplotlib.text.Text`

The tick labels.

Other Parameters

**kwargs

`*kwargs*` are optional `.Text`` properties for the labels.

See Also

```
-----  
.pyplot.thetagrids  
.projections.polar.PolarAxes.set_rgrids  
.Axis.get_gridlines  
.Axis.get_ticklabels
```

Examples

```
-----  
::  
  
# set the locations of the radial gridlines  
lines, labels = rgrids( (0.25, 0.5, 1.0) )  
  
# set the locations and labels of the radial gridlines  
lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry' ))
```

matplotlib.pyplot.savefig

```
savefig(*args, **kwargs) -> 'None'
```

Save the current figure as an image or vector graphic to a file.

Call signature::

```
savefig(fname, *, transparent=None, dpi='figure', format=None,  
metadata=None, bbox_inches=None, pad_inches=0.1,  
facecolor='auto', edgecolor='auto', backend=None,  
**kwargs  
)
```

The available output formats depend on the backend being used.

Parameters

```
-----  
fname : str or path-like or binary file-like  
A path, or a Python file-like object, or  
possibly some backend-dependent object such as  
`matplotlib.backends.backend_pdf.PdfPages`.
```

If `*format*` is set, it determines the output format, and the file is saved as `*fname*`. Note that `*fname*` is used verbatim, and there is no attempt to make the extension, if any, of `*fname*` match `*format*`, and no extension is appended.

If `*format*` is not set, then the format is inferred from the extension of `*fname*`, if there is one. If `*format*` is not set and `*fname*` has no extension, then the file is saved with `:rc:`savefig.format`` and the appropriate extension is appended to `*fname*`.

Other Parameters

transparent : bool, default: :rc:`savefig.transparent`
If **True**, the Axes patches will all be transparent; the Figure patch will also be transparent unless **facecolor** and/or **edgecolor** are specified via kwargs.

If **False** has no effect and the color of the Axes and Figure patches are unchanged (unless the Figure patch is specified via the **facecolor** and/or **edgecolor** keyword arguments in which case those colors are used).

The transparency of these patches will be restored to their original values upon exit of this function.

This is useful, for example, for displaying a plot on top of a colored background on a web page.

dpi : float or 'figure', default: :rc:`savefig.dpi`
The resolution in dots per inch. If 'figure', use the figure's dpi value.

format : str
The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under **fname**.

metadata : dict, optional
Key/value pairs to store in the image metadata. The supported keys and defaults depend on the image format and backend:

- 'png' with Agg backend: See the parameter ``metadata`` of `~.FigureCanvasAgg.print_png``.
- 'pdf' with pdf backend: See the parameter ``metadata`` of `~.backend_pdf.PdfPages``.
- 'svg' with svg backend: See the parameter ``metadata`` of `~.FigureCanvasSVG.print_svg``.
- 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

Not supported for 'pgf', 'raw', and 'rgba' as those formats do not support embedding metadata.

Does not currently support 'jpg', 'tiff', or 'webp', but may include embedding EXIF metadata in the future.

bbox_inches : str or `~.Bbox``, default: :rc:`savefig.bbox`
Bounding box in inches: only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

pad_inches : float or 'layout', default: :rc:`savefig.pad_inches`
Amount of padding in inches around the figure when `bbox_inches`` is 'tight'. If 'layout' use the padding from the constrained or compressed layout engine; ignored if one of those engines is not in use.

facecolor : `:mpltype:`color`` or 'auto', default: :rc:`savefig.facecolor`
The facecolor of the figure. If 'auto', use the current figure facecolor.

`edgecolor` : `mpltype:~color` or 'auto'`, default: `:rc:~savefig.edgecolor``
The edgecolor of the figure. If 'auto', use the current figure edgecolor.

`backend` : str, optional

Use a non-default backend to render the file, e.g. to render a png file with the "cairo" backend rather than the default "agg", or a pdf file with the "pgf" backend rather than the default "pdf". Note that the default backend is normally sufficient. See [:ref:~the-builtin-backends`](#) for a list of valid backends for each file format. Custom backends can be referenced as "module://...".

`orientation` : {'landscape', 'portrait'}

Currently only supported by the postscript backend.

`papertype` : str

One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

`bbox_extra_artists` : list of `~matplotlib.artist.Artist``, optional

A list of extra artists that will be considered when the tight bbox is calculated.

`pil_kwargs` : dict, optional

Additional keyword arguments that are passed to ``PIL.Image.Image.save`` when saving the figure.

Notes

.. note::

This is the `:ref:~pyplot wrapper <pyplot_interface>`` for ``.Figure.savefig``.

matplotlib.pyplot.sca

```
sca(ax: 'Axes') -> 'None'
```

Set the current Axes to *ax* and the current Figure to the parent of *ax*.

matplotlib.pyplot.scatter

```
scatter(x: 'float | ArrayLike', y: 'float | ArrayLike', s: 'float | ArrayLike | None' = None, c: 'ArrayLike | Sequence[ColorType] | ColorType | None' = None, *, marker: 'MarkerType | None' = None, cmap: 'str | Colormap | None' = None, norm: 'str | Normalize | None' = None, vmin: 'float | None' = None, vmax: 'float | None' = None, alpha: 'float | None' = None, linewidths: 'float | Sequence[float] | None' = None, edgecolors: "Literal['face', 'none'] | ColorType | Sequence[ColorType] | None" = None, colorizer: 'Colorizer | None' = None, plotnonfinite: 'bool' = False, data=None, **kwargs) -> 'PathCollection'
```

A scatter plot of *y* vs. *x* with varying marker size and/or color.

Parameters

`x, y` : float or array-like, shape `(n,)`
The data positions.

`s` : float or array-like, shape `(n,)`, optional
The marker size in points**2 (typographic points are 1/72 in.).
Default is `rcParams['lines.markersize'] ** 2`.

The linewidth and edgecolor can visually interact with the marker size, and can lead to artifacts if the marker size is smaller than the linewidth.

If the linewidth is greater than 0 and the edgecolor is anything but `'none'`, then the effective size of the marker will be increased by half the linewidth because the stroke will be centered on the edge of the shape.

To eliminate the marker edge either set `*linewidth=0` or `*edgecolor='none'`.

`c` : array-like or list of :mpltype:`color` or :mpltype:`color`, optional
The marker colors. Possible values:

- A scalar or sequence of `n` numbers to be mapped to colors using `*cmap*` and `*norm*`.
- A 2D array in which the rows are RGB or RGBA.
- A sequence of colors of length `n`.
- A single color format string.

Note that `*c*` should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. If you want to specify the same RGB or RGBA value for all points, use a 2D array with a single row. Otherwise, value-matching will have precedence in case of a size matching with `*x*` and `*y*`.

If you wish to specify a single color for all points prefer the `*color*` keyword argument.

Defaults to ``None``. In that case the marker color is determined by the value of `*color*`, `*facecolor*` or `*facecolors*`. In case those are not specified or ``None``, the marker color is determined by the next color of the ```Axes``` current "shape and fill" color cycle. This cycle defaults to `:rc:`axes.prop_cycle``.

`marker` : `~.markers.MarkerStyle``, default: `:rc:`scatter.marker``
The marker style. `*marker*` can be either an instance of the class or the text shorthand for a particular marker.
See `:mod:`matplotlib.markers`` for more information about marker styles.

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``
The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if **c** is RGB(A).

norm : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using **cmap**. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see :ref:`colormapnorms`).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if **c** is RGB(A).

vmin, *vmax* : float, optional

When using scalar data and no explicit **norm**, **vmin** and **vmax** define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use **vmin*/vmax** when a **norm** instance is given (but using a `str`` **norm** name together with **vmin*/vmax** is acceptable).

This parameter is ignored if **c** is RGB(A).

alpha : float, default: None

The alpha blending value, between 0 (transparent) and 1 (opaque).

linewidths : float or array-like, default: `:rc:`lines.linewidth``

The linewidth of the marker edges. Note: The default **edgecolors** is 'face'. You may want to change this as well.

edgecolors : {'face', 'none', *None*} or :mpltype:`color` or list of :mpltype:`color`, default: `:rc:`scatter.edgecolors``

The edge color of the marker. Possible values:

- 'face': The edge color will always be the same as the face color.
- 'none': No patch boundary will be drawn.
- A color or sequence of colors.

For non-filled markers, **edgecolors** is ignored. Instead, the color is determined like with 'face', i.e. from **c**, **colors**, or **facecolors**.

colorizer : `~matplotlib.colorbar.Colorizer`` or None, default: None

The Colorizer object used to map color to data. If None, a Colorizer object is created from a **norm** and **cmap**.

This parameter is ignored if **c** is RGB(A).

plotnonfinite : bool, default: False

Whether to plot points with nonfinite **c** (i.e. ```inf```, ```-inf``` or ```nan```). If ```True``` the points are drawn with the **bad**

colormap color (see `Colormap.set_bad`).

Returns

`~matplotlib.collections.PathCollection``

Other Parameters

data : indexable object, optional

If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`:

`*x*, *y*, *s*, *linewidths*, *edgecolors*, *c*, *facecolor*, *facecolors*, *color*`

`**kwargs : ~matplotlib.collections.PathCollection` properties`

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: array-like or float or None

`animated`: bool

`antialiased` or `aa` or `antialiaseds`: bool or list of bools

`array`: array-like or None

`capstyle`: `~.CapStyle`` or `{'butt', 'projecting', 'round'}`

`clim`: (vmin: float, vmax: float)

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`cmap`: `~.Colormap`` or str or None

`color`: `:mpltype:`color`` or list of RGBA tuples

`edgecolor` or `ec` or `edgecolors`: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'

`facecolor` or `facecolors` or `fc`: `:mpltype:`color`` or list of `:mpltype:`color``

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

`gid`: str

`hatch`: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`

`hatch_linewidth`: unknown

`in_layout`: bool

`joinstyle`: `~.JoinStyle`` or `{'miter', 'round', 'bevel'}`

`label`: object

`linestyle` or `dashes` or `linestyles` or `ls`: str or tuple or list thereof

`linewidth` or `linewidths` or `lw`: float or list of floats

`mouseover`: bool

`norm`: `~.Normalize`` or str or None

`offset_transform` or `transOffset`: `~.Transform``

`offsets`: (N, 2) or (2,) array-like

`path_effects`: list of `~.AbstractPathEffect``

`paths`: unknown

`picker`: None or bool or float or callable

`pickradius`: float

`rasterized`: bool

`sizes`: `~numpy.ndarray`` or None

`sketch_params`: (scale: float, length: float, randomness: float)

`snap`: bool or None

`transform`: `~matplotlib.transforms.Transform``

`url`: str

`urls`: list of str or None

`visible`: bool

zorder: float

See Also

plot : To plot scatter plots when markers are identical in size and color.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.scatter``.

* The ``plot`` function will be faster for scatterplots where markers don't vary in size or color.

* Any or all of `*x*`, `*y*`, `*s*`, and `*c*` may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.

* Fundamentally, scatter works with 1D arrays; `*x*`, `*y*`, `*s*`, and `*c*` may be input as N-D arrays, but within scatter they will be flattened. The exception is `*c*`, which will be flattened only if its size matches the size of `*x*` and `*y*`.

matplotlib.pyplot.sci

```
sci(im: 'ColorizingArtist') -> 'None'
```

Set the current image.

This image will be the target of colormap functions like ``pyplot.viridis``, and other functions such as ``~.pyplot.clim``. The current image is an attribute of the current Axes.

matplotlib.pyplot.semilogx

```
semilogx(*args, **kwargs) -> 'list[Line2D]'
```

Make a plot with log scaling on the x-axis.

Call signatures::

```
semilogx([x], y, [fmt], data=None, **kwargs)
semilogx([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around ``plot`` which additionally changes the x-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters `*base*`, `*subs*`, and `*nonpositive*` control the x-axis properties. They are just forwarded to ``Axes.set_xscale``.

Parameters

base : float, default: 10

Base of the x logarithm.

subs : array-like, optional

The location of the minor xticks. If **None**, reasonable locations are automatically chosen depending on the number of decades in the plot. See ``Axes.set_xscale`` for details.

nonpositive : {'mask', 'clip'}, default: 'clip'

Non-positive values in x can be masked as invalid, or clipped to a very small positive number.

****kwargs**

All parameters supported by ``plot``.

Returns

list of ``Line2D``

Objects representing the plotted data.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.semilogx``.

matplotlib.pyplot.semilogy

```
semilogy(*args, **kwargs) -> 'list[Line2D]'
```

Make a plot with log scaling on the y-axis.

Call signatures::

`semilogy([x], y, [fmt], data=None, **kwargs)`

`semilogy([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)`

This is just a thin wrapper around ``plot`` which additionally changes the y-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters **base**, **subs**, and **nonpositive** control the y-axis properties. They are just forwarded to ``Axes.set_yscale``.

Parameters

base : float, default: 10

Base of the y logarithm.

subs : array-like, optional

The location of the minor yticks. If **None**, reasonable locations

are automatically chosen depending on the number of decades in the plot. See ``Axes.set_yscale`` for details.

nonpositive : {'mask', 'clip'}, default: 'clip'

Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

****kwargs**

All parameters supported by ``plot``.

Returns

list of ``Line2D``

Objects representing the plotted data.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.semilogy``.

matplotlib.pyplot.set_cmap

```
set_cmap(cmap: 'Colormap | str') -> 'None'
```

Set the default colormap, and applies it to the current image if any.

Parameters

cmap : ``~matplotlib.colors.Colormap`` or str

A colormap instance or the name of a registered colormap.

See Also

colormaps

get_cmap

matplotlib.pyplot.set_loglevel

```
set_loglevel(*args, **kwargs) -> 'None'
```

Configure Matplotlib's logging levels.

Matplotlib uses the standard library ``logging`` framework under the root logger 'matplotlib'. This is a helper function to:

- set Matplotlib's root logger level
- set the root logger handler's level, creating the handler if it does not exist yet

Typically, one should call ``set_loglevel("info")`` or ``set_loglevel("debug")`` to get additional debugging information.

Users or applications that are installing their own logging handlers may want to directly manipulate `logging.getLogger('matplotlib')` rather than use this function.

Parameters

level : {"notset", "debug", "info", "warning", "error", "critical"}
The log level of the handler.

Notes

.. note::

This is equivalent to `matplotlib.set_loglevel``.

The first time this function is called, an additional handler is attached to Matplotlib's root handler; this handler is reused every time and this function simply manipulates the logger and handler's level.

matplotlib.pyplot.setp

```
setp(obj, *args, **kwargs)
```

Set one or more properties on an `Artist``, or list allowed values.

Parameters

obj : `matplotlib.artist.Artist`` or list of `Artist``
The artist(s) whose properties are being set or queried. When setting properties, all artists are affected; when querying the allowed values, only the first instance in the sequence is queried.

For example, two lines can be made thicker and red with a single call:

```
>>> x = arange(0, 1, 0.01)
>>> lines = plot(x, sin(2*pi*x), x, sin(4*pi*x))
>>> setp(lines, linewidth=2, color='r')
```

file : file-like, default: `sys.stdout``

Where `setp`` writes its output when asked to list allowed values.

```
>>> with open('output.log') as file:
...     setp(line, file=file)
```

The default, `None``, means `sys.stdout``.

*args, **kwargs

The properties to set. The following combinations are supported:

- Set the linestyle of a line to be dashed:

```
>>> line, = plot([1, 2, 3])
>>> setp(line, linestyle='--')
```

- Set multiple properties at once:

```
>>> setp(line, linewidth=2, color='r')
```

- List allowed values for a line's linestyle:

```
>>> setp(line, 'linestyle')
linestyle: {'-', '--', '-.', ':', ' ', (offset, on-off-seq), ...}
```

- List all properties that can be set, and their allowed values:

```
>>> setp(line)
agg_filter: a filter function, ...
[long output listing omitted]
```

`setp` also supports MATLAB style string/value pairs. For example, the following are equivalent:

```
>>> setp(lines, 'linewidth', 2, 'color', 'r') # MATLAB style
>>> setp(lines, linewidth=2, color='r') # Python style
```

See Also

getp

Notes

.. note::

This is equivalent to `matplotlib.artist.setp`.

matplotlib.pyplot.show

```
show(close=None, block=None)
```

Display all open figures.

Parameters

block : bool, optional

Whether to wait for all figures to be closed before returning.

If `True` block and run the GUI main loop until all figure windows are closed.

If `False` ensure that all figure windows are displayed and return immediately. In this case, you are responsible for ensuring that the event loop is running to have responsive figures.

Defaults to True in non-interactive mode and to False in interactive mode (see `pyplot.isinteractive`).

See Also

ion : Enable interactive mode, which shows / updates the figure after every plotting command, so that calling ``show()`` is not necessary.
ioff : Disable interactive mode.
savefig : Save the figure to an image file instead of showing it on screen.

Notes

Saving figures to file and showing a window at the same time

If you want an image file as well as a user interface window, use `.pyplot.savefig`` before `.pyplot.show``. At the end of (a blocking) `show()` the figure is closed and thus unregistered from pyplot. Calling `.pyplot.savefig`` afterwards would save a new and thus empty figure. This limitation of command order does not apply if the show is non-blocking or if you keep a reference to the figure and use `.Figure.savefig``.

Auto-show in jupyter notebooks

The jupyter backends (activated via ```%matplotlib inline```, ```%matplotlib notebook```, or ```%matplotlib widget```), call `show()` at the end of every cell by default. Thus, you usually don't have to call it explicitly there.

matplotlib.pyplot.specgram

```
specgram(x: 'ArrayLike', *, NFFT: 'int | None' = None, Fs: 'float | None' = None, Fc: 'int | None' = None, detrend: "Literal['none', 'mean', 'linear'] | Callable[[ArrayLike], ArrayLike] | None" = None, window: 'Callable[[ArrayLike], ArrayLike] | ArrayLike | None' = None, noverlap: 'int | None' = None, cmap: 'str | Colormap | None' = None, xextent: 'tuple[float, float] | None' = None, pad_to: 'int | None' = None, sides: "Literal['default', 'onesided', 'twosided'] | None" = None, scale_by_freq: 'bool | None' = None, mode: "Literal['default', 'psd', 'magnitude', 'angle', 'phase'] | None" = None, scale: "Literal['default', 'linear', 'dB'] | None" = None, vmin: 'float | None' = None, vmax: 'float | None' = None, data=None, **kwargs)
-> 'tuple[np.ndarray, np.ndarray, np.ndarray, AxesImage]'
```

Plot a spectrogram.

Compute and plot a spectrogram of data in `*x*`. Data are split into `*NFFT*` length segments and the spectrum of each section is computed. The windowing function `*window*` is applied to each segment, and the amount of overlap of each segment is specified with `*noverlap*`. The spectrogram is plotted as a colormap (using `imshow`).

Parameters

`x` : 1-D array or sequence
Array or sequence containing the data.

`Fs` : float, default: 2

The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, `*freqs*`, in cycles per time unit.

window : callable or ndarray, default: ``.window_hanning``

A function or a vector of length `*NFFT*`. To create window vectors see ``.window_hanning``, ``.window_none``, ``numpy.blackman``, ``numpy.hamming``, ``numpy.bartlett``, ``scipy.signal``, ``scipy.signal.get_window``, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

sides : {'default', 'onesided', 'twosided'}, optional

Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

pad_to : int, optional

The number of points to which the data segment is padded when performing the FFT. This can be different from `*NFFT*`, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the `*n*` parameter in the call to `~numpy.fft.fft``. The default is None, which sets `*pad_to*` equal to `*NFFT*`

NFFT : int, default: 256

The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use `*pad_to*` for this instead.

detrend : {'none', 'mean', 'linear'} or callable, default: 'none'

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the `*detrend*` parameter is a vector, in Matplotlib it is a function. The `:mod:`~matplotlib.mlab`` module defines ``.detrend_none``, ``.detrend_mean``, and ``.detrend_linear``, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls ``.detrend_none``. 'mean' calls ``.detrend_mean``. 'linear' calls ``.detrend_linear``.

scale_by_freq : bool, default: True

Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

mode : {'default', 'psd', 'magnitude', 'angle', 'phase'}

What sort of spectrum to use. Default is 'psd', which takes the power spectral density. 'magnitude' returns the magnitude spectrum. 'angle' returns the phase spectrum without unwrapping. 'phase' returns the phase spectrum with unwrapping.

noverlap : int, default: 128

The number of points of overlap between blocks.

scale : {'default', 'linear', 'dB'}

The scaling of the values in the `*spec*`. 'linear' is no scaling. 'dB' returns the values in dB scale. When `*mode*` is 'psd', this is dB power ($10 * \log_{10}$). Otherwise, this is dB amplitude ($20 * \log_{10}$). 'default' is 'dB' if `*mode*` is 'psd' or

'magnitude' and 'linear' otherwise. This must be 'linear' if **mode** is 'angle' or 'phase'.

Fc : int, default: 0

The center frequency of **x**, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

cmap : `~.Colormap`, default: `:rc:~image.cmap`

xextent : **None** or (xmin, xmax)

The image extent along the x-axis. The default sets **xmin** to the left border of the first bin (**spectrum** column) and **xmax** to the right border of the last bin. Note that for **noverlap>0** the width of the bins is smaller than those of the segments.

data : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

x

vmin, *vmax* : float, optional

vmin and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the data.

****kwargs**

Additional keyword arguments are passed on to `~.axes.Axes.imshow` which makes the spectrogram image. The origin keyword argument is not supported.

Returns

spectrum : 2D array

Columns are the periodograms of successive segments.

freqs : 1-D array

The frequencies corresponding to the rows in **spectrum**.

t : 1-D array

The times corresponding to midpoints of segments (i.e., the columns in **spectrum**).

im : `~.AxesImage`

The image created by `imshow` containing the spectrogram.

See Also

psd

Differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

magnitude_spectrum

A single spectrum, similar to having a single segment when **mode**

is 'magnitude'. Plots a line instead of a colormap.

angle_spectrum

A single spectrum, similar to having a single segment when *mode* is 'angle'. Plots a line instead of a colormap.

phase_spectrum

A single spectrum, similar to having a single segment when *mode* is 'phase'. Plots a line instead of a colormap.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.specgram``.

The parameters *detrend* and *scale_by_freq* do only apply when *mode* is set to 'psd'.

matplotlib.pyplot.spring

```
spring() -> 'None'
```

Set the colormap to 'spring'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

matplotlib.pyplot.spy

```
spy(Z: 'ArrayLike', *, precision: "float | Literal['present']" = 0, marker: 'str | None' = None, markersize: 'float | None' = None, aspect: "Literal['equal', 'auto'] | float | None" = 'equal', origin: "Literal['upper', 'lower']" = 'upper', **kwargs) -> 'AxesImage'
```

Plot the sparsity pattern of a 2D array.

This visualizes the non-zero values of the array.

Two plotting styles are available: image and marker. Both are available for full arrays, but only the marker style works for `~.sparse.spmatrix`` instances.

****Image style****

If *marker* and *markersize* are *None*, `~.Axes.imshow`` is used. Any extra remaining keyword arguments are passed to this method.

****Marker style****

If *Z* is a `~.sparse.spmatrix`` or *marker* or *markersize* are *None*, a `~.Line2D`` object will be returned with the value of marker determining the marker type, and any remaining keyword arguments passed to `~.Axes.plot``.

Parameters

Z : (M, N) array-like

The array to be plotted.

precision : float or 'present', default: 0

If **precision** is 0, any non-zero value will be plotted. Otherwise, values of $|Z| > \text{precision}$ will be plotted.

For `scipy.sparse.spmatrix` instances, you can also pass 'present'. In this case any value present in the array will be plotted, even if it is identically zero.

aspect : {'equal', 'auto', None} or float, default: 'equal'

The aspect ratio of the Axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling `~.Axes.set_aspect``. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square.
- 'auto': The Axes is kept fixed and the aspect is adjusted so that the data fit in the Axes. In general, this will result in non-square pixels.
- **None**: Use `~rc:`image.aspect``.

origin : {'upper', 'lower'}, default: `~rc:`image.origin``

Place the [0, 0] index of the array in the upper left or lower left corner of the Axes. The convention 'upper' is typically used for matrices and images.

Returns

`~matplotlib.image.AxesImage`` or `~.Line2D``

The return type depends on the plotting style (see above).

Other Parameters

****kwargs**

The supported additional parameters depend on the plotting style.

For the image style, you can pass the following additional parameters of `~.Axes.imshow``:

- **cmap**
- **alpha**
- **url**
- any `~.Artist`` properties (passed on to the `~.AxesImage``)

For the marker style, you can pass any `~.Line2D`` property except for **linestyle**:

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None
 animated: bool
 antialiased or aa: bool
 clip_box: `~matplotlib.transforms.BboxBase`` or None
 clip_on: bool
 clip_path: Patch or (Path, Transform) or None
 color or c: `:mpltype:`color``
 dash_capstyle: ``.CapStyle`` or {'butt', 'projecting', 'round'}
 dash_joinstyle: ``.JoinStyle`` or {'miter', 'round', 'bevel'}
 dashes: sequence of floats (on/off ink in points) or (None, None)
 data: (2, N) array or two 1D arrays
 drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
 figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``
 fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
 gapcolor: `:mpltype:`color`` or None
 gid: str
 in_layout: bool
 label: object
 linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
 linewidth or lw: float
 marker: marker style string, `~.path.Path`` or `~.markers.MarkerStyle``
 markeredgewidth or mec: `:mpltype:`color``
 markeredgewidth or mew: float
 markerfacecolor or mfc: `:mpltype:`color``
 markerfacecoloralt or mfcalt: `:mpltype:`color``
 markersize or ms: float
 markevery: None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]
 mouseover: bool
 path_effects: list of ``.AbstractPathEffect``
 picker: float or callable[[Artist, Event], tuple[bool, dict]]
 pickradius: float
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 solid_capstyle: ``.CapStyle`` or {'butt', 'projecting', 'round'}
 solid_joinstyle: ``.JoinStyle`` or {'miter', 'round', 'bevel'}
 transform: unknown
 url: str
 visible: bool
 xdata: 1D array
 ydata: 1D array
 zorder: float

Notes

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for `~.axes.Axes.spy``.

matplotlib.pyplot.stackplot

```
stackplot(x, *args, labels=(), colors=None, hatch=None, baseline='zero', data=None,
**kwargs)
```

Draw a stacked area plot or a streamgraph.

Parameters

`x` : (N,) array-like

`y` : (M, N) array-like

The data can be either stacked or unstacked. Each of the following calls is legal::

`stackplot(x, y)` # where `y` has shape (M, N) e.g. `y = [y1, y2, y3, y4]`

`stackplot(x, y1, y2, y3, y4)` # where `y1, y2, y3, y4` have length N

`baseline` : {'zero', 'sym', 'wiggle', 'weighted_wiggle'}

Method used to calculate the baseline:

- `''zero''`: Constant zero baseline, i.e. a simple stacked plot.

- `''sym''`: Symmetric around zero and is sometimes called 'ThemeRiver'.

- `''wiggle''`: Minimizes the sum of the squared slopes.

- `''weighted_wiggle''`: Does the same but weights to account for size of each layer. It is also called 'Streamgraph'-layout. More details can be found at <http://leebyron.com/streamgraph/>.

`labels` : list of str, optional

A sequence of labels to assign to each data series. If unspecified, then no labels will be applied to artists.

`colors` : list of :mpltype:`color`, optional

A sequence of colors to be cycled through and used to color the stacked areas. The sequence need not be exactly the same length as the number of provided `*y*`, in which case the colors will repeat from the beginning.

If not specified, the colors from the Axes property cycle will be used.

`hatch` : list of str, default: None

A sequence of hatching styles. See

:doc:`/gallery/shapes_and_collections/hatch_style_reference`.

The sequence will be cycled through for filling the stacked areas from bottom to top.

It need not be exactly the same length as the number of provided `*y*`, in which case the styles will repeat from the beginning.

.. versionadded:: 3.9

Support for list input

`data` : indexable object, optional

If given, all parameters also accept a string `''s''`, which is interpreted as `''data[s]''` if `''s''` is a key in `''data''`.

****kwargs**

All other keyword arguments are passed to ``.Axes.fill_between``.

Returns

list of ``PolyCollection``
A list of ``PolyCollection`` instances, one for each element in the stacked area plot.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.stackplot``.

matplotlib.pyplot.stairs

```
stairs(values: 'ArrayLike', edges: 'ArrayLike | None' = None, *, orientation:
"Literal['vertical', 'horizontal']" = 'vertical', baseline: 'float | ArrayLike | None'
= 0, fill: 'bool' = False, data=None, **kwargs) -> 'StepPatch'
```

Draw a stepwise constant function as a line or a filled plot.

`*edges*` define the x-axis positions of the steps. `*values*` the function values between these steps. Depending on `*fill*`, the function is drawn either as a continuous line with vertical segments at the edges, or as a filled area.

Parameters

values : array-like
The step heights.

edges : array-like
The step positions, with `len(edges) == len(vals) + 1`,
between which the curve takes on vals values.

orientation : {'vertical', 'horizontal'}, default: 'vertical'
The direction of the steps. Vertical means that `*values*` are along the y-axis, and edges are along the x-axis.

baseline : float, array-like or None, default: 0
The bottom value of the bounding edges or when
`fill=True`, position of lower edge. If `*fill*` is
True or an array is passed to `*baseline*`, a closed
path is drawn.

If None, then drawn as an unclosed Path.

fill : bool, default: False
Whether the area under the step curve should be filled.

Passing both `fill=True` and `baseline=None` will likely result in undesired filling: the first and last points will be connected with a straight line and the fill will be between this line and the stairs.

Returns

StepPatch : `~matplotlib.patches.StepPatch``

Other Parameters

data : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

****kwargs**

`~matplotlib.patches.StepPatch`` properties

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``.axes.Axes.stairs``.

matplotlib.pyplot.stem

```
stem(*args: 'ArrayLike | str', linefmt: 'str | None' = None, markerfmt: 'str | None' =
None, basefmt: 'str | None' = None, bottom: 'float' = 0, label: 'str | None' = None,
orientation: "Literal['vertical', 'horizontal']" = 'vertical', data=None) ->
'StemContainer'
```

Create a stem plot.

A stem plot draws lines perpendicular to a baseline at each location `*locs*` from the baseline to `*heads*`, and places a marker there. For vertical stem plots (the default), the `*locs*` are `*x*` positions, and the `*heads*` are `*y*` values. For horizontal stem plots, the `*locs*` are `*y*` positions, and the `*heads*` are `*x*` values.

Call signature::

`stem([locs,] heads, linefmt=None, markerfmt=None, basefmt=None)`

The `*locs*`-positions are optional. `*linefmt*` may be provided as positional, but all other formats must be provided as keyword arguments.

Parameters

locs : array-like, default: (0, 1, ..., len(heads) - 1)

For vertical stem plots, the x-positions of the stems.

For horizontal stem plots, the y-positions of the stems.

heads : array-like

For vertical stem plots, the y-values of the stem heads.

For horizontal stem plots, the x-values of the stem heads.

linefmt : str, optional

A string defining the color and/or linestyle of the vertical lines:

=====

Character Line Style

```
=====
'''-''' solid line
'''--''' dashed line
'''-.'''' dash-dot line
'''.'''' dotted line
=====
```

Default: 'C0-', i.e. solid line with the first color of the color cycle.

Note: Markers specified through this parameter (e.g. 'x') will be silently ignored. Instead, markers should be specified using `*markerfmt*`.

`markerfmt` : str, optional

A string defining the color and/or shape of the markers at the stem heads. If the marker is not given, use the marker 'o', i.e. filled circles. If the color is not given, use the color from `*linefmt*`.

`basefmt` : str, default: 'C3-' ('C2-' in classic mode)

A format string defining the properties of the baseline.

`orientation` : {'vertical', 'horizontal'}, default: 'vertical'

The orientation of the stems.

`bottom` : float, default: 0

The y/x-position of the baseline (depending on `*orientation*`).

`label` : str, optional

The label to use for the stems in legends.

`data` : indexable object, optional

If given, all parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```.

Returns

``.StemContainer``

The container may be treated like a tuple

(`*markerline*`, `*stemlines*`, `*baseline*`)

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``.axes.Axes.stem``.

.. seealso::

The MATLAB function

``.stem`` <<https://www.mathworks.com/help/matlab/ref/stem.html>>`_

which inspired this method.

```
step(x: 'ArrayLike', y: 'ArrayLike', *args, where: "Literal['pre', 'post', 'mid']" = 'pre', data=None, **kwargs) -> 'list[Line2D]'
```

Make a step plot.

Call signatures::

```
step(x, y, [fmt], *, data=None, where='pre', **kwargs)
step(x, y, [fmt], x2, y2, [fmt2], ..., *, where='pre', **kwargs)
```

This is just a thin wrapper around `.plot` which changes some formatting options. Most of the concepts and parameters of plot can be used here as well.

.. note::

This method uses a standard plot with a step drawstyle: The `*x*` values are the reference positions and steps extend left/right/both directions depending on `*where*`.

For the common case where you know the values and edges of the steps, use `~.Axes.stairs` instead.

Parameters

`x` : array-like

1D sequence of x positions. It is assumed, but not checked, that it is uniformly increasing.

`y` : array-like

1D sequence of y levels.

`fmt` : str, optional

A format string, e.g. 'g' for a green line. See `.plot` for a more detailed description.

Note: While full format strings are accepted, it is recommended to only specify the color. Line styles are currently ignored (use the keyword argument `*linestyle*` instead). Markers are accepted and plotted on the given positions, however, this is a rarely needed feature for step plots.

`where` : {'pre', 'post', 'mid'}, default: 'pre'

Define where the steps should be placed:

- 'pre': The y value is continued constantly to the left from every `*x*` position, i.e. the interval `[(x[i-1], x[i])]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every `*x*` position, i.e. the interval `[(x[i], x[i+1])]` has the value `y[i]`.
- 'mid': Steps occur half-way between the `*x*` positions.

`data` : indexable object, optional

An object with labelled data. If given, provide the label names to plot in **x** and **y**.

****kwargs**

Additional parameters are the same as those for `.plot`.

Returns

list of `.Line2D`

Objects representing the plotted data.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.step`.

matplotlib.pyplot.streamplot

```
streamplot(x, y, u, v, density=1, linewidth=None, color=None, cmap=None, norm=None,
           arrowsize=1, arrowstyle='->', minlength=0.1, transform=None, zorder=None,
           start_points=None, maxlength=4.0, integration_direction='both',
           broken_streamlines=True, *, data=None)
```

Draw streamlines of a vector flow.

Parameters

x, y : 1D/2D arrays

Evenly spaced strictly increasing arrays to make a grid. If 2D, all rows of **x** must be equal and all columns of **y** must be equal; i.e., they must be as if generated by ```np.meshgrid(x_1d, y_1d)```.

u, v : 2D arrays

x and **y**-velocities. The number of rows and columns must match the length of **y** and **x**, respectively.

density : float or (float, float)

Controls the closeness of streamlines. When ```density = 1```, the domain is divided into a 30x30 grid. **density** linearly scales this grid.

Each cell in the grid can have, at most, one traversing streamline.

For different densities in each direction, use a tuple

(*density_x*, *density_y*).

linewidth : float or 2D array

The width of the streamlines. With a 2D array the line width can be varied across the grid. The array must have the same shape as **u** and **v**.

color : :mpltype:``color`` or 2D array

The streamline color. If given an array, its values are converted to colors using **cmap** and **norm**. The array must have the same shape as **u** and **v**.

cmap, *norm*

Data normalization and colormapping parameters for **color**; only used if **color** is an array of floats. See `~.Axes.imshow` for a detailed description.

arrowsize : float

Scaling factor for the arrow size.
 arrowstyle : str
 Arrow style specification.
 See `~matplotlib.patches.FancyArrowPatch``.
 minlength : float
 Minimum length of streamline in axes coordinates.
 start_points : (N, 2) array
 Coordinates of starting points for the streamlines in data coordinates
 (the same coordinates as the `*x*` and `*y*` arrays).
 zorder : float
 The zorder of the streamlines and arrows.
 Artists with lower zorder values are drawn first.
 maxlength : float
 Maximum length of streamline in axes coordinates.
 integration_direction : {'forward', 'backward', 'both'}, default: 'both'
 Integrate the streamline in forward, backward or both directions.
 data : indexable object, optional
 If given, the following parameters also accept a string `s``, which is
 interpreted as `data[s]`` if `s`` is a key in `data``:

`*x*, *y*, *u*, *v*, *start_points*`
 broken_streamlines : boolean, default: True
 If False, forces streamlines to continue until they
 leave the plot domain. If True, they may be terminated if they
 come too close to another streamline.

Returns

 StreamplotSet
 Container object with attributes

- `lines``: `.LineCollection`` of streamlines

- `arrows``: `.PatchCollection`` containing `.FancyArrowPatch``
 objects representing the arrows half-way along streamlines.

This container will probably change in the future to allow changes
 to the colormap, alpha, etc. for both lines and arrows, but these
 changes should be backward compatible.

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for `.axes.Axes.streamplot``.

matplotlib.pyplot.subplot

```
subplot(*args, **kwargs) -> 'Axes'
```

Add an Axes to the current figure or retrieve an existing Axes.

This is a wrapper of `.Figure.add_subplot`` which provides additional

behavior when working with the implicit API (see the notes section).

Call signatures::

```
subplot(nrows, ncols, index, **kwargs)
subplot(pos, **kwargs)
subplot(**kwargs)
subplot(ax)
```

Parameters

`*args` : int, (int, int, `*index*`), or `~.SubplotSpec``, default: (1, 1, 1)

The position of the subplot described by one of

- Three integers (`*nrows*`, `*ncols*`, `*index*`). The subplot will take the `*index*` position on a grid with `*nrows*` rows and `*ncols*` columns. `*index*` starts at 1 in the upper left corner and increases to the right. `*index*` can also be a two-tuple specifying the (`*first*`, `*last*`) indices (1-based, and including `*last*`) of the subplot, e.g., ```fig.add_subplot(3, 1, (1, 2))``` makes a subplot that spans the upper 2/3 of the figure.
- A 3-digit integer. The digits are interpreted as if given separately as three single-digit integers, i.e. ```fig.add_subplot(235)``` is the same as ```fig.add_subplot(2, 3, 5)```. Note that this can only be used if there are no more than 9 subplots.
- A `~.SubplotSpec``.

`projection` : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional

The projection type of the subplot (`~.axes.Axes``). `*str*` is the name of a custom projection, see `~matplotlib.projections``. The default None results in a 'rectilinear' projection.

`polar` : bool, default: False

If True, equivalent to `projection='polar'`.

`sharex`, `sharey` : `~matplotlib.axes.Axes``, optional

Share the x or y `~matplotlib.axis`` with `sharex` and/or `sharey`. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

`label` : str

A label for the returned Axes.

Returns

`~.axes.Axes``

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as `~.projections.polar.PolarAxes`` for polar projections.

Other Parameters

`**kwargs`

This method also takes the keyword arguments for the returned Axes

base class; except for the **figure** argument. The keyword arguments for the rectilinear base class `~.axes.Axes`` can be found in the following table but there might also be other keyword arguments if another projection is used.

Properties:

adjustable: {'box', 'datalim'}

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

alpha: float or None

anchor: (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...}

animated: bool

aspect: {'auto', 'equal'} or float

autoscale_on: bool

autoscalex_on: unknown

autoscaley_on: unknown

axes_locator: Callable[[Axes, Renderer], Bbox]

axisbelow: bool or 'line'

box_aspect: float or None

clip_box: `~matplotlib.transforms.BboxBase`` or None

clip_on: bool

clip_path: Patch or (Path, Transform) or None

facecolor or fc: `:mpltype:`color``

figure: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

forward_navigation_events: bool or "auto"

frame_on: bool

gid: str

in_layout: bool

label: object

mouseover: bool

navigate: bool

navigate_mode: unknown

path_effects: list of `~.AbstractPathEffect``

picker: None or bool or float or callable

position: [left, bottom, width, height] or `~matplotlib.transforms.Bbox``

prop_cycle: `~cycler.Cycler``

rasterization_zorder: float or None

rasterized: bool

sketch_params: (scale: float, length: float, randomness: float)

snap: bool or None

subplotspec: unknown

title: str

transform: `~matplotlib.transforms.Transform``

url: str

visible: bool

xbound: (lower: float, upper: float)

xlabel: str

xlim: (left: float, right: float)

xmargin: float greater than -0.5

xscale: unknown

xticklabels: unknown

xticks: unknown

ybound: (lower: float, upper: float)

ylabel: str

ylim: (bottom: float, top: float)

ymargin: float greater than -0.5
yscale: unknown
yticklabels: unknown
yticks: unknown
zorder: float

Notes

.. versionchanged:: 3.8

In versions prior to 3.8, any preexisting Axes that overlap with the new Axes beyond sharing a boundary was deleted. Deletion does not happen in more recent versions anymore. Use ``Axes.remove`` explicitly if needed.

If you do not want this behavior, use the ``Figure.add_subplot`` method or the ``pyplot.axes`` function instead.

If no `*kwargs*` are passed and there exists an Axes in the location specified by `*args*` then that Axes will be returned rather than a new Axes being created.

If `*kwargs*` are passed and there exists an Axes in the location specified by `*args*`, the projection type is the same, and the `*kwargs*` match with the existing Axes, then the existing Axes is returned. Otherwise a new Axes is created with the specified parameters. We save a reference to the `*kwargs*` which we use for this comparison. If any of the values in `*kwargs*` are mutable we will not detect the case where they are mutated. In these cases we suggest using ``Figure.add_subplot`` and the explicit Axes API rather than the implicit pyplot API.

See Also

`.Figure.add_subplot`
`.pyplot.subplots`
`.pyplot.axes`
`.Figure.subplots`

Examples

::

```
plt.subplot(221)
```

```
# equivalent but more general  
ax1 = plt.subplot(2, 2, 1)
```

```
# add a subplot with no frame  
ax2 = plt.subplot(222, frameon=False)
```

```
# add a polar subplot  
plt.subplot(223, projection='polar')
```

```
# add a red subplot that shares the x-axis with ax1  
plt.subplot(224, sharex=ax1, facecolor='red')
```

```
# delete ax2 from the figure
plt.delaxes(ax2)

# add ax2 to the figure again
plt.subplot(ax2)

# make the first Axes "current" again
plt.subplot(221)
```

matplotlib.pyplot.subplot2grid

```
subplot2grid(shape: 'tuple[int, int]', loc: 'tuple[int, int]', rowspan: 'int' = 1,
             colspan: 'int' = 1, fig: 'Figure | None' = None, **kwargs) -> 'matplotlib.axes.Axes'
```

Create a subplot at a specific location inside a regular grid.

Parameters

shape : (int, int)

Number of rows and of columns of the grid in which to place axis.

loc : (int, int)

Row number and column number of the axis location within the grid.

rowspan : int, default: 1

Number of rows for the axis to span downwards.

colspan : int, default: 1

Number of columns for the axis to span to the right.

fig : `Figure`, optional

Figure to place the subplot in. Defaults to the current figure.

****kwargs**

Additional keyword arguments are handed to `~.Figure.add_subplot`.

Returns

`~.axes.Axes`

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as `~.projections.polar.PolarAxes` for polar projections.

Notes

The following call ::

```
ax = subplot2grid((nrows, ncols), (row, col), rowspan, colspan)
```

is identical to ::

```
fig = gcf()
gs = fig.add_gridspec(nrows, ncols)
ax = fig.add_subplot(gs[row:row+rowspan, col:col+colspan])
```

matplotlib.pyplot.subplot_mosaic

```
subplot_mosaic(mosaic: 'str | list[HashableList[_T]] | list[HashableList[Hashable]]',
*, sharex: 'bool' = False, sharey: 'bool' = False, width_ratios: 'ArrayLike | None' =
None, height_ratios: 'ArrayLike | None' = None, empty_sentinel: 'Any' = '.',
subplot_kw: 'dict[str, Any] | None' = None, gridspec_kw: 'dict[str, Any] | None' =
None, per_subplot_kw: 'dict[str | tuple[str, ...], dict[str, Any]] | dict[_T |
tuple[_T, ...], dict[str, Any]] | dict[Hashable | tuple[Hashable, ...], dict[str,
Any]] | None' = None, **fig_kw: 'Any') -> 'tuple[Figure, dict[str,
matplotlib.axes.Axes]] | tuple[Figure, dict[_T, matplotlib.axes.Axes]] | tuple[Figure,
dict[Hashable, matplotlib.axes.Axes]]'
```

Build a layout of Axes based on ASCII art or nested lists.

This is a helper function to build complex GridSpec layouts visually.

See :ref:`mosaic`
for an example and full API documentation

Parameters

mosaic : list of list of {hashable or nested} or str

A visual layout of how you want your Axes to be arranged
labeled as strings. For example ::

```
x = [['A panel', 'A panel', 'edge'],
['C panel', '.', 'edge']]
```

produces 4 Axes:

- 'A panel' which is 1 row high and spans the first two columns
- 'edge' which is 2 rows high and is on the right edge
- 'C panel' which in 1 row and 1 column wide in the bottom left
- a blank space 1 row and 1 column wide in the bottom center

Any of the entries in the layout can be a list of lists
of the same form to create nested layouts.

If input is a str, then it must be of the form ::

```
""
AAE
C.E
""
```

where each character is a column and each line is a row.
This only allows only single character Axes labels and does
not allow nesting but is very terse.

sharex, sharey : bool, default: False

If True, the x-axis (*sharex*) or y-axis (*sharey*) will be shared
among all subplots. In that case, tick label visibility and axis units
behave as for `subplots`. If False, each subplot's x- or y-axis will
be independent.

width_ratios : array-like of length *ncols*, optional

Defines the relative widths of the columns. Each column gets a
relative width of ``width_ratios[i] / sum(width_ratios)``.

If not given, all columns will have the same width. Convenience for ``gridspec_kw={width_ratios: [...]}``.

`height_ratios` : array-like of length `*nrows*`, optional
Defines the relative heights of the rows. Each row gets a relative height of ``height_ratios[i] / sum(height_ratios)``.
If not given, all rows will have the same height. Convenience for ``gridspec_kw={'height_ratios': [...]}``.

`empty_sentinel` : object, optional
Entry in the layout to mean "leave this space empty". Defaults to ``"."``. Note, if `*layout*` is a string, it is processed via ``inspect.cleandoc`` to remove leading white space, which may interfere with using white-space as the empty sentinel.

`subplot_kw` : dict, optional
Dictionary with keywords passed to the ``Figure.add_subplot`` call used to create each subplot. These values may be overridden by values in `*per_subplot_kw*`.

`per_subplot_kw` : dict, optional
A dictionary mapping the Axes identifiers or tuples of identifiers to a dictionary of keyword arguments to be passed to the ``Figure.add_subplot`` call used to create each subplot. The values in these dictionaries have precedence over the values in `*subplot_kw*`.

If `*mosaic*` is a string, and thus all keys are single characters, it is possible to use a single string instead of a tuple as keys; i.e. ``"AB"`` is equivalent to ``("A", "B")``.

.. versionadded:: 3.7

`gridspec_kw` : dict, optional
Dictionary with keywords passed to the ``GridSpec`` constructor used to create the grid the subplots are placed on.

****fig_kw**
All additional keyword arguments are passed to the ``pyplot.figure`` call.

Returns

`fig` : ``Figure``
The new figure

`dict[label, Axes]`
A dictionary mapping the labels to the Axes objects. The order of the Axes is left-to-right and top-to-bottom of their position in the total layout.

matplotlib.pyplot.subplot_tool

```
subplot_tool(targetfig: 'Figure | None' = None) -> 'SubplotTool | None'
```

Launch a subplot tool window for a figure.

Returns

`matplotlib.widgets.SubplotTool`

matplotlib.pyplot.subplots

```
subplots(nrows: 'int' = 1, ncols: 'int' = 1, *, sharex: "bool | Literal['none', 'all', 'row', 'col']" = False, sharey: "bool | Literal['none', 'all', 'row', 'col']" = False, squeeze: 'bool' = True, width_ratios: 'Sequence[float] | None' = None, height_ratios: 'Sequence[float] | None' = None, subplot_kw: 'dict[str, Any] | None' = None, gridspec_kw: 'dict[str, Any] | None' = None, **fig_kw) -> 'tuple[Figure, Any]'
```

Create a figure and a set of subplots.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

Parameters

nrows, ncols : int, default: 1

Number of rows/columns of the subplot grid.

sharex, sharey : bool or {'none', 'all', 'row', 'col'}, default: False

Controls sharing of properties among x (*sharex*) or y (*sharey*) axes:

- True or 'all': x- or y-axis will be shared among all subplots.
- False or 'none': each subplot x- or y-axis will be independent.
- 'row': each subplot row will share an x- or y-axis.
- 'col': each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created. To later turn other subplots' ticklabels on, use `~matplotlib.axes.Axes.tick_params`.

When subplots have a shared axis that has units, calling `Axis.set_units` will update each axis with the new units.

Note that it is not possible to unshare axes.

squeeze : bool, default: True

- If True, extra dimensions are squeezed out from the returned array of `~matplotlib.axes.Axes`:

- if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
- for Nx1 or 1xM subplots, the returned object is a 1D numpy object array of Axes objects.
- for NxM, subplots with N>1 and M>1 are returned as a 2D array.

- If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up

being 1x1.

`width_ratios` : array-like of length `*ncols*`, optional
Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`.
If not given, all columns will have the same width. Equivalent to `gridspec_kw={'width_ratios': [...]}`.

`height_ratios` : array-like of length `*nrows*`, optional
Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`.
If not given, all rows will have the same height. Convenience for `gridspec_kw={'height_ratios': [...]}`.

`subplot_kw` : dict, optional
Dict with keywords passed to the `~matplotlib.figure.Figure.add_subplot`` call used to create each subplot.

`gridspec_kw` : dict, optional
Dict with keywords passed to the `~matplotlib.gridspec.GridSpec`` constructor used to create the grid the subplots are placed on.

****fig_kw**
All additional keyword arguments are passed to the `~pyplot.figure`` call.

Returns

`fig` : `~.Figure``

`ax` : `~matplotlib.axes.Axes`` or array of Axes
`*ax*` can be either a single `~.axes.Axes`` object, or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the `squeeze` keyword, see above.

Typical idioms for handling the return value are::

```
# using the variable ax for single a Axes
fig, ax = plt.subplots()
```

```
# using the variable axs for multiple Axes
fig, axs = plt.subplots(2, 2)
```

```
# using tuple unpacking for multiple Axes
fig, (ax1, ax2) = plt.subplots(1, 2)
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
```

The names `ax`` and pluralized `axs`` are preferred over `axes`` because for the latter it's not clear if it refers to a single `~.axes.Axes`` instance or a collection of these.

See Also

`~pyplot.figure`

```
.pyplot.subplot  
.pyplot.axes  
.Figure.subplots  
.Figure.add_subplot
```

Examples

```
-----  
::
```

```
# First create some toy data:  
x = np.linspace(0, 2*np.pi, 400)  
y = np.sin(x**2)
```

```
# Create just a figure and only one subplot  
fig, ax = plt.subplots()  
ax.plot(x, y)  
ax.set_title('Simple plot')
```

```
# Create two subplots and unpack the output array immediately  
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)  
ax1.plot(x, y)  
ax1.set_title('Sharing Y axis')  
ax2.scatter(x, y)
```

```
# Create four polar Axes and access them through the returned array  
fig, axs = plt.subplots(2, 2, subplot_kw=dict(projection="polar"))  
axs[0, 0].plot(x, y)  
axs[1, 1].scatter(x, y)
```

```
# Share a X axis with each column of subplots  
plt.subplots(2, 2, sharex='col')
```

```
# Share a Y axis with each row of subplots  
plt.subplots(2, 2, sharey='row')
```

```
# Share both X and Y axes with all subplots  
plt.subplots(2, 2, sharex='all', sharey='all')
```

```
# Note that this is the same as  
plt.subplots(2, 2, sharex=True, sharey=True)
```

```
# Create figure number 10 with a single subplot  
# and clears it if it already exists.  
fig, ax = plt.subplots(num=10, clear=True)
```

matplotlib.pyplot.subplots_adjust

```
subplots_adjust(left: 'float | None' = None, bottom: 'float | None' = None, right:  
'float | None' = None, top: 'float | None' = None, wspace: 'float | None' = None,  
hspace: 'float | None' = None) -> 'None'
```

Adjust the subplot layout parameters.

Unset parameters are left unmodified; initial values are given by
:rc:`figure.subplot.[name]`.

.. plot:: _embedded_plots/figure_subplots_adjust.py

Parameters

left : float, optional

The position of the left edge of the subplots,
as a fraction of the figure width.

right : float, optional

The position of the right edge of the subplots,
as a fraction of the figure width.

bottom : float, optional

The position of the bottom edge of the subplots,
as a fraction of the figure height.

top : float, optional

The position of the top edge of the subplots,
as a fraction of the figure height.

wspace : float, optional

The width of the padding between subplots,
as a fraction of the average Axes width.

hspace : float, optional

The height of the padding between subplots,
as a fraction of the average Axes height.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `Figure.subplots_adjust``.

matplotlib.pyplot.summer

```
summer() -> 'None'
```

Set the colormap to 'summer'.

This changes the default colormap as well as the colormap of the current image if there is one. See `help(colormaps)`` for more information.

matplotlib.pyplot.suptitle

```
suptitle(t: 'str', **kwargs) -> 'Text'
```

Add a centered super title to the figure.

Parameters

t : str

The super title text.

x : float, default: 0.5

The x location of the text in figure coordinates.

y : float, default: 0.98

The y location of the text in figure coordinates.

horizontalalignment, ha : {'center', 'left', 'right'}, default: center
The horizontal alignment of the text relative to (*x*, *y*).

verticalalignment, va : {'top', 'center', 'bottom', 'baseline'}, default: top
The vertical alignment of the text relative to (*x*, *y*).

fontsize, size : default: :rc:`figure.titlesize`
The font size of the text. See ``Text.set_size`` for possible values.

fontweight, weight : default: :rc:`figure.titleweight`
The font weight of the text. See ``Text.set_weight`` for possible values.

Returns

text

The ``Text`` instance of the super title.

Other Parameters

fontproperties : None or dict, optional

A dict of font properties. If `*fontproperties*` is given the default values for font size and weight are taken from the ``FontProperties`` defaults. `:rc:`figure.titlesize`` and `:rc:`figure.titleweight`` are ignored in this case.

**kwargs

Additional kwargs are ``matplotlib.text.Text`` properties.

Notes

.. note::

This is the `:ref:`pyplot`` wrapper `<pyplot_interface>` for ``Figure.suptitle``.

matplotlib.pyplot.switch_backend

```
switch_backend(newbackend: 'str') -> 'None'
```

Set the pyplot backend.

Switching to an interactive backend is possible only if no event loop for another interactive backend has started. Switching to and from non-interactive backends is always possible.

If the new backend is different than the current backend then all open Figures will be closed via ``plt.close('all')``.

Parameters

newbackend : str

The case-insensitive name of the backend to use.

matplotlib.pyplot.table

```
table(cellText=None, cellColours=None, cellLoc='right', colWidths=None,
rowLabels=None, rowColours=None, rowLoc='left', colLabels=None, colColours=None,
colLoc='center', loc='bottom', bbox=None, edges='closed', **kwargs)
```

Add a table to an `~.axes.Axes``.

At least one of `*cellText*` or `*cellColours*` must be specified. These parameters must be 2D lists, in which the outer lists define the rows and the inner list define the column values per row. Each row must have the same number of elements.

The table can optionally have row and column headers, which are configured using `*rowLabels*`, `*rowColours*`, `*rowLoc*` and `*colLabels*`, `*colColours*`, `*colLoc*` respectively.

For finer grained control over tables, use the ``.Table`` class and add it to the Axes with ``.Axes.add_table``.

Parameters

`cellText` : 2D list of str or pandas.DataFrame, optional
The texts to place into the table cells.

Note: Line breaks in the strings are currently not accounted for and will result in the text exceeding the cell boundaries.

`cellColours` : 2D list of :mpltype:`color`, optional
The background colors of the cells.

`cellLoc` : {'right', 'center', 'left'}
The alignment of the text within the cells.

`colWidths` : list of float, optional
The column widths in units of the axes. If not given, all columns will have a width of `*1 / ncols*`.

`rowLabels` : list of str, optional
The text of the row header cells.

`rowColours` : list of :mpltype:`color`, optional
The colors of the row header cells.

`rowLoc` : {'left', 'center', 'right'}
The text alignment of the row header cells.

`colLabels` : list of str, optional
The text of the column header cells.

`colColours` : list of :mpltype:`color`, optional
The colors of the column header cells.

`colLoc` : {'center', 'left', 'right'}
The text alignment of the column header cells.

`loc` : str, default: 'bottom'
The position of the cell with respect to `*ax*`. This must be one of

the `~.Table.codes``.

`bbox`` : `~.Bbox`` or `[xmin, ymin, width, height]`, optional
A bounding box to draw the table into. If this is not `*None*`, this overrides `*loc*`.

`edges`` : `{'closed', 'open', 'horizontal', 'vertical'}` or substring of `'BRTL'`
The cell edges to be drawn with a line. See also
`~.Cell.visible_edges``.

Returns

`~matplotlib.table.Table``
The created table.

Other Parameters

`**kwargs`
`~.Table`` properties.

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
`alpha`: float or None
`animated`: bool
`clip_box`` : `~matplotlib.transforms.BboxBase`` or None
`clip_on`: bool
`clip_path`: Patch or (Path, Transform) or None
`figure`` : `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``
`fontsize`: float
`gid`: str
`in_layout`: bool
`label`: object
`mouseover`: bool
`path_effects`: list of `~.AbstractPathEffect``
`picker`: None or bool or float or callable
`rasterized`: bool
`sketch_params`: (scale: float, length: float, randomness: float)
`snap`: bool or None
`transform`` : `~matplotlib.transforms.Transform``
`url`: str
`visible`: bool
`zorder`: float

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.table``.

matplotlib.pyplot.text

```
text(x: 'float', y: 'float', s: 'str', fontdict: 'dict[str, Any] | None' = None,
**kwargs) -> 'Text'
```

Add text to the Axes.

Add the text `*s*` to the Axes at location `*x*`, `*y*` in data coordinates, with a default `horizontalalignment` on the `left` and `verticalalignment` at the `baseline`. See [:doc:`gallery/text_labels_and_annotations/text_alignment`](https://matplotlib.org/gallery/text_labels_and_annotations/text_alignment.html).

Parameters

`x, y` : float

The position to place the text. By default, this is in data coordinates. The coordinate system can be changed using the `*transform*` parameter.

`s` : str

The text.

`fontdict` : dict, default: None

.. admonition:: Discouraged

The use of `*fontdict*` is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `text(..., **fontdict)`.

A dictionary to override the default text properties. If `fontdict` is None, the defaults are determined by `.rcParams`.

Returns

`Text`

The created `Text` instance.

Other Parameters

`**kwargs` : `~matplotlib.text.Text` properties.

Other miscellaneous text parameters.

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: float or None

`animated`: bool

`antialiased`: bool

`backgroundcolor`: `mpltype:color`

`bbox`: dict with properties for `.patches.FancyBboxPatch`

`clip_box`: unknown

`clip_on`: unknown

`clip_path`: unknown

`color` or `c`: `mpltype:color`

`figure`: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`

`fontfamily` or `family` or `fontname`: {`FONTNAME`, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}

`fontproperties` or `font` or `font_properties`: `.font_manager.FontProperties` or `str` or `pathlib.Path`

fontsize or size: float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}
 fontstretch or stretch: {a numeric value in range 0-1000, 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'}
 fontstyle or style: {'normal', 'italic', 'oblique'}
 fontvariant or variant: {'normal', 'small-caps'}
 fontweight or weight: {a numeric value in range 0-1000, 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}
 gid: str
 horizontalalignment or ha: {'left', 'center', 'right'}
 in_layout: bool
 label: object
 linespacing: float (multiple of font size)
 math_fontfamily: str
 mouseover: bool
 multialignment or ma: {'left', 'right', 'center'}
 parse_math: bool
 path_effects: list of `AbstractPathEffect`
 picker: None or bool or float or callable
 position: (float, float)
 rasterized: bool
 rotation: float or {'vertical', 'horizontal'}
 rotation_mode: {None, 'default', 'anchor'}
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 text: object
 transform: `~matplotlib.transforms.Transform`
 transform_rotates_text: bool
 url: str
 usetex: bool, default: `:rc:'text.usetex'`
 verticalalignment or va: {'baseline', 'bottom', 'center', 'center_baseline', 'top'}
 visible: bool
 wrap: bool
 x: float
 y: float
 zorder: float

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for `~.axes.Axes.text``.

Examples

Individual keyword arguments can be used to override any given parameter::

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords ((0, 0) is lower-left and (1, 1) is upper-right). The example below places text in the center of the Axes::

```
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
... verticalalignment='center', transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword `*bbox*`. `*bbox*` is a dictionary of `~matplotlib.patches.Rectangle`` properties. For example::

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

matplotlib.pyplot.thetagrids

```
thetagrids(angles: 'ArrayLike | None' = None, labels: 'Sequence[str | Text] | None' =
None, fmt: 'str | None' = None, **kwargs) -> 'tuple[list[Line2D], list[Text]]'
```

Get or set the theta gridlines on the current polar plot.

Call signatures::

```
lines, labels = thetagrids()
lines, labels = thetagrids(angles, labels=None, fmt=None, **kwargs)
```

When called with no arguments, `~.thetagrids`` simply returns the tuple `(*lines*, *labels*)`. When called with arguments, the labels will appear at the specified angles.

Parameters

`angles` : tuple with floats, degrees
The angles of the theta gridlines.

`labels` : tuple with strings or None
The labels to use at each radial gridline. The `~.projections.polar.ThetaFormatter`` will be used if None.

`fmt` : str or None
Format string used in `~matplotlib.ticker.FormatStrFormatter``. For example `'%f'`. Note that the angle in radians will be used.

Returns

`lines` : list of `~.lines.Line2D``
The theta gridlines.

`labels` : list of `~.text.Text``
The tick labels.

Other Parameters

`**kwargs`
`*kwargs*` are optional `~.Text`` properties for the labels.

See Also

`~.pyplot.rgrids`

```
.projections.polar.PolarAxes.set_thetagrids
.Axis.get_gridlines
.Axis.get_ticklabels
```

Examples

```
-----
::
```

```
# set the locations of the angular gridlines
lines, labels = thetagrids(range(45, 360, 90))
```

```
# set the locations and labels of the angular gridlines
lines, labels = thetagrids(range(45, 360, 90), ('NE', 'NW', 'SW', 'SE'))
```

matplotlib.pyplot.tick_params

```
tick_params(axis: "Literal['both', 'x', 'y']" = 'both', **kwargs) -> 'None'
```

Change the appearance of ticks, tick labels, and gridlines.

Tick properties that are not explicitly set using the keyword arguments remain unchanged unless `*reset*` is True. For the current style settings, see `.Axis.get_tick_params``.

Parameters

```
-----
```

`axis` : {'x', 'y', 'both'}, default: 'both'

The axis to which the parameters are applied.

`which` : {'major', 'minor', 'both'}, default: 'major'

The group of ticks to which the parameters are applied.

`reset` : bool, default: False

Whether to reset the ticks to defaults before updating them.

Other Parameters

```
-----
```

`direction` : {'in', 'out', 'inout'}

Puts ticks inside the Axes, outside the Axes, or both.

`length` : float

Tick length in points.

`width` : float

Tick width in points.

`color` : :mpltype:`color`

Tick color.

`pad` : float

Distance in points between tick and label.

`labelsize` : float or str

Tick label font size in points or as a string (e.g., 'large').

`labelcolor` : :mpltype:`color`

Tick label color.

`labelfontfamily` : str

Tick label font.

`colors` : :mpltype:`color`

Tick color and label color.

`zorder` : float

Tick and label zorder.
 bottom, top, left, right : bool
 Whether to draw the respective ticks.
 labelbottom, labeltop, labelleft, labelright : bool
 Whether to draw the respective tick labels.
 labelrotation : float
 Tick label rotation
 grid_color : :mpltype:`color`
 Gridline color.
 grid_alpha : float
 Transparency of gridlines: 0 (transparent) to 1 (opaque).
 grid_linewidth : float
 Width of gridlines in points.
 grid_linestyle : str
 Any valid ``Line2D`` line style spec.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``axes.Axes.tick_params``.

Examples

::

```
ax.tick_params(direction='out', length=6, width=2, colors='r',
               grid_color='r', grid_alpha=0.5)
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red. Gridlines will be red and translucent.

matplotlib.pyplot.ticklabel_format

```
ticklabel_format(*, axis: "Literal['both', 'x', 'y']" = 'both', style: "Literal['', 'sci', 'scientific', 'plain'] | None" = None, scilimits: 'tuple[int, int] | None' = None, useOffset: 'bool | float | None' = None, useLocale: 'bool | None' = None, useMathText: 'bool | None' = None) -> 'None'
```

Configure the ``ScalarFormatter`` used by default for linear Axes.

If a parameter is not set, the corresponding property of the formatter is left unchanged.

Parameters

axis : {'x', 'y', 'both'}, default: 'both'

The axis to configure. Only major ticks are affected.

style : {'sci', 'scientific', 'plain'}

Whether to use scientific notation.

The formatter default is to use scientific notation.

'sci' is equivalent to 'scientific'.

scilimits : pair of ints (m, n)

Scientific notation is used only for numbers outside the range 10^m to 10^n (and only if the formatter is configured to use scientific notation at all). Use (0, 0) to include all numbers. Use (m, m) where $m \neq 0$ to fix the order of magnitude to 10^m .
The formatter default is `:rc:`axes.formatter.limits``.

useOffset : bool or float

If True, the offset is calculated as needed.

If False, no offset is used.

If a numeric value, it sets the offset.

The formatter default is `:rc:`axes.formatter.useoffset``.

useLocale : bool

Whether to format the number using the current locale or using the C (English) locale. This affects e.g. the decimal separator. The formatter default is `:rc:`axes.formatter.use_locale``.

useMathText : bool

Render the offset and scientific notation in mathtext.

The formatter default is `:rc:`axes.formatter.use_mathtext``.

Raises

AttributeError

If the current formatter is not a ``ScalarFormatter``.

Notes

.. note::

This is the `:ref:`pyplot` wrapper <pyplot_interface> for `axes.Axes.ticklabel_format`.`

matplotlib.pyplot.tight_layout

```
tight_layout(*, pad: 'float' = 1.08, h_pad: 'float | None' = None, w_pad: 'float | None' = None, rect: 'tuple[float, float, float, float] | None' = None) -> 'None'
```

Adjust the padding between and around subplots.

To exclude an artist on the Axes from the bounding box calculation that determines the subplot parameters (i.e. legend, or annotation), set ``a.set_in_layout(False)`` for that artist.

Parameters

pad : float, default: 1.08

Padding between the figure edge and the edges of subplots, as a fraction of the font size.

h_pad, w_pad : float, default: *pad*

Padding (height/width) between edges of adjacent subplots, as a fraction of the font size.

rect : tuple (left, bottom, right, top), default: (0, 0, 1, 1)
A rectangle in normalized figure coordinates into which the whole subplots area (including labels) will fit.

See Also

.Figure.set_layout_engine
.pyplot.tight_layout

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.Figure.tight_layout``.

matplotlib.pyplot.title

```
title(label: 'str', fontdict: 'dict[str, Any] | None' = None, loc: "Literal['left', 'center', 'right'] | None" = None, pad: 'float | None' = None, *, y: 'float | None' = None, **kwargs) -> 'Text'
```

Set a title for the Axes.

Set one of the three available Axes titles. The available titles are positioned above the Axes in the center, flush with the left edge, and flush with the right edge.

Parameters

label : str
Text to use for the title

fontdict : dict

.. admonition:: Discouraged

The use of `*fontdict*` is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking ```set_title(..., **fontdict)```.

A dictionary controlling the appearance of the title text, the default `*fontdict*` is::

```
{'fontsize': rcParams['axes.titlesize'],  
'fontweight': rcParams['axes.titleweight'],  
'color': rcParams['axes.titlecolor'],  
'verticalalignment': 'baseline',  
'horizontalalignment': loc}
```

loc : {'center', 'left', 'right'}, default: :rc:`axes.titlelocation`
Which title to set.

y : float, default: :rc:`axes.titley`
Vertical Axes location for the title (1.0 is the top). If

None (the default) and `:rc:`axes.title`` is also None, y is determined automatically to avoid decorators on the Axes.

`pad` : float, default: `:rc:`axes.titlepad``
The offset of the title from the top of the Axes, in points.

Returns

``Text``
The matplotlib text instance representing the title

Other Parameters

`**kwargs` : ``~matplotlib.text.Text`` properties
Other keyword arguments are text properties, see ``Text`` for a list of valid text properties.

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for ``axes.Axes.set_title``.

matplotlib.pyplot.tricontour

```
tricontour(*args, **kwargs)
```

Draw contour lines on an unstructured triangular grid.

Call signatures::

```
tricontour(triangulation, z, [levels], ...)
tricontour(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a ``Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. See ``Triangulation`` for an explanation of these parameters. If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly.

It is possible to pass `*triangles*` positionally, i.e. ``tricontour(x, y, triangles, z, ...)``. However, this is discouraged. For more clarity, pass `*triangles*` via keyword argument.

Parameters

`triangulation` : ``Triangulation``, optional
An already created triangular grid.

`x`, `y`, `triangles`, `mask`
Parameters defining the triangular grid. See ``Triangulation``.
This is mutually exclusive with specifying `*triangulation*`.

`z` : array-like

The height values over which the contour is drawn. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

.. note::

All values in `*z*` must be finite. Hence, nan and inf values must either be removed or `~.Triangulation.set_mask`` be used.

`levels` : int or array-like, optional

Determines the number and positions of the contour lines / regions.

If an int `*n*`, use `~matplotlib.ticker.MaxNLocator``, which tries to automatically choose no more than `*n+1*` "nice" contour levels between between minimum and maximum numeric values of `*Z*`.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

`~matplotlib.tri.TriContourSet``

Other Parameters

`colors` : `:mpltype:`color`` or list of `:mpltype:`color``, optional

The colors of the levels, i.e., the contour lines.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. ```red``` instead of ```[red]``` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value `*None*`), the colormap specified by `*cmap*` will be used.

`alpha` : float, default: 1

The alpha blending value, between 0 (transparent) and 1 (opaque).

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*colors*` is set.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `~.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).

- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()`. In that case, a suitable `.Normalize` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin, vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/*vmax*` when a `*norm*` instance is given (but using a `str`*norm*` name together with `*vmin*/*vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`origin` : {`*None*`, 'upper', 'lower', 'image'}, default: None

Determines the orientation and exact position of `*z*` by specifying the position of `z[0, 0]`. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: `z[0, 0]` is at $X=0$, $Y=0$ in the lower left corner.
- 'lower': `z[0, 0]` is at $X=0.5$, $Y=0.5$ in the lower left corner.
- 'upper': `z[0, 0]` is at $X=N+0.5$, $Y=0.5$ in the upper left corner.
- 'image': Use the value from `rc:'image.origin'`.

`extent` : (x0, x1, y0, y1), optional

If `*origin*` is not `*None*`, then `*extent*` is interpreted as in `.imshow`: it gives the outer pixel boundaries. In this case, the position of `z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then (`*x0*`, `*y0*`) is the position of `z[0, 0]`, and (`*x1*`, `*y1*`) is the position of `z[-1, -1]`.

This argument is ignored if `*X*` and `*Y*` are specified in the call to `contour`.

`locator` : `ticker.Locator` subclass, optional

The locator is used to determine the contour levels if they are not given explicitly via `*levels*`.

Defaults to `~.ticker.MaxNLocator`.

`extend` : {'neither', 'both', 'min', 'max'}, default: 'neither'

Determines the `tricontour`-coloring of values that are outside the `*levels*` range.

If 'neither', values outside the `*levels*` range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the `*levels*` range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the `.Colormap`. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values

explicitly using ``Colormap.set_under`` and ``Colormap.set_over``.

.. note::

An existing ``TriContourSet`` does not get notified if properties of its colormap are changed. Therefore, an explicit call to ``ContourSet.changed()`` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the ``TriContourSet`` because it internally calls ``ContourSet.changed()``.

xunits, yunits : registered units, optional
Override axis units by specifying an instance of a
:class:`matplotlib.units.ConversionInterface`.

antialiased : bool, optional
Enable antialiasing, overriding the defaults. For
filled contours, the default is `*True*`. For line contours,
it is taken from `:rc:`lines.antialiased``.

linewidths : float or array-like, default: `:rc:`contour.linewidth``
The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with
the linewidths in the order specified.

If None, this falls back to `:rc:`lines.linewidth``.

linestyles : `{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}`, optional
If `*linestyles*` is `*None*`, the default is 'solid' unless the lines are
monochrome. In that case, negative contours will take their linestyle
from `:rc:`contour.negative_linestyle`` setting.

`*linestyles*` can also be an iterable of the above strings specifying a
set of linestyles to be used. If this iterable is shorter than the
number of contour levels it will be repeated as necessary.

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for ``axes.Axes.tricontour``.

matplotlib.pyplot.tricontourf

```
tricontourf(*args, **kwargs)
```

Draw contour regions on an unstructured triangular grid.

Call signatures::

```
tricontourf(triangulation, z, [levels], ...)
```

```
tricontourf(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a `~.Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. See `~.Triangulation`` for an explanation of these parameters. If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly.

It is possible to pass `*triangles*` positionally, i.e. ```tricontourf(x, y, triangles, z, ...)``. However, this is discouraged. For more clarity, pass *triangles* via keyword argument.`

Parameters

`triangulation` : `~.Triangulation``, optional
An already created triangular grid.

`x`, `y`, `triangles`, `mask`
Parameters defining the triangular grid. See `~.Triangulation``.
This is mutually exclusive with specifying `*triangulation*`.

`z` : array-like
The height values over which the contour is drawn. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

.. note::
All values in `*z*` must be finite. Hence, nan and inf values must either be removed or `~.Triangulation.set_mask`` be used.

`levels` : int or array-like, optional
Determines the number and positions of the contour lines / regions.

If an int `*n*`, use `~matplotlib.ticker.MaxNLocator``, which tries to automatically choose no more than `*n+1*` "nice" contour levels between between minimum and maximum numeric values of `*Z*`.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

`~matplotlib.tri.TriContourSet``

Other Parameters

`colors` : `:mpltype:`color`` or list of `:mpltype:`color``, optional
The colors of the levels, i.e., the contour regions.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. ```red``` instead of ```[red]``` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value `*None*`), the colormap specified by `*cmap*` will be used.

`alpha` : float, default: 1

The alpha blending value, between 0 (transparent) and 1 (opaque).

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*colors*` is set.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin`, `vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`origin` : `{*None*, 'upper', 'lower', 'image'}`, default: None

Determines the orientation and exact position of `*z*` by specifying the position of ```z[0, 0]```. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: ```z[0, 0]``` is at X=0, Y=0 in the lower left corner.
- `'lower'`: ```z[0, 0]``` is at X=0.5, Y=0.5 in the lower left corner.
- `'upper'`: ```z[0, 0]``` is at X=N+0.5, Y=0.5 in the upper left corner.
- `'image'`: Use the value from `:rc:`image.origin``.

`extent` : (x0, x1, y0, y1), optional

If `*origin*` is not `*None*`, then `*extent*` is interpreted as in `.imshow``: it gives the outer pixel boundaries. In this case, the position of `z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then `(*x0*, *y0*)` is the position of `z[0, 0]`, and `(*x1*, *y1*)` is the position of `z[-1, -1]`.

This argument is ignored if **X** and **Y** are specified in the call to `contour`.

`locator` : `ticker.Locator` subclass, optional

The locator is used to determine the contour levels if they are not given explicitly via **levels**.

Defaults to `~.ticker.MaxNLocator`.

`extend` : {'neither', 'both', 'min', 'max'}, default: 'neither'

Determines the ```tricontourf```-coloring of values that are outside the **levels** range.

If 'neither', values outside the **levels** range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the **levels** range.

Values below ```min(levels)``` and above ```max(levels)``` are mapped to the under/over values of the ``.Colormap``. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using ``.Colormap.set_under`` and ``.Colormap.set_over``.

.. note::

An existing ``.TriContourSet`` does not get notified if properties of its colormap are changed. Therefore, an explicit call to ``.ContourSet.changed()`` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the ``.TriContourSet`` because it internally calls ``.ContourSet.changed()``.

`xunits`, `yunits` : registered units, optional

Override axis units by specifying an instance of a :class:`matplotlib.units.ConversionInterface`.

`antialiased` : bool, optional

Enable antialiasing, overriding the defaults. For filled contours, the default is **True**. For line contours, it is taken from `:rc:`lines.antialiased``.

`hatches` : list[str], optional

A list of crosshatch patterns to use on the filled areas. If None, no hatching will be added to the contour.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``.axes.Axes.tricontourf``.

``.tricontourf`` fills intervals that are closed at the top; that is, for boundaries **z1** and **z2**, the filled region is::

$z1 < Z \leq z2$

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

matplotlib.pyplot.tripcolor

```
tripcolor(*args, alpha=1.0, norm=None, cmap=None, vmin=None, vmax=None,
          shading='flat', facecolors=None, **kwargs)
```

Create a pseudocolor plot of an unstructured triangular grid.

Call signatures::

```
tripcolor(triangulation, c, *, ...)
tripcolor(x, y, c, *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a `.Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. See `.Triangulation`` for an explanation of these parameters.

It is possible to pass the triangles positionally, i.e. `tripcolor(x, y, triangles, c, ...)``. However, this is discouraged. For more clarity, pass `*triangles*` via keyword argument.

If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly. In this case, it does not make sense to provide colors at the triangle faces via `*c*` or `*facecolors*` because there are multiple possible triangulations for a group of points and you don't know which triangles will be constructed.

Parameters

triangulation : `.Triangulation``

An already created triangular grid.

x, y, triangles, mask

Parameters defining the triangular grid. See `.Triangulation``.

This is mutually exclusive with specifying `*triangulation*`.

c : array-like

The color values, either for the points or for the triangles. Which one is automatically inferred from the length of `*c*`, i.e. does it match the number of points or the number of triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the keyword argument `facecolors=c`` instead of just `c``.

This parameter is position-only.

facecolors : array-like, optional

Can be used alternatively to `*c*` to specify colors at the triangle faces. This parameter takes precedence over `*c*`.

shading : {'flat', 'gouraud'}, default: 'flat'

If 'flat' and the color values `*c*` are defined at points, the color values used for each triangle are from the mean c of the triangle's three points. If `*shading*` is 'gouraud' then color values must be defined at points.

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``
The Colormap instance or registered colormap name used to map scalar data to colors.

`norm` : str or `~matplotlib.colors.Normalize``, optional
The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

`vmin`, `vmax` : float, optional
When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

`colorizer` : `~matplotlib.colorbar.Colorizer`` or None, default: None
The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

Returns

`~matplotlib.collections.PolyCollection`` or `~matplotlib.collections.TriMesh``
The result depends on `*shading*`: For ```shading='flat'``` the result is a `.PolyCollection``, for ```shading='gouraud'``` the result is a `.TriMesh``.

Other Parameters

`**kwargs` : `~matplotlib.collections.Collection`` properties

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
`alpha`: array-like or float or None
`animated`: bool
`antialiased` or `aa` or `antialiaseds`: bool or list of bools
`array`: array-like or None
`capstyle`: `.CapStyle`` or {'butt', 'projecting', 'round'}
`clim`: (vmin: float, vmax: float)
`clip_box`: `~matplotlib.transforms.BboxBase`` or None
`clip_on`: bool
`clip_path`: Patch or (Path, Transform) or None
`cmap`: `.Colormap`` or str or None
`color`: `:mpltype:`color`` or list of RGBA tuples
`edgecolor` or `ec` or `edgcolors`: `:mpltype:`color`` or list of `:mpltype:`color`` or 'face'
`facecolor` or `facecolors` or `fc`: `:mpltype:`color`` or list of `:mpltype:`color``

figure: ``~matplotlib.figure.Figure`` or ``~matplotlib.figure.SubFigure``
gid: str
hatch: `{'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}`
hatch_linewidth: unknown
in_layout: bool
joinstyle: ``~.JoinStyle`` or `{'miter', 'round', 'bevel'}`
label: object
linestyle or dashes or linestyles or ls: str or tuple or list thereof
linewidth or linewidths or lw: float or list of floats
mouseover: bool
norm: ``~.Normalize`` or str or None
offset_transform or transOffset: ``~.Transform``
offsets: (N, 2) or (2,) array-like
path_effects: list of ``~.AbstractPathEffect``
paths: unknown
picker: None or bool or float or callable
pickradius: float
rasterized: bool
sketch_params: (scale: float, length: float, randomness: float)
snap: bool or None
transform: ``~matplotlib.transforms.Transform``
url: str
urls: list of str or None
visible: bool
zorder: float

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``~.axes.Axes.tripcolor``.

matplotlib.pyplot.tripplot

```
tripplot(*args, **kwargs)
```

Draw an unstructured triangular grid as lines and/or markers.

Call signatures::

```
tripplot(triangulation, ...)
tripplot(x, y, [triangles], *, [mask=mask], ...)
```

The triangular grid can be specified either by passing a ``~.Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly.

Parameters

triangulation : ``~.Triangulation``
An already created triangular grid.
x, y, triangles, mask

Parameters defining the triangular grid. See `~.Triangulation`.
This is mutually exclusive with specifying `*triangulation*`.
`other_parameters`
All other args and kwargs are forwarded to `~.Axes.plot`.

Returns

lines : `~matplotlib.lines.Line2D`
The drawn triangles edges.
markers : `~matplotlib.lines.Line2D`
The drawn marker nodes.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `~.axes.Axes.triplot`.

matplotlib.pyplot.twinx

```
twinx(ax: 'matplotlib.axes.Axes | None' = None) -> '_AxesBase'
```

Make and return a second Axes that shares the `*x*`-axis. The new Axes will overlay `*ax*` (or the current Axes if `*ax*` is `*None*`), and its ticks will be on the right.

Examples

:doc:`/gallery/subplots_axes_and_figures/two_scales`

matplotlib.pyplot.twiny

```
twiny(ax: 'matplotlib.axes.Axes | None' = None) -> '_AxesBase'
```

Make and return a second Axes that shares the `*y*`-axis. The new Axes will overlay `*ax*` (or the current Axes if `*ax*` is `*None*`), and its ticks will be on the top.

Examples

:doc:`/gallery/subplots_axes_and_figures/two_scales`

matplotlib.pyplot.uninstall_repl_displayhook

```
uninstall_repl_displayhook() -> 'None'
```

Disconnect from the display hook of the current shell.

matplotlib.pyplot.violinplot

```
violinplot(dataset: 'ArrayLike | Sequence[ArrayLike]', positions: 'ArrayLike | None' =
None, *, vert: 'bool | None' = None, orientation: "Literal['vertical', 'horizontal']"
= 'vertical', widths: 'float | ArrayLike' = 0.5, showmeans: 'bool' = False,
showextrema: 'bool' = True, showmedians: 'bool' = False, quantiles: 'Sequence[float |
Sequence[float]] | None' = None, points: 'int' = 100, bw_method: "Literal['scott',
'silverman'] | float | Callable[[GaussianKDE], float] | None" = None, side:
"Literal['both', 'low', 'high']" = 'both', data=None) -> 'dict[str, Collection]'
```

Make a violin plot.

Make a violin plot for each column of **dataset** or each vector in sequence **dataset**. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, the maximum, and user-specified quantiles.

Parameters

dataset : Array or a sequence of vectors.
The input data.

positions : array-like, default: [1, 2, ..., n]
The positions of the violins; i.e. coordinates on the x-axis for vertical violins (or y-axis for horizontal violins).

vert : bool, optional
.. deprecated:: 3.10
Use **orientation** instead.

If this is given during the deprecation period, it overrides the **orientation** parameter.

If True, plots the violins vertically.
If False, plots the violins horizontally.

orientation : {'vertical', 'horizontal'}, default: 'vertical'
If 'horizontal', plots the violins horizontally.
Otherwise, plots the violins vertically.

.. versionadded:: 3.10

widths : float or array-like, default: 0.5
The maximum width of each violin in units of the **positions** axis.
The default is 0.5, which is half the available space when using default **positions**.

showmeans : bool, default: False
Whether to show the mean with a line.

showextrema : bool, default: True
Whether to show extrema with a line.

showmedians : bool, default: False
Whether to show the median with a line.

quantiles : array-like, default: None
If not None, set a list of floats in interval [0, 1] for each violin, which stands for the quantiles that will be rendered for that

violin.

points : int, default: 100

The number of points to evaluate each of the gaussian kernel density estimations at.

bw_method : {'scott', 'silverman'} or float or callable, default: 'scott'

The method used to calculate the estimator bandwidth. If a float, this will be used directly as `kde.factor`. If a callable, it should take a `matplotlib.mlab.GaussianKDE` instance as its only parameter and return a float.

side : {'both', 'low', 'high'}, default: 'both'

'both' plots standard violins. 'low'/'high' only plots the side below/above the positions value.

data : indexable object, optional

If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` if `s` is a key in `data`:

dataset

Returns

dict

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the `~.collections.PolyCollection` instances containing the filled area of each violin.

- `cmeans`: A `~.collections.LineCollection` instance that marks the mean values of each of the violin's distribution.

- `cmins`: A `~.collections.LineCollection` instance that marks the bottom of each violin's distribution.

- `cmaxes`: A `~.collections.LineCollection` instance that marks the top of each violin's distribution.

- `cbars`: A `~.collections.LineCollection` instance that marks the centers of each violin's distribution.

- `cmedians`: A `~.collections.LineCollection` instance that marks the median values of each of the violin's distribution.

- `cquantiles`: A `~.collections.LineCollection` instance created to identify the quantile values of each of the violin's distribution.

See Also

`.Axes.violin` : Draw a violin from pre-computed statistics.

`boxplot` : Draw a box and whisker plot.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.violinplot``.

matplotlib.pyplot.viridis

```
viridis() -> 'None'
```

Set the colormap to 'viridis'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

matplotlib.pyplot.vlines

```
vlines(x: 'float | ArrayLike', ymin: 'float | ArrayLike', ymax: 'float | ArrayLike',
       colors: 'ColorType | Sequence[ColorType] | None' = None, linestyles: 'LineStyleType' =
       'solid', *, label: 'str' = '', data=None, **kwargs) -> 'LineCollection'
```

Plot vertical lines at each `*x*` from `*ymin*` to `*ymax*`.

Parameters

`x` : float or array-like

x-indexes where to plot the lines.

`ymin`, `ymax` : float or array-like

Respective beginning and end of each line. If scalars are provided, all lines will have the same length.

`colors` : :mpltype:`color` or list of color, default: `:rc:`lines.color``

`linestyles` : {'solid', 'dashed', 'dashdot', 'dotted'}, default: 'solid'

`label` : str, default: ''

Returns

`~matplotlib.collections.LineCollection``

Other Parameters

`data` : indexable object, optional

If given, the following parameters also accept a string ```s```, which is interpreted as ```data[s]``` if ```s``` is a key in ```data```:

`*x*`, `*ymin*`, `*ymax*`, `*colors*`

`**kwargs` : `~matplotlib.collections.LineCollection`` properties.

See Also

hlines : horizontal lines
axvline : vertical line across the Axes

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.vlines``.

matplotlib.pyplot.waitforbuttonpress

```
waitforbuttonpress(timeout: 'float' = -1) -> 'None | bool'
```

Blocking call to interact with the figure.

Wait for user input and return True if a key was pressed, False if a mouse button was pressed and None if no input was given within `*timeout*` seconds. Negative values deactivate `*timeout*`.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.Figure.waitforbuttonpress``.

matplotlib.pyplot.winter

```
winter() -> 'None'
```

Set the colormap to 'winter'.

This changes the default colormap as well as the colormap of the current image if there is one. See ```help(colormaps)``` for more information.

matplotlib.pyplot.xcorr

```
xcorr(x: 'ArrayLike', y: 'ArrayLike', *, normed: 'bool' = True, detrend:  
'Callable[[ArrayLike], ArrayLike]' = , usevlines: 'bool' = True, maxlags: 'int' = 10,  
data=None, **kwargs) -> 'tuple[np.ndarray, np.ndarray, LineCollection | Line2D, Line2D  
| None]'
```

Plot the cross correlation between `*x*` and `*y*`.

The correlation with lag `k` is defined as
:math:\sum_n x[n+k] \cdot y^*[n], where :math:y^* is the complex
conjugate of :math:y.

Parameters

`x, y` : array-like of length `n`

Neither `*x*` nor `*y*` are run through Matplotlib's unit conversion, so these should be unit-less arrays.

detrend : callable, default: ``mlab.detrend_none`` (no detrending)
A detrending function applied to `*x*` and `*y*`. It must have the signature ::

`detrend(x: np.ndarray) -> np.ndarray`

normed : bool, default: True

If `True``, input vectors are normalised to unit length.

usevlines : bool, default: True

Determines the plot style.

If `True``, vertical lines are plotted from 0 to the xcorr value using ``Axes.vlines``. Additionally, a horizontal line is plotted at `y=0` using ``Axes.axhline``.

If `False``, markers are plotted at the xcorr values using ``Axes.plot``.

maxlags : int, default: 10

Number of lags to show. If None, will return all `2 * len(x) - 1`` lags.

Returns

lags : array (length `2*maxlags+1``)

The lag vector.

c : array (length `2*maxlags+1``)

The auto correlation vector.

line : ``LineCollection`` or ``Line2D``

``Artist`` added to the Axes of the correlation:

- ``LineCollection`` if `*usevlines*` is True.

- ``Line2D`` if `*usevlines*` is False.

b : ``~matplotlib.lines.Line2D`` or None

Horizontal line at 0 if `*usevlines*` is True

None `*usevlines*` is False.

Other Parameters

linestyle : ``~matplotlib.lines.Line2D`` property, optional

The linestyle for plotting the data points.

Only used if `*usevlines*` is `False``.

marker : str, default: 'o'

The marker for plotting the data points.

Only used if `*usevlines*` is `False``.

data : indexable object, optional

If given, the following parameters also accept a string `s``, which is interpreted as `data[s]` if `s`` is a key in `data``:

`*x*`, `*y*`

`**kwargs`

Additional parameters are passed to `.Axes.vlines` and `.Axes.axhline` if `*usevlines*` is `True`; otherwise they are passed to `.Axes.plot`.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for `.axes.Axes.xcorr`.

The cross correlation is performed with `numpy.correlate` with `mode = "full"`.

matplotlib.pyplot.xkcd

```
xkcd(scale: 'float' = 1, length: 'float' = 100, randomness: 'float' = 2) ->
'ExitStack'
```

Turn on `xkcd` <<https://xkcd.com/>> `_` sketch-style drawing mode.

This will only have an effect on things drawn after this function is called.

For best results, install the `xkcd` script <<https://github.com/ipython/xkcd-font/>> `_` font; `xkcd` fonts are not packaged with Matplotlib.

Parameters

`scale` : float, optional

The amplitude of the wiggle perpendicular to the source line.

`length` : float, optional

The length of the wiggle along the line.

`randomness` : float, optional

The scale factor by which the length is shrunk or expanded.

Notes

This function works by a number of rcParams, so it will probably override others you have set before.

If you want the effects of this function to be temporary, it can be used as a context manager, for example::

```
with plt.xkcd():
```

```
# This figure will be in XKCD-style
```

```
fig1 = plt.figure()
```

```
# ...
```

```
# This figure will be in regular style
```

```
fig2 = plt.figure()
```

matplotlib.pyplot.xlabel

```
xlabel(xlabel: 'str', fontdict: 'dict[str, Any] | None' = None, labelpad: 'float | None' = None, *, loc: "Literal['left', 'center', 'right'] | None" = None, **kwargs) -> 'Text'
```

Set the label for the x-axis.

Parameters

xlabel : str

The label text.

labelpad : float, default: :rc:`axes.labelpad`

Spacing in points from the Axes bounding box including ticks and tick labels. If None, the previous value is left as is.

loc : {'left', 'center', 'right'}, default: :rc:`xaxis.labellocation`

The label position. This is a high-level alternative for passing parameters **x** and **horizontalalignment**.

Other Parameters

****kwargs** : ~matplotlib.text.Text` properties

`.Text` properties control the appearance of the label.

See Also

text : Documents the properties supported by ``.Text``.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``.axes.Axes.set_xlabel``.

matplotlib.pyplot.xlim

```
xlim(*args, **kwargs) -> 'tuple[float, float]'
```

Get or set the x limits of the current Axes.

Call signatures::

left, right = xlim() # return the current xlim

xlim((left, right)) # set the xlim to left, right

xlim(left, right) # set the xlim to left, right

If you do not specify args, you can pass **left** or **right** as kwargs, i.e.::

xlim(right=3) # adjust the right leaving left unchanged

xlim(left=1) # adjust the left leaving right unchanged

Setting limits turns autoscaling off for the x-axis.

Returns

left, right
A tuple of the new x-axis limits.

Notes

Calling this function with no arguments (e.g. ``xlim()``) is the pyplot equivalent of calling ``~.Axes.get_xlim`` on the current Axes.
Calling this function with arguments is the pyplot equivalent of calling ``~.Axes.set_xlim`` on the current Axes. All arguments are passed though.

matplotlib.pyplot.xscale

```
xscale(value: 'str | ScaleBase', **kwargs) -> 'None'
```

Set the xaxis' scale.

Parameters

value : str or ``ScaleBase``

The axis scale type to apply. Valid string values are the names of scale classes ("linear", "log", "function",...). These may be the names of any of the :ref:`built-in scales<builtin_scales>` or of any custom scales registered using ``matplotlib.scale.register_scale``.

**kwargs

If *value* is a string, keywords are passed to the instantiation method of the respective class.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``~.axes.Axes.set_xscale``.

matplotlib.pyplot.xticks

```
xticks(ticks: 'ArrayLike | None' = None, labels: 'Sequence[str] | None' = None, *,  
minor: 'bool' = False, **kwargs) -> 'tuple[list[Tick] | np.ndarray, list[Text]]'
```

Get or set the current tick locations and labels of the x-axis.

Pass no arguments to return the current values without modifying them.

Parameters

ticks : array-like, optional

The list of xtick locations. Passing an empty list removes all xticks.

labels : array-like, optional

The labels to place at the given *ticks* locations. This argument can only be passed if *ticks* is passed as well.

minor : bool, default: False

If ``False``, get/set the major ticks/labels; if ``True``, the minor ticks/labels.

****kwargs**
`.Text` properties can be used to control the appearance of the labels.

.. warning::

This only sets the properties of the current ticks, which is only sufficient if you either pass `*ticks*`, resulting in a fixed list of ticks, or if the plot is static.

Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).

Use `~.pyplot.tick_params`` instead if possible.

Returns

locs

The list of xtick locations.

labels

The list of xlabel `.Text`` objects.

Notes

Calling this function with no arguments (e.g. `xticks()`) is the pyplot equivalent of calling `~.Axes.get_xticks`` and `~.Axes.get_xticklabels`` on the current Axes.

Calling this function with arguments is the pyplot equivalent of calling `~.Axes.set_xticks`` and `~.Axes.set_xticklabels`` on the current Axes.

Examples

```
>>> locs, labels = xticks() # Get the current locations and labels.
>>> xticks(np.arange(0, 1, step=0.2)) # Set label locations.
>>> xticks(np.arange(3), ['Tom', 'Dick', 'Sue']) # Set text labels.
>>> xticks([0, 1, 2], ['January', 'February', 'March'],
... rotation=20) # Set text labels and properties.
>>> xticks([]) # Disable xticks.
```

matplotlib.pyplot.ylabel

```
ylabel(ylabel: 'str', fontdict: 'dict[str, Any] | None' = None, labelpad: 'float | None' = None, *, loc: "Literal['bottom', 'center', 'top'] | None" = None, **kwargs) -> 'Text'
```

Set the label for the y-axis.

Parameters

ylabel : str

The label text.

`labelpad` : float, default: `:rc:`axes.labelpad``
Spacing in points from the Axes bounding box including ticks and tick labels. If None, the previous value is left as is.

`loc` : {'bottom', 'center', 'top'}, default: `:rc:`yaxis.labellocation``
The label position. This is a high-level alternative for passing parameters `*y*` and `*horizontalalignment*`.

Other Parameters

`**kwargs` : `~matplotlib.text.Text`` properties
`.Text`` properties control the appearance of the label.

See Also

`text` : Documents the properties supported by `.Text``.

Notes

.. note::

This is the `:ref:`pyplot wrapper <pyplot_interface>`` for `.axes.Axes.set_ylabel``.

matplotlib.pyplot.ylim

```
ylim(*args, **kwargs) -> 'tuple[float, float]'
```

Get or set the y-limits of the current Axes.

Call signatures::

```
bottom, top = ylim() # return the current ylim  
ylim((bottom, top)) # set the ylim to bottom, top  
ylim(bottom, top) # set the ylim to bottom, top
```

If you do not specify args, you can alternatively pass `*bottom*` or `*top*` as kwargs, i.e.::

```
ylim(top=3) # adjust the top leaving bottom unchanged  
ylim(bottom=1) # adjust the bottom leaving top unchanged
```

Setting limits turns autoscaling off for the y-axis.

Returns

bottom, top
A tuple of the new y-axis limits.

Notes

Calling this function with no arguments (e.g. ```ylim()```) is the pyplot equivalent of calling `~.Axes.get_ylim`` on the current Axes.
Calling this function with arguments is the pyplot equivalent of calling

``~.Axes.set_ylim`` on the current Axes. All arguments are passed though.

matplotlib.pyplot.yscale

```
yscale(value: 'str | ScaleBase', **kwargs) -> 'None'
```

Set the yaxis' scale.

Parameters

value : str or ``~.ScaleBase``

The axis scale type to apply. Valid string values are the names of scale classes ("linear", "log", "function",...). These may be the names of any of the :ref:`built-in scales<builtin_scales>` or of any custom scales registered using ``matplotlib.scale.register_scale``.

****kwargs**

If **value** is a string, keywords are passed to the instantiation method of the respective class.

Notes

.. note::

This is the :ref:`pyplot wrapper <pyplot_interface>` for ``~.axes.Axes.set_yscale``.

matplotlib.pyplot.yticks

```
yticks(ticks: 'ArrayLike | None' = None, labels: 'Sequence[str] | None' = None, *, minor: 'bool' = False, **kwargs) -> 'tuple[list[Tick] | np.ndarray, list[Text]]'
```

Get or set the current tick locations and labels of the y-axis.

Pass no arguments to return the current values without modifying them.

Parameters

ticks : array-like, optional

The list of ytick locations. Passing an empty list removes all yticks.

labels : array-like, optional

The labels to place at the given **ticks** locations. This argument can only be passed if **ticks** is passed as well.

minor : bool, default: False

If ``False``, get/set the major ticks/labels; if ``True``, the minor ticks/labels.

****kwargs**

``~.Text`` properties can be used to control the appearance of the labels.

.. warning::

This only sets the properties of the current ticks, which is only sufficient if you either pass **ticks**, resulting in a fixed list of ticks, or if the plot is static.

Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).

Use `~.pyplot.tick_params`` instead if possible.

Returns

locs

The list of ytick locations.

labels

The list of ylabel ``Text`` objects.

Notes

Calling this function with no arguments (e.g. ``yticks()``) is the pyplot equivalent of calling `~.Axes.get_yticks`` and `~.Axes.get_yticklabels`` on the current Axes.

Calling this function with arguments is the pyplot equivalent of calling `~.Axes.set_yticks`` and `~.Axes.set_yticklabels`` on the current Axes.

Examples

```
>>> locs, labels = yticks() # Get the current locations and labels.
>>> yticks(np.arange(0, 1, step=0.2)) # Set label locations.
>>> yticks(np.arange(3), ['Tom', 'Dick', 'Sue']) # Set text labels.
>>> yticks([0, 1, 2], ['January', 'February', 'March'],
... rotation=45) # Set text labels and properties.
>>> yticks([]) # Disable yticks.
```

matplotlib.quiver

```
quiver(...)
```

Support for plotting vector fields.

Presently this contains Quiver and Barb. Quiver plots an arrow in the direction of the vector, with the size of the arrow related to the magnitude of the vector.

Barbs are like quiver in that they point along a vector, but the magnitude of the vector is given schematically by the presence of barbs or flags on the barb.

This will also become a home for things such as standard deviation ellipses, which can and will be derived very easily from the Quiver code.

matplotlib.quiver.Barbs

```
Barbs(ax, *args, pivot='tip', length=7, barbcolor=None, flagcolor=None, sizes=None,
fill_empty=False, barb_increments=None, rounding=True, flip_barb=False, **kwargs)
```

Specialized PolyCollection for barbs.

The only API method is :meth:`set_UVC`, which can be used to change the size, orientation, and color of the arrows. Locations are changed using the :meth:`set_offsets` collection method. Possibly this method will be useful in animations.

There is one internal function :meth:`_find_tails` which finds exactly what should be put on the barb given the vector magnitude. From there :meth:`_make_barbs` is used to find the vertices of the polygon to represent the barb based on this information.

matplotlib.quiver.CirclePolygon

```
CirclePolygon(xy, radius=5, *, resolution=20, **kwargs)
```

A polygon-approximation of a circle patch.

matplotlib.quiver.Quiver

```
Quiver(ax, *args, scale=None, headwidth=3, headlength=5, headaxislength=4.5,  
minshaft=1, minlength=1, units='width', scale_units=None, angles='uv', width=None,  
color='k', pivot='tail', **kwargs)
```

Specialized PolyCollection for arrows.

The only API method is set_UVC(), which can be used to change the size, orientation, and color of the arrows; their locations are fixed when the class is instantiated. Possibly this method will be useful in animations.

Much of the work in this class is done in the draw() method so that as much information as possible is available about the plot. In subsequent draw() calls, recalculation is limited to things that might have changed, so there should be no performance penalty from putting the calculations in the draw() method.

matplotlib.quiver.QuiverKey

```
QuiverKey(Q, X, Y, U, label, *, angle=0, coordinates='axes', color=None, labelsep=0.1,  
labelpos='N', labelcolor=None, fontproperties=None, zorder=None, **kwargs)
```

Labelled arrow for use as a quiver plot scale key.

matplotlib.rc

```
rc(group, **kwargs)
```

Set the current `rcParams`. *group* is the grouping for the rc, e.g., for ``lines.linewidth`` the group is ``lines``, for ``axes.facecolor``, the group is ``axes``, and so on. Group may also be a list or tuple of group names, e.g., (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, e.g.,::

```
rc('lines', linewidth=2, color='r')
```

sets the current `.rcParams`` and is equivalent to::

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

```
=====
Alias Property
=====
'lw' 'linewidth'
'ls' 'linestyle'
'c' 'color'
'fc' 'facecolor'
'ec' 'edgecolor'
'mew' 'markeredgewidth'
'aa' 'antialiased'
=====
```

Thus you could abbreviate the above call as::

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows::

```
font = {'family' : 'monospace',
'weight' : 'bold',
'size' : 'larger'}
rc('font', **font) # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use ```matplotlib.style.use('default')``` or `:func:`~matplotlib.rcdefaults`` to restore the default `.rcParams`` after changes.

Notes

Similar functionality is available by using the normal dict interface, i.e. ```rcParams.update({"lines.linewidth": 2, ...})``` (but ```rcParams.update``` does not support abbreviations or grouping).

matplotlib.rc_context

```
rc_context(rc=None, fname=None)
```

Return a context manager for temporarily changing rcParams.

The `:rc:`backend`` will not be reset by the context manager.

rcParams changed both through the context manager invocation and in the body of the context will be reset on context exit.

Parameters

rc : dict

The rcParams to temporarily set.

fname : str or path-like

A file with Matplotlib rc settings. If both *fname* and *rc* are given, settings from *rc* take precedence.

See Also

:ref:`customizing-with-matplotlibrc-files`

Examples

Passing explicit values via a dict::

```
with mpl.rc_context({'interactive': False}):
    fig, ax = plt.subplots()
    ax.plot(range(3), range(3))
    fig.savefig('example.png')
    plt.close(fig)
```

Loading settings from a file::

```
with mpl.rc_context(fname='print.rc'):
    plt.plot(x, y) # uses 'print.rc'
```

Setting in the context body::

```
with mpl.rc_context():
    # will be reset
    mpl.rcParams['lines.linewidth'] = 5
    plt.plot(x, y)
```

matplotlib.rc_file

```
rc_file(fname, *, use_default_template=True)
```

Update `.rcParams` from file.

Style-blacklisted `.rcParams` (defined in `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

Parameters

fname : str or path-like

A file with Matplotlib rc settings.

use_default_template : bool

If True, initialize with default parameters before updating with those in the given file. If False, the current configuration persists and only the parameters specified in the file are updated.

matplotlib.rc_file_defaults

```
rc_file_defaults()
```

Restore the `.rcParams`` from the original rc file loaded by Matplotlib.

Style-blacklisted `.rcParams`` (defined in ```matplotlib.style.core.STYLE_BLACKLIST```) are not updated.

matplotlib.rc_params

```
rc_params(fail_on_error=False)
```

Construct a ``RcParams`` instance from the default Matplotlib rc file.

matplotlib.rc_params_from_file

```
rc_params_from_file(fname, fail_on_error=False, use_default_template=True)
```

Construct a ``RcParams`` from file `*fname*`.

Parameters

`fname` : str or path-like

A file with Matplotlib rc settings.

`fail_on_error` : bool

If True, raise an error when the parser fails to convert a parameter.

`use_default_template` : bool

If True, initialize with default parameters before updating with those in the given file. If False, the configuration class only contains the parameters specified in the file. (Useful for updating dicts.)

matplotlib.rcdefaults

```
rcdefaults()
```

Restore the `.rcParams`` from Matplotlib's internal default style.

Style-blacklisted `.rcParams`` (defined in ```matplotlib.style.core.STYLE_BLACKLIST```) are not updated.

See Also

`matplotlib.rc_file_defaults`

Restore the `.rcParams`` from the rc file originally loaded by Matplotlib.

`matplotlib.style.use`

Use a specific style file. Call ```style.use('default')``` to restore the default style.

matplotlib.rcsetup

```
rcsetup(...)
```

The `rcsetup` module contains the validation code for customization using Matplotlib's rc settings.

Each rc setting is assigned a function used to validate any attempted changes to that setting. The validation functions are defined in the rcsetup module, and are used to construct the rcParams global object which stores the settings and is referenced throughout Matplotlib.

The default values of the rc settings are set in the default matplotlibrc file. Any additions or deletions to the parameter set listed here should also be propagated to the :file:`lib/matplotlib/mpl-data/matplotlibrc` in Matplotlib's root source directory.

matplotlib.rcsetup.BackendFilter

```
BackendFilter(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Filter used with :meth:`~matplotlib.backends.registry.BackendRegistry.list_builtin`

.. versionadded:: 3.9

matplotlib.rcsetup.CapStyle

```
CapStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the two endpoints (caps) of an unclosed line are drawn.

How to draw the start and end points of lines that represent a closed curve (i.e. that end in a `~.path.Path.CLOSEPOLY`) is controlled by the line's `JoinStyle`. For all other lines, how the start and end points are drawn is controlled by the `*CapStyle*`.

For a visual impression of each `*CapStyle*`, view these docs online `<CapStyle>` or run `CapStyle.demo`.

By default, `~.backend_bases.GraphicsContextBase` draws a stroked line as squared off at its endpoints.

Supported values:

.. rst-class:: value-list

'butt'

the line is squared off at its endpoint.

'projecting'

the line is squared off as in `*butt*`, but the filled in area extends beyond the endpoint a distance of `linewidth/2`.

'round'

like `*butt*`, but a semicircular cap is added to the end of the line, of radius `linewidth/2`.

.. plot::

:alt: Demo of possible CapStyle's

```
from matplotlib._enums import CapStyle
```

CapStyle.demo()

matplotlib.rcsetup.Colormap

```
Colormap(name, N=256)
```

Baseclass for all scalar to RGBA mappings.

Typically, Colormap instances are used to convert data values (floats) from the interval `[0, 1]` to the RGBA color that the respective Colormap represents. For scaling of data into the `[0, 1]` interval see `matplotlib.colors.Normalize`. Subclasses of `matplotlib.cm.ScalarMappable` make heavy use of this `data -> normalize -> map-to-color` processing chain.

matplotlib.rcsetup.JoinStyle

```
JoinStyle(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
```

Define how the connection between two line segments is drawn.

For a visual impression of each `*JoinStyle*`, view these docs online `<JoinStyle>`, or run `JoinStyle.demo`.

Lines in Matplotlib are typically defined by a 1D `~.path.Path` and a finite `linewidth`, where the underlying 1D `~.path.Path` represents the center of the stroked line.

By default, `~.backend_bases.GraphicsContextBase` defines the boundaries of a stroked line to simply be every point within some radius, `linewidth/2`, away from any point of the center line. However, this results in corners appearing "rounded", which may not be the desired behavior if you are drawing, for example, a polygon or pointed star.

****Supported values:****

.. rst-class:: value-list

'miter'

the "arrow-tip" style. Each boundary of the filled-in area will extend in a straight line parallel to the tangent vector of the centerline at the point it meets the corner, until they meet in a sharp point.

'round'

stokes every point within a radius of `linewidth/2` of the center lines.

'bevel'

the "squared-off" style. It can be thought of as a rounded corner where the "circular" part of the corner has been cut off.

.. note::

Very long miter tips are cut off (to form a `*bevel*`) after a backend-dependent limit called the "miter limit", which specifies the

maximum allowed ratio of miter length to line width. For example, the PDF backend uses the default value of 10 specified by the PDF standard, while the SVG backend does not even specify the miter limit, resulting in a default value of 4 per the SVG specification. Matplotlib does not currently allow the user to adjust this parameter.

A more detailed description of the effect of a miter limit can be found in the `Mozilla Developer Docs`
<<https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-miterlimit>>`_

.. plot::
:alt: Demo of possible JoinStyle's

```
from matplotlib._enums import JoinStyle
JoinStyle.demo()
```

matplotlib.rcsetup.ValidateInStrings

```
ValidateInStrings(key, valid, ignorecase=False, *, _deprecated_since=None)
```

No description available.

matplotlib.rcsetup.cycler

```
cycler(*args, **kwargs)
```

Create a `~matplotlib.cycler.Cycler` object much like :func:`matplotlib.cycler`, but includes input validation.

Call signatures::

```
cycler(cycler)
cycler(label=values, label2=values2, ...)
cycler(label, values)
```

Form 1 copies a given `~matplotlib.cycler.Cycler` object.

Form 2 creates a `~matplotlib.cycler.Cycler` which cycles over one or more properties simultaneously. If multiple properties are given, their value lists must have the same length.

Form 3 creates a `~matplotlib.cycler.Cycler` for a single property. This form exists for compatibility with the original cycler. Its use is discouraged in favor of the kwarg form, i.e. ``cycler(label=values)``.

Parameters

cycler : Cycler
Copy constructor for Cycler.

label : str
The property key. Must be a valid `Artist` property.
For example, 'color' or 'linestyle'. Aliases are allowed, such as 'c' for 'color' and 'lw' for 'linewidth'.

values : iterable

Finite-length iterable of the property values. These values are validated and will raise a `ValueError` if invalid.

Returns

Cycler

A new `:class:`~cycler.Cycler`` for the given properties.

Examples

Creating a cycler for a single property:

```
>>> c = cycler(color=['red', 'green', 'blue'])
```

Creating a cycler for simultaneously cycling over multiple properties (e.g. red circle, green plus, blue cross):

```
>>> c = cycler(color=['red', 'green', 'blue'],  
... marker=['o', '+', 'x'])
```

matplotlib.rcsetup.is_color_like

```
is_color_like(c)
```

Return whether `*c*` can be interpreted as an RGB(A) color.

matplotlib.rcsetup.validate_any

```
validate_any(s)
```

No description available.

matplotlib.rcsetup.validate_anymap

```
validate_anymap(s)
```

No description available.

matplotlib.rcsetup.validate_aspect

```
validate_aspect(s)
```

No description available.

matplotlib.rcsetup.validate_axisbelow

```
validate_axisbelow(s)
```

No description available.

matplotlib.rcsetup.validate_backend

```
validate_backend(s)
```

No description available.

matplotlib.rcsetup.validate_bbox

```
validate_bbox(s)
```

No description available.

matplotlib.rcsetup.validate_bool

```
validate_bool(b)
```

Convert b to ``bool`` or raise.

matplotlib.rcsetup.validate_color

```
validate_color(s)
```

Return a valid color arg.

matplotlib.rcsetup.validate_color_for_prop_cycle

```
validate_color_for_prop_cycle(s)
```

No description available.

matplotlib.rcsetup.validate_color_or_auto

```
validate_color_or_auto(s)
```

No description available.

matplotlib.rcsetup.validate_color_or_inherit

```
validate_color_or_inherit(s)
```

Return a valid color arg.

matplotlib.rcsetup.validate_colorlist

```
validate_colorlist(s)
```

return a list of colorspecs

matplotlib.rcsetup.validate_cycler

```
validate_cycler(s)
```

Return a Cycler object from a string repr or the object itself.

matplotlib.rcsetup.validate_dashlist

```
validate_dashlist(s)
```

return a list of floats

matplotlib.rcsetup.validate_dpi

```
validate_dpi(s)
```

Confirm s is string 'figure' or convert s to float or raise.

matplotlib.rcsetup.validate_fillstylelist

```
validate_fillstylelist(s)
```

No description available.

matplotlib.rcsetup.validate_float

```
validate_float(s)
```

No description available.

matplotlib.rcsetup.validate_float_or_None

```
validate_float_or_None(s)
```

No description available.

matplotlib.rcsetup.validate_floatlist

```
validate_floatlist(s)
```

return a list of floats

matplotlib.rcsetup.validate_font_properties

```
validate_font_properties(s)
```

No description available.

matplotlib.rcsetup.validate_fontsize

```
validate_fontsize(s)
```

No description available.

matplotlib.rcsetup.validate_fontsize_None

```
validate_fontsize_None(s)
```

No description available.

matplotlib.rcsetup.validate_fontsizelist

```
validate_fontsizelist(s)
```

No description available.

matplotlib.rcsetup.validate_fontstretch

```
validate_fontstretch(s)
```

No description available.

matplotlib.rcsetup.validate_fonttype

```
validate_fonttype(s)
```

Confirm that this is a Postscript or PDF font type that we know how to convert to.

matplotlib.rcsetup.validate_fontweight

```
validate_fontweight(s)
```

No description available.

matplotlib.rcsetup.validate_hatch

```
validate_hatch(s)
```

Validate a hatch pattern.
A hatch pattern string can have any sequence of the following characters: ``\ / | - + * . x o O``.

matplotlib.rcsetup.validate_hatchlist

```
validate_hatchlist(s)
```

Validate a hatch pattern.
A hatch pattern string can have any sequence of the following characters: ``\ / | - + * . x o O``.

matplotlib.rcsetup.validate_hist_bins

```
validate_hist_bins(s)
```

No description available.

matplotlib.rcsetup.validate_int

```
validate_int(s)
```

No description available.

matplotlib.rcsetup.validate_int_or_None

```
validate_int_or_None(s)
```

No description available.

matplotlib.rcsetup.validate_markevery

```
validate_markevery(s)
```

Validate the markevery property of a Line2D object.

Parameters

s : None, int, (int, int), slice, float, (float, float), or list[int]

Returns

None, int, (int, int), slice, float, (float, float), or list[int]

matplotlib.rcsetup.validate_markeverylist

```
validate_markeverylist(s)
```

Validate the markevery property of a Line2D object.

Parameters

s : None, int, (int, int), slice, float, (float, float), or list[int]

Returns

None, int, (int, int), slice, float, (float, float), or list[int]

matplotlib.rcsetup.validate_ps_distiller

```
validate_ps_distiller(s)
```

No description available.

matplotlib.rcsetup.validate_sketch

```
validate_sketch(s)
```

No description available.

matplotlib.rcsetup.validate_string

```
validate_string(s)
```

No description available.

matplotlib.rcsetup.validate_string_or_None

```
validate_string_or_None(s)
```

No description available.

matplotlib.rcsetup.validate_stringlist

```
validate_stringlist(s)
```

return a list of strings

matplotlib.rcsetup.validate_whiskers

```
validate_whiskers(s)
```

No description available.

matplotlib.sanitize_sequence

```
sanitize_sequence(data)
```

[*Deprecated*]

Notes

.. deprecated:: 3.10

Use matplotlib.cbook.sanitize_sequence instead.\

matplotlib.sankey

```
sankey(...)
```

Module for creating Sankey diagrams using Matplotlib.

matplotlib.sankey.Affine2D

```
Affine2D(matrix=None, **kwargs)
```

A mutable 2D affine transformation.

matplotlib.sankey.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- `*vertices*`: an (N, 2) float array of vertices
- `*codes*`: an N-length ``numpy.uint8`` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three ``CURVE4`` codes.

The code types are:

- ``STOP`` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- ``MOVETO`` : 1 vertex

Pick up the pen and move to the given vertex.

- ``LINETO`` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'.

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.sankey.PathPatch

```
PathPatch(path, **kwargs)
```

A general polycurve path patch.

matplotlib.sankey.Sankey

```
Sankey(ax=None, scale=1.0, unit='', format='%G', gap=0.25, radius=0.1, shoulder=0.03, offset=0.15, head_angle=100, margin=0.4, tolerance=1e-06, **kwargs)
```

Sankey diagram.

Sankey diagrams are a specific type of flow diagram, in which the width of the arrows is shown proportionally to the flow quantity. They are typically used to visualize energy or material or cost transfers between processes.

``Wikipedia (6/1/2011) <https://en.wikipedia.org/wiki/Sankey_diagram>``

matplotlib.scale

```
scale(...)
```

Scales define the distribution of data values on an axis, e.g. a log scaling.

The mapping is implemented through ``Transform`` subclasses.

The following scales are built-in:

.. `_builtin_scales`:

```
=====
=====
Name Class Transform Inverted transform
=====
=====
"asinh" `AsinhScale` `AsinhTransform` `InvertedAsinhTransform`
"function" `FuncScale` `FuncTransform` `FuncTransform`
"functionlog" `FuncScaleLog` `FuncTransform` + `LogTransform` `InvertedLogTransform` + `FuncTransform`
"linear" `LinearScale` `.IdentityTransform` `.IdentityTransform`
"log" `LogScale` `LogTransform` `InvertedLogTransform`
"logit" `LogitScale` `LogitTransform` `LogisticTransform`
"symlog" `SymmetricalLogScale` `SymmetricalLogTransform` `InvertedSymmetricalLogTransform`
=====
=====
```

A user will often only use the scale name, e.g. when setting the scale through
``~.Axes.set_xscale`: ``ax.set_xscale("log")```.

See also the :ref:`scales` examples `<sphinx_glr_gallery_scales>` in the documentation.

Custom scaling can be achieved through ``FuncScale``, or by creating your own
``ScaleBase`` subclass and corresponding transforms (see :doc:`/gallery/scales/custom_scale`).
Third parties can register their scales by name through ``register_scale``.

matplotlib.scale.AsinhLocator

```
AsinhLocator(linear_width, numticks=11, symthresh=0.2, base=10, subs=None)
```

Place ticks spaced evenly on an inverse-sinh scale.

Generally used with the ``~.scale.AsinhScale`` class.

.. note::

This API is provisional and may be revised in the future
based on early user feedback.

matplotlib.scale.AsinhScale

```
AsinhScale(axis, *, linear_width=1.0, base=10, subs='auto', **kwargs)
```

A quasi-logarithmic scale based on the inverse hyperbolic sine (asinh)

For values close to zero, this is essentially a linear scale,
but for large magnitude values (either positive or negative)
it is asymptotically logarithmic. The transition between these

linear and logarithmic regimes is smooth, and has no discontinuities in the function gradient in contrast to the `.SymmetricalLogScale` ("symlog") scale.

Specifically, the transformation of an axis coordinate a is $a \rightarrow a_0 \sinh^{-1}(a / a_0)$ where a_0 is the effective width of the linear region of the transformation. In that region, the transformation is $a \rightarrow a + \mathcal{O}(a^3)$. For large values of a the transformation behaves as $a \rightarrow a_0 \ln |a| + \mathcal{O}(1)$.

.. note::

This API is provisional and may be revised in the future based on early user feedback.

matplotlib.scale.AsinhTransform

```
AsinhTransform(linear_width)
```

Inverse hyperbolic-sine transformation used by `.AsinhScale`

matplotlib.scale.AutoLocator

```
AutoLocator()
```

Place evenly spaced ticks, with the step size and maximum number of ticks chosen automatically.

This is a subclass of `~matplotlib.ticker.MaxNLocator`, with parameters `*nbins = 'auto'` and `*steps = [1, 2, 2.5, 5, 10]`.

matplotlib.scale.AutoMinorLocator

```
AutoMinorLocator(n=None)
```

Place evenly spaced minor ticks, with the step size and maximum number of ticks chosen automatically.

The Axis must use a linear scale and have evenly spaced major ticks.

matplotlib.scale.FuncScale

```
FuncScale(axis, functions)
```

Provide an arbitrary scale with user-supplied function for the axis.

matplotlib.scale.FuncScaleLog

```
FuncScaleLog(axis, functions, base=10)
```

Provide an arbitrary scale with user-supplied function for the axis and then put on a logarithmic axes.

matplotlib.scale.FuncTransform

```
FuncTransform(forward, inverse)
```

A simple transform that takes an arbitrary function for the forward and inverse transform.

matplotlib.scale.IdentityTransform

```
IdentityTransform(*args, **kwargs)
```

A special class that does one thing, the identity transform, in a fast way.

matplotlib.scale.InvertedAsinhTransform

```
InvertedAsinhTransform(linear_width)
```

Hyperbolic sine transformation used by `.AsinhScale`

matplotlib.scale.InvertedLogTransform

```
InvertedLogTransform(base)
```

The base class of all `TransformNode` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of `Affine2D`.

Subclasses of this class should override the following members (at minimum):

- :attr: `input_dims`
- :attr: `output_dims`
- :meth: `transform`
- :meth: `inverted` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr: `is_separable` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr: `has_inverse` (defaults to True if `:meth: inverted` is overridden, False otherwise)

If the transform needs to do something non-standard with `matplotlib.path.Path` objects, such as adding curves where there were once line segments, it should override:

- :meth: `transform_path`

matplotlib.scale.InvertedSymmetricalLogTransform

```
InvertedSymmetricalLogTransform(base, linthresh, linscale)
```

The base class of all `TransformNode`` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class.
New affine transformations should be subclasses of `Affine2D``.

Subclasses of this class should override the following members (at minimum):

- :attr:`input_dims`
- :attr:`output_dims`
- :meth:`transform`
- :meth:`inverted` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr:`is_separable` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr:`has_inverse` (defaults to True if :meth:`inverted` is overridden, False otherwise)

If the transform needs to do something non-standard with `matplotlib.path.Path`` objects, such as adding curves where there were once line segments, it should override:

- :meth:`transform_path`

matplotlib.scale.LinearScale

```
LinearScale(axis)
```

The default linear scale.

matplotlib.scale.LogFormatterSciNotation

```
LogFormatterSciNotation(base=10.0, labelOnlyBase=False, minor_thresholds=None, linthresh=None)
```

Format values following scientific notation in a logarithmic axis.

matplotlib.scale.LogLocator

```
LogLocator(base=10.0, subs=(1.0,), *, numticks=None)
```

Place logarithmically spaced ticks.

Places ticks at the values `subs[j] * base**i``.

matplotlib.scale.LogScale

```
LogScale(axis, *, base=10, subs=None, nonpositive='clip')
```

A standard logarithmic scale. Care is taken to only plot positive values.

matplotlib.scale.LogTransform

```
LogTransform(base, nonpositive='clip')
```

The base class of all `TransformNode`` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class.
New affine transformations should be subclasses of `Affine2D``.

Subclasses of this class should override the following members (at minimum):

- :attr:`input_dims`
- :attr:`output_dims`
- :meth:`transform`
- :meth:`inverted` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr:`is_separable` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr:`has_inverse` (defaults to True if :meth:`inverted` is overridden, False otherwise)

If the transform needs to do something non-standard with `matplotlib.path.Path`` objects, such as adding curves where there were once line segments, it should override:

- :meth:`transform_path`

matplotlib.scale.LogisticTransform

```
LogisticTransform(nonpositive='mask')
```

The base class of all `TransformNode`` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class.
New affine transformations should be subclasses of `Affine2D``.

Subclasses of this class should override the following members (at minimum):

- :attr:`input_dims`
- :attr:`output_dims`
- :meth:`transform`
- :meth:`inverted` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr:`is_separable` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr:`has_inverse` (defaults to True if :meth:`inverted` is overridden, False otherwise)

If the transform needs to do something non-standard with ``matplotlib.path.Path`` objects, such as adding curves where there were once line segments, it should override:

- :meth:`transform_path`

matplotlib.scale.LogitFormatter

```
LogitFormatter(*, use_overline=False, one_half='\\frac{1}{2}', minor=False,
minor_threshold=25, minor_number=6)
```

Probability formatter (using Math text).

matplotlib.scale.LogitLocator

```
LogitLocator(minor=False, *, nbins='auto')
```

Place ticks spaced evenly on a logit scale.

matplotlib.scale.LogitScale

```
LogitScale(axis, nonpositive='mask', *, one_half='\\frac{1}{2}', use_overline=False)
```

Logit scale for data between zero and one, both excluded.

This scale is similar to a log scale close to zero and to one, and almost linear around 0.5. It maps the interval $]0, 1[$ onto $]-\infty, +\infty[$.

matplotlib.scale.LogitTransform

```
LogitTransform(nonpositive='mask')
```

The base class of all ``TransformNode`` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class.
New affine transformations should be subclasses of ``Affine2D``.

Subclasses of this class should override the following members (at minimum):

- :attr:`input_dims`
- :attr:`output_dims`
- :meth:`transform`
- :meth:`inverted` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr:`is_separable` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr:`has_inverse` (defaults to True if :meth:`inverted` is overridden, False otherwise)

If the transform needs to do something non-standard with ``matplotlib.path.Path`` objects, such as adding curves

where there were once line segments, it should override:

```
- :meth:`transform_path`
```

matplotlib.scale.NullFormatter

```
NullFormatter()
```

Always return the empty string.

matplotlib.scale.NullLocator

```
NullLocator()
```

No ticks

matplotlib.scale.ScalarFormatter

```
ScalarFormatter(useOffset=None, useMathText=None, useLocale=None, *, usetex=None)
```

Format tick values as a number.

Parameters

useOffset : bool or float, default: :rc:`axes.formatter.useoffset`
Whether to use offset notation. See `.set_useOffset``.
useMathText : bool, default: :rc:`axes.formatter.use_mathtext`
Whether to use fancy math formatting. See `.set_useMathText``.
useLocale : bool, default: :rc:`axes.formatter.use_locale`
Whether to use locale settings for decimal sign and positive sign.
See `.set_useLocale``.
usetex : bool, default: :rc:`text.usetex`
To enable/disable the use of TeX's math mode for rendering the
numbers in the formatter.

.. versionadded:: 3.10

Notes

In addition to the parameters above, the formatting of scientific vs.
floating point representation can be configured via `.set_scientific``
and `.set_powerlimits``).

****Offset notation and scientific notation****

Offset notation and scientific notation look quite similar at first sight.
Both split some information from the formatted tick values and display it
at the end of the axis.

- The scientific notation splits up the order of magnitude, i.e. a
multiplicative scaling factor, e.g. ```1e6```.

- The offset notation separates an additive constant, e.g. ```+1e6```. The
offset notation label is always prefixed with a ```+``` or ```-``` sign

and is thus distinguishable from the order of magnitude label.

The following plot with x limits ``1_000_000`` to ``1_000_010`` illustrates the different formatting. Note the labels at the right edge of the x axis.

.. plot::

```
lim = (1_000_000, 1_000_010)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, gridspec_kw={'hspace': 2})
ax1.set(title='offset notation', xlim=lim)
ax2.set(title='scientific notation', xlim=lim)
ax2.xaxis.get_major_formatter().set_useOffset(False)
ax3.set(title='floating-point notation', xlim=lim)
ax3.xaxis.get_major_formatter().set_useOffset(False)
ax3.xaxis.get_major_formatter().set_scientific(False)
```

matplotlib.scale.ScaleBase

```
ScaleBase(axis)
```

The base class for all scales.

Scales are separable transformations, working on a single dimension.

Subclasses should override

:attr:`name`

The scale's name.

:meth:`get_transform`

A method returning a `Transform`, which converts data coordinates to scaled coordinates. This transform should be invertible, so that e.g. mouse positions can be converted back to data coordinates.

:meth:`set_default_locators_and_formatters`

A method that sets default locators and formatters for an `~.axis.Axis` that uses this scale.

:meth:`limit_range_for_scale`

An optional method that "fixes" the axis range to acceptable values, e.g. restricting log-scaled axes to positive values.

matplotlib.scale.SymmetricalLogLocator

```
SymmetricalLogLocator(transform=None, subs=None, linthresh=None, base=None)
```

Place ticks spaced linearly near zero and spaced logarithmically beyond a threshold.

matplotlib.scale.SymmetricalLogScale

```
SymmetricalLogScale(axis, *, base=10, linthresh=2, subs=None, linscale=1)
```

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a

need to have a range around zero that is linear. The parameter `*linthresh*` allows the user to specify the size of this range (`-*linthresh*, *linthresh*`).

See `:doc:`/gallery/scales/symlog_demo`` for a detailed description.

Parameters

`base` : float, default: 10
The base of the logarithm.

`linthresh` : float, default: 2
Defines the range ``(-x, x)``, within which the plot is linear.
This avoids having the plot go to infinity around zero.

`subs` : sequence of int
Where to place the subticks between each major tick.
For example, in a log10 scale: ``[2, 3, 4, 5, 6, 7, 8, 9]`` will place 8 logarithmically spaced minor ticks between each major tick.

`linscale` : float, optional
This allows the linear range ``(-linthresh, linthresh)`` to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when `*linscale* == 1.0` (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

matplotlib.scale.SymmetricalLogTransform

```
SymmetricalLogTransform(base, linthresh, linscale)
```

The base class of all ``TransformNode`` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class.
New affine transformations should be subclasses of ``Affine2D``.

Subclasses of this class should override the following members (at minimum):

- `:attr:`input_dims``
- `:attr:`output_dims``
- `:meth:`transform``
- `:meth:`inverted`` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- `:attr:`is_separable`` (defaults to True for 1D -> 1D transforms, False otherwise)
- `:attr:`has_inverse`` (defaults to True if `:meth:`inverted`` is overridden, False otherwise)

If the transform needs to do something non-standard with

``matplotlib.path.Path`` objects, such as adding curves where there were once line segments, it should override:

- :meth:`transform_path`

matplotlib.scale.Transform

```
Transform(shorthand_name=None)
```

The base class of all ``TransformNode`` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of ``Affine2D``.

Subclasses of this class should override the following members (at minimum):

- :attr:`input_dims`
- :attr:`output_dims`
- :meth:`transform`
- :meth:`inverted` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr:`is_separable` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr:`has_inverse` (defaults to True if :meth:`inverted` is overridden, False otherwise)

If the transform needs to do something non-standard with ``matplotlib.path.Path`` objects, such as adding curves where there were once line segments, it should override:

- :meth:`transform_path`

matplotlib.scale.get_scale_names

```
get_scale_names()
```

Return the names of the available scales.

matplotlib.scale.register_scale

```
register_scale(scale_class)
```

Register a new kind of scale.

Parameters

`scale_class` : subclass of ``ScaleBase``
The scale to register.

matplotlib.scale.scale_factory

```
scale_factory(scale, axis, **kwargs)
```

Return a scale class by name.

Parameters

scale : {'asinh', 'function', 'functionlog', 'linear', 'log', 'logit', 'symlog'}
axis : ~matplotlib.axis.Axis

matplotlib.set_loglevel

```
set_loglevel(level)
```

Configure Matplotlib's logging levels.

Matplotlib uses the standard library `logging` framework under the root logger 'matplotlib'. This is a helper function to:

- set Matplotlib's root logger level
- set the root logger handler's level, creating the handler if it does not exist yet

Typically, one should call ``set_loglevel("info")`` or ``set_loglevel("debug")`` to get additional debugging information.

Users or applications that are installing their own logging handlers may want to directly manipulate ``logging.getLogger('matplotlib')`` rather than use this function.

Parameters

level : {"notset", "debug", "info", "warning", "error", "critical"}
The log level of the handler.

Notes

The first time this function is called, an additional handler is attached to Matplotlib's root handler; this handler is reused every time and this function simply manipulates the logger and handler's level.

matplotlib.sphinxext

```
sphinxext(...)
```

No description available.

matplotlib.sphinxext.figmpl_directive

```
figmpl_directive(...)
```

Add a ``figure-mpl`` directive that is a responsive version of ``figure``.

This implementation is very similar to ``.. figure::``, except it also allows a ``srcset`` argument to be passed to the image tag, hence allowing responsive

resolution images.

There is no particular reason this could not be used standalone, but is meant to be used with `:doc:`/api/sphinxext_plot_directive_api``.

Note that the directory organization is a bit different than `.. figure::``. See the *FigureMpl* documentation below.

matplotlib.sphinxext.mathmpl

```
mathmpl(...)
```

A role and directive to display mathtext in Sphinx

=====

The ```mathmpl``` Sphinx extension creates a mathtext image in Matplotlib and shows it in html output. Thus, it is a true and faithful representation of what you will see if you pass a given LaTeX string to Matplotlib (see `:ref:`mathtext``).

.. warning::

In most cases, you will likely want to use one of `Sphinx's builtin Math extensions

`<https://www.sphinx-doc.org/en/master/usage/extensions/math.html>`__`

instead of this one. The builtin Sphinx math directive uses MathJax to render mathematical expressions, and addresses accessibility concerns that ```mathmpl``` doesn't address.

Mathtext may be included in two ways:

1. Inline, using the role::

This text uses inline math: `:mathmpl:`\alpha > \beta``.

which produces:

This text uses inline math: `:mathmpl:`\alpha > \beta``.

2. Standalone, using the directive::

Here is some standalone math:

.. mathmpl::

$\alpha > \beta$

which produces:

Here is some standalone math:

.. mathmpl::

$\alpha > \beta$

Options

The ```mathmpl``` role and directive both support the following options:

`fontset` : str, default: 'cm'

The font set to use when displaying math. See `:rc:`mathtext.fontset``.

`fontsize` : float

The font size, in points. Defaults to the value from the extension configuration option defined below.

Configuration options

The `mathtext` extension has the following configuration options:

`mathmpl_fontsize` : float, default: 10.0

Default font size, in points.

`mathmpl_srcset` : list of str, default: []

Additional image sizes to generate when embedding in HTML, to support `responsive resolution images`

<https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Responsive_images>`

—.

The list should contain additional x-descriptors (```1.5x```, ```2x```, etc.) to generate (1x is the default and always included.)

matplotlib.sphinxext.plot_directive

```
plot_directive(...)
```

A directive for including a Matplotlib plot in a Sphinx document

=====

This is a Sphinx extension providing a `reStructuredText` directive ```.. plot::``` for including a plot in a Sphinx document.

In HTML output, ```.. plot::``` will include a `.png` file with a link to a high-res `.png` and `.pdf`. In LaTeX output, it will include a `.pdf`.

The plot content may be defined in one of three ways:

1. **A path to a source file** as the argument to the directive::

`.. plot:: path/to/plot.py`

When a path to a source file is given, the content of the directive may optionally contain a caption for the plot::

`.. plot:: path/to/plot.py`

The plot caption.

Additionally, one may specify the name of a function to call (with no arguments) immediately after importing the module::

```
.. plot:: path/to/plot.py plot_function1
```

2. Included as **inline content** to the directive::

```
.. plot::
```

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [4, 5, 6])
plt.title("A plotting exammple")
```

3. Using **doctest** syntax::

```
.. plot::
```

A plotting example:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1, 2, 3], [4, 5, 6])
```

Options

The `.. plot::` directive supports the following options:

```:format:``` : {'python', 'doctest'}

The format of the input. If unset, the format is auto-detected.

```:include-source:``` : bool

Whether to display the source code. The default can be changed using the ```plot_include_source``` variable in `:file:`conf.py`` (which itself defaults to False).

```:show-source-link:``` : bool

Whether to show a link to the source in HTML. The default can be changed using the ```plot_html_show_source_link``` variable in `:file:`conf.py`` (which itself defaults to True).

```:context:``` : bool or str

If provided, the code will be run in the context of all previous plot directives for which the ```:context:``` option was specified. This only applies to inline code plot directives, not those run from files. If the ```:context: reset``` option is specified, the context is reset for this and future plots, and previous figures are closed prior to running the code. ```:context: close-figs``` keeps the context but closes previous figures before running the code.

```:nofigs:``` : bool

If specified, the code block will be run, but no figures will be inserted. This is usually useful with the ```:context:``` option.

```:caption:``` : str

If specified, the option's argument will be used as a caption for the figure. This overwrites the caption given in the content, when the plot

is generated from a file.

Additionally, this directive supports all the options of the ``image`` directive <https://docutils.sourceforge.io/docs/ref/rst/directives.html#image>, except for ```:target:``` (since plot will add its own target). These include ```:alt:```, ```:height:```, ```:width:```, ```:scale:```, ```:align:``` and ```:class:```.

Configuration options

The plot directive has the following configuration options:

`plot_include_source`

Default value for the include-source option (default: False).

`plot_html_show_source_link`

Whether to show a link to the source in HTML (default: True).

`plot_pre_code`

Code that should be executed before each plot. If None (the default), it will default to a string containing::

```
import numpy as np
from matplotlib import pyplot as plt
```

`plot_basedir`

Base directory, to which ```plot:``` file names are relative to. If None or empty (the default), file names are relative to the directory where the file containing the directive is.

`plot_formats`

File formats to generate (default: ['png', 'hires.png', 'pdf']). List of tuples or strings::

```
[(suffix, dpi), suffix, ...]
```

that determine the file format and the DPI. For entries whose DPI was omitted, sensible defaults are chosen. When passing from the command line through `sphinx_build` the list should be passed as `suffix:dpi,suffix:dpi, ...`

`plot_html_show_formats`

Whether to show links to the files in HTML (default: True).

`plot_rcparams`

A dictionary containing any non-standard rcParams that should be applied before each plot (default: {}).

`plot_apply_rcparams`

By default, rcParams are applied when ```:context:``` option is not used in a plot directive. If set, this configuration option overrides this behavior and applies rcParams before each plot.

`plot_working_directory`

By default, the working directory will be changed to the directory of

the example, so the code can get at its data files, if any. Also its path will be added to ``sys.path`` so it can import any helper modules sitting beside it. This configuration option can be used to specify a central directory (also added to ``sys.path``) where data files and helper modules for all code are located.

`plot_template`

Provide a customized template for preparing restructured text.

`plot_srcset`

Allow the `srcset` image option for responsive image resolutions. List of strings with the multiplicative factors followed by an "x".

e.g. `["2.0x", "1.5x"]`. "2.0x" will create a png with the default "png"

resolution from `plot_formats`, multiplied by 2. If `plot_srcset` is specified, the plot directive uses the

`:doc:`/api/sphinxext_figmpl_directive_api`` (instead of the usual figure directive) in the intermediary rst file that is generated.

The `plot_srcset` option is incompatible with `*singlehtml*` builds, and an error will be raised.

Notes on how it works

The plot directive runs the code it is given, either in the source file or the code under the directive. The figure created (if any) is saved in the sphinx build directory under a subdirectory named ```plot_directive```. It then creates an intermediate rst file that calls a ```.. figure:``` directive (or ```.. figmpl:``` directive if ```plot_srcset``` is being used) and has links to the ```*.png``` files in the ```plot_directive``` directory. These translations can be customized by changing the `*plot_template*`. See the source of `:doc:`/api/sphinxext_plot_directive_api`` for the templates defined in `*TEMPLATE*` and `*TEMPLATE_SRCSET*`.

matplotlib.sphinxext.roles

```
roles(...)
```

Custom roles for the Matplotlib documentation.

`.. warning::`

These roles are considered semi-public. They are only intended to be used in the Matplotlib documentation.

However, it can happen that downstream packages end up pulling these roles into their documentation, which will result in documentation build errors. The following describes the exact mechanism and how to fix the errors.

There are two ways, Matplotlib docstrings can end up in downstream documentation. You have to subclass a Matplotlib class and either use the ```:inherited-members:``` option in your autodoc configuration, or you have to override a method without specifying a new docstring; the new method will inherit the original docstring and still render in your autodoc. If the docstring contains one of the custom sphinx roles, you'll see one of the following error messages:

.. code-block:: none

Unknown interpreted text role "mpltype".

Unknown interpreted text role "rc".

To fix this, you can add this module as extension to your sphinx `:file:`conf.py`::`

```
extensions = [  
    'matplotlib.sphinxext.roles',  
    # Other extensions.  
]
```

.. warning::

Direct use of these roles in other packages is not officially supported. We reserve the right to modify or remove these roles without prior notification.

matplotlib.spines

```
spines(...)
```

No description available.

matplotlib.spines.Spine

```
Spine(axes, spine_type, path, **kwargs)
```

An axis spine -- the line noting the data area boundaries.

Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions. See `~.Spine.set_position`` for more information.

The default position is ```('outward', 0)```.

Spines are subclasses of ``Patch``, and inherit much of their behavior.

Spines draw a line, a circle, or an arc depending on if `~.Spine.set_patch_line``, `~.Spine.set_patch_circle``, or `~.Spine.set_patch_arc`` has been called. Line-like is the default.

For examples see `:ref:`spines_examples``.

matplotlib.spines.Spines

```
Spines(**kwargs)
```

The container of all ``Spine``s in an Axes.

The interface is dict-like mapping names (e.g. 'left') to ``Spine`` objects. Additionally, it implements some pandas.Series-like features like accessing elements by attribute::

```
spines['top'].set_visible(False)
spines.top.set_visible(False)
```

Multiple spines can be addressed simultaneously by passing a list::

```
spines[['top', 'right']].set_visible(False)
```

Use an open slice to address all spines::

```
spines[:].set_visible(False)
```

The latter two indexing methods will return a `SpinesProxy` that broadcasts all `set_*` and `set()` calls to its members, but cannot be used for any other operation.

matplotlib.spines.SpinesProxy

```
SpinesProxy(spine_dict)
```

A proxy to broadcast `set_*` and `set()` method calls to contained `.Spines`.

The proxy cannot be used for any other operations on its members.

The supported methods are determined dynamically based on the contained spines. If not all spines support a given method, it's executed only on the subset of spines that support it.

matplotlib.spines.allow_rasterization

```
allow_rasterization(draw)
```

Decorator for `Artist.draw` method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependent renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

matplotlib.stackplot

```
stackplot(...)
```

Stacked area plot for 1D arrays inspired by Douglas Y'barbo's [stackoverflow](https://stackoverflow.com/q/2225995/) answer:

<https://stackoverflow.com/q/2225995/>

(<https://stackoverflow.com/users/66549/doug>)

matplotlib.stackplot.stackplot

```
stackplot(axes, x, *args, labels=(), colors=None, hatch=None, baseline='zero',
**kwargs)
```

Draw a stacked area plot or a streamgraph.

Parameters

x : (N,) array-like

y : (M, N) array-like

The data can be either stacked or unstacked. Each of the following calls is legal::

stackplot(x, y) # where y has shape (M, N) e.g. y = [y1, y2, y3, y4]

stackplot(x, y1, y2, y3, y4) # where y1, y2, y3, y4 have length N

baseline : {'zero', 'sym', 'wiggle', 'weighted_wiggle'}

Method used to calculate the baseline:

- ``zero``: Constant zero baseline, i.e. a simple stacked plot.

- ``sym``: Symmetric around zero and is sometimes called 'ThemeRiver'.

- ``wiggle``: Minimizes the sum of the squared slopes.

- ``weighted_wiggle``: Does the same but weights to account for size of each layer. It is also called 'Streamgraph'-layout. More details can be found at <http://leebyron.com/streamgraph/>.

labels : list of str, optional

A sequence of labels to assign to each data series. If unspecified, then no labels will be applied to artists.

colors : list of :mpltype:`color`, optional

A sequence of colors to be cycled through and used to color the stacked areas. The sequence need not be exactly the same length as the number of provided *y*, in which case the colors will repeat from the beginning.

If not specified, the colors from the Axes property cycle will be used.

hatch : list of str, default: None

A sequence of hatching styles. See

:doc:`/gallery/shapes_and_collections/hatch_style_reference`.

The sequence will be cycled through for filling the stacked areas from bottom to top.

It need not be exactly the same length as the number of provided *y*, in which case the styles will repeat from the beginning.

.. versionadded:: 3.9

Support for list input

data : indexable object, optional

DATA_PARAMETER_PLACEHOLDER

**kwargs

All other keyword arguments are passed to ``.Axes.fill_between``.

Returns

list of ``.PolyCollection``

A list of ``.PolyCollection`` instances, one for each element in the

stacked area plot.

matplotlib.streamplot

```
streamplot(...)
```

Streamline plotting for 2D vector fields.

matplotlib.streamplot.DomainMap

```
DomainMap(grid, mask)
```

Map representing different coordinate systems.

Coordinate definitions:

- * axes-coordinates goes from 0 to 1 in the domain.
- * data-coordinates are specified by the input x-y coordinates.
- * grid-coordinates goes from 0 to N and 0 to M for an N x M grid, where N and M match the shape of the input data.
- * mask-coordinates goes from 0 to N and 0 to M for an N x M mask, where N and M are user-specified to control the density of streamlines.

This class also has methods for adding trajectories to the StreamMask. Before adding a trajectory, run ``start_trajectory`` to keep track of regions crossed by a given trajectory. Later, if you decide the trajectory is bad (e.g., if the trajectory is very short) just call ``undo_trajectory``.

matplotlib.streamplot.Grid

```
Grid(x, y)
```

Grid of data.

matplotlib.streamplot.InvalidIndexError

```
InvalidIndexError(...)
```

Common base class for all non-exit exceptions.

matplotlib.streamplot.OutOfBounds

```
OutOfBounds(...)
```

Sequence index out of range.

matplotlib.streamplot.StreamMask

```
StreamMask(density)
```

Mask to keep track of discrete regions crossed by streamlines.

The resolution of this grid determines the approximate spacing between trajectories. Streamlines are only allowed to pass through zeroed cells:

When a streamline enters a cell, that cell is set to 1, and no new streamlines are allowed to enter.

matplotlib.streamplot.StreamplotSet

```
StreamplotSet(lines, arrows)
```

No description available.

matplotlib.streamplot.TerminateTrajectory

```
TerminateTrajectory(...)
```

Common base class for all non-exit exceptions.

matplotlib.streamplot.interpgrid

```
interpgrid(a, xi, yi)
```

Fast 2D, linear interpolation on an integer grid

matplotlib.streamplot.streamplot

```
streamplot(axes, x, y, u, v, density=1, linewidth=None, color=None, cmap=None, norm=None, arrowsize=1, arrowstyle='->', minlength=0.1, transform=None, zorder=None, start_points=None, maxlength=4.0, integration_direction='both', broken_streamlines=True)
```

Draw streamlines of a vector flow.

Parameters

x, y : 1D/2D arrays

Evenly spaced strictly increasing arrays to make a grid. If 2D, all rows of *x* must be equal and all columns of *y* must be equal; i.e., they must be as if generated by `np.meshgrid(x_1d, y_1d)`.

u, v : 2D arrays

x and *y*-velocities. The number of rows and columns must match the length of *y* and *x*, respectively.

density : float or (float, float)

Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid. *density* linearly scales this grid.

Each cell in the grid can have, at most, one traversing streamline.

For different densities in each direction, use a tuple

(*density_x*, *density_y*).

linewidth : float or 2D array

The width of the streamlines. With a 2D array the line width can be varied across the grid. The array must have the same shape as *u* and *v*.

color : `mpltype:color` or 2D array

The streamline color. If given an array, its values are converted to colors using *cmap* and *norm*. The array must have the same shape as *u* and *v*.

cmap, norm

Data normalization and colormapping parameters for *color*; only used

if **color** is an array of floats. See `~.Axes.imshow`` for a detailed description.

arrowsize : float
Scaling factor for the arrow size.

arrowstyle : str
Arrow style specification.
See `~matplotlib.patches.FancyArrowPatch``.

minlength : float
Minimum length of streamline in axes coordinates.

start_points : (N, 2) array
Coordinates of starting points for the streamlines in data coordinates (the same coordinates as the **x** and **y** arrays).

zorder : float
The zorder of the streamlines and arrows.
Artists with lower zorder values are drawn first.

maxlength : float
Maximum length of streamline in axes coordinates.

integration_direction : {'forward', 'backward', 'both'}, default: 'both'
Integrate the streamline in forward, backward or both directions.

data : indexable object, optional
DATA_PARAMETER_PLACEHOLDER

broken_streamlines : boolean, default: True
If False, forces streamlines to continue until they leave the plot domain. If True, they may be terminated if they come too close to another streamline.

Returns

StreamplotSet

Container object with attributes

- ```lines```: `.LineCollection`` of streamlines

- ```arrows```: `.PatchCollection`` containing `.FancyArrowPatch`` objects representing the arrows half-way along streamlines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

matplotlib.style

```
style(...)
```

No description available.

matplotlib.style.context

```
context(style, after_reset=False)
```

Context manager for using style settings temporarily.

Parameters

style : str, dict, Path or list

A style specification. Valid options are:

str

- One of the style names in ``.style.available`` (a builtin style or a style installed in the user library path).

- A dotted name of the form "package.style_name"; in that case, "package" should be an importable Python package name, e.g. at ```/path/to/package/__init__.py```; the loaded style file is ```/path/to/package/style_name.mplstyle```. (Style files in subpackages are likewise supported.)

- The path or URL to a style file, which gets loaded by ``.rc_params_from_file``.

dict

A mapping of key/value pairs for ``.matplotlib.rcParams``.

Path

The path to a style file, which gets loaded by ``.rc_params_from_file``.

list

A list of style specifiers (str, Path or dict), which are applied from first to last in the list.

after_reset : bool

If True, apply style after resetting settings to their defaults; otherwise, apply style on top of the current settings.

matplotlib.style.core

```
core(...)
```

Core functions and attributes for the matplotlib style library:

```use```

Select style sheet to override the current matplotlib settings.

```context```

Context manager to use a style sheet temporarily.

```available```

List available style sheets.

```library```

A dictionary of style names and matplotlib settings.

matplotlib.style.reload_library

```
reload_library()
```

Reload the style library.

matplotlib.style.use

```
use(style)
```

Use Matplotlib style settings from a style specification.

The style name of 'default' is reserved for reverting back to the default style settings.

.. note::

This updates the ``rcParams`` with the settings from the style. ``rcParams`` not defined in the style are kept.

Parameters

style : str, dict, Path or list

A style specification. Valid options are:

str

- One of the style names in ``style.available`` (a builtin style or a style installed in the user library path).
- A dotted name of the form "package.style_name"; in that case, "package" should be an importable Python package name, e.g. at ``/path/to/package/__init__.py``; the loaded style file is ``/path/to/package/style_name.mplstyle``. (Style files in subpackages are likewise supported.)

- The path or URL to a style file, which gets loaded by ``rc_params_from_file``.

dict

A mapping of key/value pairs for ``matplotlib.rcParams``.

Path

The path to a style file, which gets loaded by ``rc_params_from_file``.

list

A list of style specifiers (str, Path or dict), which are applied from first to last in the list.

Notes

The following ``rcParams`` are not related to style and will be ignored if found in a style specification:

- backend
- backend_fallback
- date.epoch
- docstring.hardcopy
- figure.max_open_warning
- figure.raise_window
- interactive
- savefig.directory
- timezone
- tk.window_focus
- toolbar

- webagg.address
- webagg.open_in_browser
- webagg.port
- webagg.port_retries

matplotlib.table

```
table(...)
```

Tables drawing.

.. note::

The table implementation in Matplotlib is lightly maintained. For a more featureful table implementation, you may wish to try `blume` <<https://github.com/swfua/blume>>`_.

Use the factory function `~matplotlib.table.table` to create a ready-made table from texts. If you need more control, use the `.Table` class and its methods.

The table consists of a grid of cells, which are indexed by (row, column). The cell (0, 0) is positioned at the top left.

Thanks to John Gill for providing the class and table.

matplotlib.table.Artist

```
Artist()
```

Abstract base class for objects that render into a FigureCanvas.

Typically, all visible elements in a figure are subclasses of Artist.

matplotlib.table.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" ``[[xmin, ymin], [xmax, ymax]]``.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" ``(xmin, ymin, xmax, ymax)``

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" `(xmin, ymin, width, height)`.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting `ignore=True` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify `ignore` explicitly. If not, the default value of `ignore` can be changed at any time by code with access to your Bbox, for example using the method `~.Bbox.ignore`.

****Properties of the `null` bbox****

.. note::

The current behavior of `Bbox.null()` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[inf, inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[ -inf, -inf], [inf, inf]])
```

matplotlib.table.Cell

```
Cell(xy, width, height, *, edgecolor='k', facecolor='w', fill=True, text='',
     loc='right', fontproperties=None, visible_edges='closed')
```

A cell is a `.Rectangle` with some associated `.Text`.

As a user, you'll most likely not create cells yourself. Instead, you should use either the `~matplotlib.table.table` factory function or `.Table.add_cell`.

matplotlib.table.CustomCell

```
CustomCell(xy, width, height, *, edgecolor='k', facecolor='w', fill=True, text='',
           loc='right', fontproperties=None, visible_edges='closed')
```

A cell is a `.Rectangle` with some associated `.Text`.

As a user, you'll most likely not create cells yourself. Instead, you should use either the `~matplotlib.table.table` factory function or `.Table.add_cell`.

matplotlib.table.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- `*vertices*`: an (N, 2) float array of vertices
- `*codes*`: an N-length `numpy.uint8` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three `CURVE4` codes.

The code types are:

- `STOP` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- `MOVETO` : 1 vertex

Pick up the pen and move to the given vertex.

- `LINETO` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.table.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point `*xy*` and its `*width*` and `*height*`.

The rectangle extends from ``xy[0]`` to ``xy[0] + width`` in x-direction and from ``xy[1]`` to ``xy[1] + height`` in y-direction. ::

```
: +-----+
: |
: height |
: |
: (xy)--- width ----+
```

One may picture `*xy*` as the bottom left corner, but which corner `*xy*` is actually depends on the direction of the axis and the sign of `*width*` and `*height*`; e.g. `*xy*` would be the bottom right corner if the x-axis was inverted or if `*width*` was negative.

matplotlib.table.Table

```
Table(ax, loc=None, bbox=None, **kwargs)
```

A table of cells.

The table consists of a grid of cells, which are indexed by (row, column).

For a simple table, you'll have a full grid of cells with indices from (0, 0) to (num_rows-1, num_cols-1), in which the cell (0, 0) is positioned at the top left. However, you can also add cells with negative indices. You don't have to add a cell to every grid position, so you can create tables that have holes.

***Note*:** You'll usually not create an empty table from scratch. Instead use `~matplotlib.table.table`` to create a table from data.

matplotlib.table.Text

```
Text(x=0, y=0, text='', *, color=None, verticalalignment='baseline',
horizontalalignment='left', multialignment=None, fontproperties=None, rotation=None,
linespacing=None, rotation_mode=None, usetex=None, wrap=False,
transform_rotates_text=False, parse_math=None, antialiased=None, **kwargs)
```

Handle storing and drawing of text in window or data coordinates.

matplotlib.table.allow_rasterization

```
allow_rasterization(draw)
```

Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependent renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

matplotlib.table.table

```
table(ax, cellText=None, cellColours=None, cellLoc='right', colWidths=None,
rowLabels=None, rowColours=None, rowLoc='left', colLabels=None, colColours=None,
colLoc='center', loc='bottom', bbox=None, edges='closed', **kwargs)
```

Add a table to an `~.axes.Axes``.

At least one of `*cellText*` or `*cellColours*` must be specified. These parameters must be 2D lists, in which the outer lists define the rows and the inner list define the column values per row. Each row must have the same number of elements.

The table can optionally have row and column headers, which are configured using `*rowLabels*`, `*rowColours*`, `*rowLoc*` and `*colLabels*`, `*colColours*`, `*colLoc*` respectively.

For finer grained control over tables, use the `~.Table`` class and add it to the Axes with `~.Axes.add_table``.

Parameters

`cellText` : 2D list of str or pandas.DataFrame, optional
The texts to place into the table cells.

Note: Line breaks in the strings are currently not accounted for and will result in the text exceeding the cell boundaries.

cellColours : 2D list of :mpltype:`color`, optional
The background colors of the cells.

cellLoc : {'right', 'center', 'left'}
The alignment of the text within the cells.

colWidths : list of float, optional
The column widths in units of the axes. If not given, all columns will have a width of $*1 / \text{ncols}$.

rowLabels : list of str, optional
The text of the row header cells.

rowColours : list of :mpltype:`color`, optional
The colors of the row header cells.

rowLoc : {'left', 'center', 'right'}
The text alignment of the row header cells.

colLabels : list of str, optional
The text of the column header cells.

colColours : list of :mpltype:`color`, optional
The colors of the column header cells.

colLoc : {'center', 'left', 'right'}
The text alignment of the column header cells.

loc : str, default: 'bottom'
The position of the cell with respect to **ax**. This must be one of the `~.Table.codes``.

bbox : `~.Bbox`` or [xmin, ymin, width, height], optional
A bounding box to draw the table into. If this is not **None**, this overrides **loc**.

edges : {'closed', 'open', 'horizontal', 'vertical'} or substring of 'BRTL'
The cell edges to be drawn with a line. See also `~.Cell.visible_edges``.

Returns

`~matplotlib.table.Table``
The created table.

Other Parameters

****kwargs**
`~.Table`` properties.

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
alpha: float or None
animated: bool
clip_box: `~matplotlib.transforms.BboxBase` or None
clip_on: bool
clip_path: Patch or (Path, Transform) or None
figure: `~matplotlib.figure.Figure` or `~matplotlib.figure.SubFigure`
fontsize: float
gid: str
in_layout: bool
label: object
mouseover: bool
path_effects: list of `.AbstractPathEffect`
picker: None or bool or float or callable
rasterized: bool
sketch_params: (scale: float, length: float, randomness: float)
snap: bool or None
transform: `~matplotlib.transforms.Transform`
url: str
visible: bool
zorder: float

matplotlib.testing

```
testing(...)
```

Helper functions for testing.

matplotlib.testing.compare

```
compare(...)
```

Utilities for comparing image results.

matplotlib.testing.decorators

```
decorators(...)
```

No description available.

matplotlib.testing.exceptions

```
exceptions(...)
```

No description available.

matplotlib.testing.ipython_in_subprocess

```
ipython_in_subprocess(requested_backend_or_gui_framework, all_expected_backends)
```

No description available.

matplotlib.testing.is_ci_environment

```
is_ci_environment()
```

No description available.

matplotlib.testing.jpl_units

```
jpl_units(...)
```

A sample set of units for use with testing unit conversion of Matplotlib routines. These are used because they use very strict enforcement of unitized data which will test the entire spectrum of how unitized data might be used (it is not always meaningful to convert to a float without specific units given).

UnitDbI is essentially a unitized floating point number. It has a minimal set of supported units (enough for testing purposes). All of the mathematical operation are provided to fully test any behaviour that might occur with unitized data. Remember that unitized data has rules as to how it can be applied to one another (a value of distance cannot be added to a value of time). Thus we need to guard against any accidental "default" conversion that will strip away the meaning of the data and render it neutered.

Epoch is different than a UnitDbI of time. Time is something that can be measured where an Epoch is a specific moment in time. Epochs are typically referenced as an offset from some predetermined epoch.

A difference of two epochs is a Duration. The distinction between a Duration and a UnitDbI of time is made because an Epoch can have different frames (or units). In the case of our test Epoch class the two allowed frames are 'UTC' and 'ET' (Note that these are rough estimates provided for testing purposes and should not be used in production code where accuracy of time frames is desired). As such a Duration also has a frame of reference and therefore needs to be called out as different that a simple measurement of time since a delta-t in one frame may not be the same in another.

matplotlib.testing.set_font_settings_for_testing

```
set_font_settings_for_testing()
```

No description available.

matplotlib.testing.set_reproducibility_for_testing

```
set_reproducibility_for_testing()
```

No description available.

matplotlib.testing.setup

```
setup()
```

No description available.

matplotlib.testing.subprocess_run_for_testing

```
subprocess_run_for_testing(command, env=None, timeout=60, stdout=None, stderr=None,
                           check=False, text=True, capture_output=False)
```

Create and run a subprocess.

Thin wrapper around `subprocess.run`, intended for testing. Will mark `fork()` failures on Cygwin as expected failures: not a success, but not indicating a problem with the code either.

Parameters

`args` : list of str

`env` : dict[str, str]

`timeout` : float

`stdout`, `stderr`

`check` : bool

`text` : bool

Also called `universal_newlines` in subprocess. I chose this name since the main effect is returning bytes (`False`) vs. str (`True`), though it also tries to normalize newlines across platforms.

`capture_output` : bool

Set `stdout` and `stderr` to `subprocess.PIPE`

Returns

`proc` : `subprocess.Popen`

See Also

`subprocess.run`

Raises

`pytest.xfail`

If platform is Cygwin and subprocess reports a `fork()` failure.

matplotlib.testing.subprocess_run_helper

```
subprocess_run_helper(func, *args, timeout, extra_env=None)
```

Run a function in a sub-process.

Parameters

`func` : function

The function to be run. It must be in a module that is importable.

`*args` : str

Any additional command line arguments to be passed in the first argument to `subprocess.run`.

`extra_env` : dict[str, str]

Any additional environment variables to be set for the subprocess.

matplotlib.testing.widgets

```
widgets(...)
```

```
=====
Widget testing utilities
=====
```

See also :mod:`matplotlib.tests.test_widgets`.

matplotlib.texmanager

```
texmanager(...)
```

Support for embedded TeX expressions in Matplotlib.

Requirements:

- * LaTeX.
- * `\Agg` backends: dvipng \geq 1.6.
- * PS backend: PSfrag, dvips, and Ghostscript \geq 9.0.
- * PDF and SVG backends: if LuaTeX is present, it will be used to speed up some post-processing steps, but note that it is not used to parse the TeX string itself (only LaTeX is supported).

To enable TeX rendering of all text in your Matplotlib figure, set :rc:`text.usetex` to True.

TeX and dvipng/dvips processing results are cached in `~/.matplotlib/tex.cache` for reuse between sessions.

`TexManager.get_rgba`` can also be used to directly obtain raster output as RGBA NumPy arrays.

matplotlib.texmanager.TextManager

```
TextManager()
```

Convert strings to dvi files using TeX, caching the results to a directory.

The cache directory is called `tex.cache` and is located in the directory returned by `.get_cachedir``.

Repeated calls to this constructor always return the same instance.

matplotlib.text

```
text(...)
```

Classes for including text in a figure.

matplotlib.text.Affine2D

```
Affine2D(matrix=None, **kwargs)
```

A mutable 2D affine transformation.

matplotlib.text.Annotation

```
Annotation(text, xy, xytext=None, xycoords='data', textcoords=None, arrowprops=None,
annotation_clip=None, **kwargs)
```

An `.Annotation` is a `.Text` that can refer to a specific position `*xy*`.
Optionally an arrow pointing from the text to `*xy*` can be drawn.

Attributes

`xy`

The annotated position.

`xycoords`

The coordinate system for `*xy*`.

`arrow_patch`

A `.FancyArrowPatch` to point from `*xytext*` to `*xy*`.

matplotlib.text.Artist

```
Artist()
```

Abstract base class for objects that render into a `FigureCanvas`.

Typically, all visible elements in a figure are subclasses of `Artist`.

matplotlib.text.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" `[[xmin, ymin], [xmax, ymax]]`.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a `Bbox` can be created from the flattened points array, the so-called "extents" `(xmin, ymin, xmax, ymax)`

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" `(xmin, ymin, width, height)`.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating `Bboxes` is the null `bbox`, which is a

stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting ``ignore=True`` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify ``ignore`` explicitly. If not, the default value of ``ignore`` can be changed at any time by code with access to your Bbox, for example using the method `~.Bbox.ignore`.

****Properties of the ``null`` bbox****

.. note::

The current behavior of `Bbox.null()` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[ -inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[ -inf, -inf], [inf, inf]])
```

matplotlib.text.BboxBase

```
BboxBase(shorthand_name=None)
```

The base class of all bounding boxes.

This class is immutable; `Bbox` is a mutable subclass.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

matplotlib.text.BboxTransformTo

```
BboxTransformTo(boxout, **kwargs)
```

`BboxTransformTo` is a transformation that linearly transforms points from the unit bounding box to a given `Bbox`.

matplotlib.text.FancyArrowPatch

```
FancyArrowPatch(posA=None, posB=None, *, path=None, arrowstyle='simple',  
connectionstyle='arc3', patchA=None, patchB=None, shrinkA=2, shrinkB=2,  
mutation_scale=1, mutation_aspect=1, **kwargs)
```

A fancy arrow patch.

It draws an arrow using the `ArrowStyle`. It is primarily used by the `~.axes.Axes.annotate` method. For most purposes, use the `annotate` method for drawing arrows.

The head and tail positions are fixed at the specified start and end points of the arrow, but the size and shape (in display coordinates) of the arrow does not change when the axis is moved or zoomed.

matplotlib.text.FancyBboxPatch

```
FancyBboxPatch(xy, width, height, boxstyle='round', *, mutation_scale=1,  
mutation_aspect=1, **kwargs)
```

A fancy box around a rectangle with lower left at `*xy* = (*x*, *y*)` with specified width and height.

`.FancyBboxPatch` is similar to `.Rectangle`, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the style classes defined in `.BoxStyle`.

matplotlib.text.FontProperties

```
FontProperties(family=None, style=None, variant=None, weight=None, stretch=None,  
size=None, fname=None, math_fontfamily=None)
```

A class for storing and manipulating font properties.

The font properties are the six properties described in the `W3C Cascading Style Sheet, Level 1`
<<http://www.w3.org/TR/1998/REC-CSS2-19980512/>>_ font

specification and `*math_fontfamily*` for math fonts:

- family: A list of font names in decreasing order of priority. The items may include a generic font family name, either 'sans-serif', 'serif', 'cursive', 'fantasy', or 'monospace'. In that case, the actual font to be used will be looked up from the associated rcParam during the search process in ``font.findfont``. Default: `:rc:`font.family``
- style: Either 'normal', 'italic' or 'oblique'. Default: `:rc:`font.style``
- variant: Either 'normal' or 'small-caps'. Default: `:rc:`font.variant``
- stretch: A numeric value in the range 0-1000 or one of 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded'. Default: `:rc:`font.stretch``
- weight: A numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'. Default: `:rc:`font.weight``
- size: Either a relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 10. Default: `:rc:`font.size``
- math_fontfamily: The family of fonts used to render math text. Supported values are: 'dejavusans', 'dejavuserif', 'cm', 'stix', 'stixsans' and 'custom'. Default: `:rc:`mathtext.fontset``

Alternatively, a font may be specified using the absolute path to a font file, by using the `*fname*` kwarg. However, in this case, it is typically simpler to just pass the path (as a ``pathlib.Path``, not a ``str``) to the `*font*` kwarg of the ``Text`` object.

The preferred usage of font sizes is to use the relative values, e.g., 'large', instead of absolute font sizes, e.g., 12. This approach allows all text sizes to be made larger or smaller based on the font manager's default font size.

This class accepts a single positional string as `fontconfig_pattern_`, or alternatively individual properties as keyword arguments::

```
FontProperties(pattern)
FontProperties(*, family=None, style=None, variant=None, ...)
```

This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

```
.. _fontconfig: https://www.freedesktop.org/wiki/Software/fontconfig/
.. _pattern: https://www.freedesktop.org/software/fontconfig/fontconfig-user.html
```

Note that Matplotlib's internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in Matplotlib than in other applications that use fontconfig.

matplotlib.text.IdentityTransform

```
IdentityTransform(*args, **kwargs)
```

A special class that does one thing, the identity transform, in a fast way.

matplotlib.text.OffsetFrom

```
OffsetFrom(artist, ref_coord, unit='points')
```

Callable helper class for working with `Annotation`.

matplotlib.text.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point *xy* and its *width* and *height*.

The rectangle extends from `xy[0]` to `xy[0] + width` in x-direction and from `xy[1]` to `xy[1] + height` in y-direction. ::

```
: +-----+
: ||
: height |
: ||
: (xy)--- width ----+
```

One may picture *xy* as the bottom left corner, but which corner *xy* is actually depends on the direction of the axis and the sign of *width* and *height*; e.g. *xy* would be the bottom right corner if the x-axis was inverted or if *width* was negative.

matplotlib.text.Text

```
Text(x=0, y=0, text='', *, color=None, verticalalignment='baseline',
horizontalalignment='left', multialignment=None, fontproperties=None, rotation=None,
linespacing=None, rotation_mode=None, usetex=None, wrap=False,
transform_rotates_text=False, parse_math=None, antialiased=None, **kwargs)
```

Handle storing and drawing of text in window or data coordinates.

matplotlib.text.TextPath

```
TextPath(xy, s, size=None, prop=None, _interpolation_steps=1, usetex=False)
```

Create a path from the text.

matplotlib.text.TextToPath

```
TextToPath()
```

A class that converts strings to paths.

matplotlib.text.Transform

```
Transform(shorthand_name=None)
```

The base class of all `TransformNode` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class.
New affine transformations should be subclasses of `Affine2D`.

Subclasses of this class should override the following members (at minimum):

- :attr: `input_dims`
- :attr: `output_dims`
- :meth: `transform`
- :meth: `inverted` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr: `is_separable` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr: `has_inverse` (defaults to True if :meth: `inverted` is overridden, False otherwise)

If the transform needs to do something non-standard with `matplotlib.path.Path` objects, such as adding curves where there were once line segments, it should override:

- :meth: `transform_path`

matplotlib.ticker

```
ticker(...)
```

Tick locating and formatting

=====

This module contains classes for configuring tick locating and formatting. Generic tick locators and formatters are provided, as well as domain specific custom ones.

Although the locators know nothing about major or minor ticks, they are used by the `Axis` class to support major and minor tick locating and formatting.

```
.. _tick_locating:  
.. _locators:
```

Tick locating

The `Locator` class is the base class for all tick locators. The locators

handle autoscaling of the view limits based on the data limits, and the choosing of tick locations. A useful semi-automatic tick locator is ``MultipleLocator``. It is initialized with a base, e.g., 10, and it picks axis limits and ticks that are multiples of that base.

The Locator subclasses defined here are:

```
=====
`AutoLocator` `MaxNLocator` with simple defaults. This is the default
tick locator for most plotting.
`MaxNLocator` Finds up to a max number of intervals with ticks at
nice locations.
`LinearLocator` Space ticks evenly from min to max.
`LogLocator` Space ticks logarithmically from min to max.
`MultipleLocator` Ticks and range are a multiple of base; either integer
or float.
`FixedLocator` Tick locations are fixed.
`IndexLocator` Locator for index plots (e.g., where
``x = range(len(y))``).
`NullLocator` No ticks.
`SymmetricalLogLocator` Locator for use with the symlog norm; works like
`LogLocator` for the part outside of the threshold and
adds 0 if inside the limits.
`AsinhLocator` Locator for use with the asinh norm, attempting to
space ticks approximately uniformly.
`LogitLocator` Locator for logit scaling.
`AutoMinorLocator` Locator for minor ticks when the axis is linear and the
major ticks are uniformly spaced. Subdivides the major
tick interval into a specified number of minor
intervals, defaulting to 4 or 5 depending on the major
interval.
=====
```

There are a number of locators specialized for date locations - see the `:mod:`.dates`` module.

You can define your own locator by deriving from `Locator`. You must override the `__call__` method, which returns a sequence of locations, and you will probably want to override the `autoscale` method to set the view limits from the data limits.

If you want to override the default locator, use one of the above or a custom locator and pass it to the x- or y-axis instance. The relevant methods are::

```
ax.xaxis.set_major_locator(xmajor_locator)
ax.xaxis.set_minor_locator(xminor_locator)
ax.yaxis.set_major_locator(ymajor_locator)
ax.yaxis.set_minor_locator(yminor_locator)
```

The default minor locator is ``NullLocator``, i.e., no minor ticks on by default.

.. note::

``Locator`` instances should not be used with more than one ``~matplotlib.axis.Axis`` or ``~matplotlib.axes.Axes``. So instead of::

```
locator = MultipleLocator(5)
ax.xaxis.set_major_locator(locator)
ax2.xaxis.set_major_locator(locator)
```

do the following instead::

```
ax.xaxis.set_major_locator(MultipleLocator(5))
ax2.xaxis.set_major_locator(MultipleLocator(5))
```

.. _formatters:

Tick formatting

Tick formatting is controlled by classes derived from `Formatter`. The formatter operates on a single tick value and returns a string to the axis.

```
=====
`NullFormatter` No labels on the ticks.
`FixedFormatter` Set the strings manually for the labels.
`FuncFormatter` User defined function sets the labels.
`StrMethodFormatter` Use string `format` method.
`FormatStrFormatter` Use an old-style sprintf format string.
`ScalarFormatter` Default formatter for scalars: autopick the format
string.
`LogFormatter` Formatter for log axes.
`LogFormatterExponent` Format values for log axis using
`exponent = log_base(value)`.
`LogFormatterMathtext` Format values for log axis using
`exponent = log_base(value)` using Math text.
`LogFormatterSciNotation` Format values for log axis using scientific notation.
`LogitFormatter` Probability formatter.
`EngFormatter` Format labels in engineering notation.
`PercentFormatter` Format labels as a percentage.
=====
```

You can derive your own formatter from the `Formatter` base class by simply overriding the `__call__` method. The formatter class has access to the axis view and data limits.

To control the major and minor tick label formats, use one of the following methods::

```
ax.xaxis.set_major_formatter(xmajor_formatter)
ax.xaxis.set_minor_formatter(xminor_formatter)
ax.yaxis.set_major_formatter(ymajor_formatter)
ax.yaxis.set_minor_formatter(yminor_formatter)
```

In addition to a `.Formatter` instance, `~.Axis.set_major_formatter` and `~.Axis.set_minor_formatter` also accept a `str` or function. `str` input will be internally replaced with an autogenerated `.StrMethodFormatter` with the input `str`. For function input, a `.FuncFormatter` with the input function will be generated and used.

See `:doc:/gallery/ticks/major_minor_demo` for an example of setting major

and minor ticks. See the `:mod:`matplotlib.dates`` module for more information and examples of using date locators and formatters.

`matplotlib.ticker.AsinhLocator`

```
AsinhLocator(linear_width, numticks=11, symthresh=0.2, base=10, subs=None)
```

Place ticks spaced evenly on an inverse-sinh scale.

Generally used with the `~.scale.AsinhScale`` class.

.. note::

This API is provisional and may be revised in the future based on early user feedback.

`matplotlib.ticker.AutoLocator`

```
AutoLocator()
```

Place evenly spaced ticks, with the step size and maximum number of ticks chosen automatically.

This is a subclass of `~matplotlib.ticker.MaxNLocator``, with parameters `*nbins = 'auto'*` and `*steps = [1, 2, 2.5, 5, 10]*`.

`matplotlib.ticker.AutoMinorLocator`

```
AutoMinorLocator(n=None)
```

Place evenly spaced minor ticks, with the step size and maximum number of ticks chosen automatically.

The Axis must use a linear scale and have evenly spaced major ticks.

`matplotlib.ticker.EngFormatter`

```
EngFormatter(unit='', places=None, sep=' ', *, usetex=None, useMathText=None, useOffset=False)
```

Format axis values using engineering prefixes to represent powers of 1000, plus a specified unit, e.g., 10 MHz instead of 1e7.

`matplotlib.ticker.FixedFormatter`

```
FixedFormatter(seq)
```

Return fixed strings for tick labels based only on position, not value.

.. note::

`~.FixedFormatter`` should only be used together with `~.FixedLocator``. Otherwise, the labels may end up in unexpected positions.

matplotlib.ticker.FixedLocator

```
FixedLocator(locs, nbins=None)
```

Place ticks at a set of fixed values.

If **nbins** is *None* ticks are placed at all values. Otherwise, the **locs** array of possible positions will be subsampled to keep the number of ticks $\leq \text{nbins} + 1$. The subsampling will be done to include the smallest absolute value; for example, if zero is included in the array of possibilities, then it will be included in the chosen ticks.

matplotlib.ticker.FormatStrFormatter

```
FormatStrFormatter(fmt)
```

Use an old-style ('%' operator) format string to format the tick.

The format string should have a single variable format (%) in it. It will be applied to the value (not the position) of the tick.

Negative numeric values (e.g., -1) will use a dash, not a Unicode minus; use `mathtext` to get a Unicode minus by wrapping the format specifier with \$ (e.g. "\$%g\$").

matplotlib.ticker.Formatter

```
Formatter()
```

Create a string based on a tick value and location.

matplotlib.ticker.FuncFormatter

```
FuncFormatter(func)
```

Use a user-defined function for formatting.

The function should take in two inputs (a tick value ```x``` and a position ```pos```), and return a string containing the corresponding tick label.

matplotlib.ticker.IndexLocator

```
IndexLocator(base, offset)
```

Place ticks at every *n*th point plotted.

`IndexLocator` assumes index plotting; i.e., that the ticks are placed at integer values in the range between 0 and `len(data)` inclusive.

matplotlib.ticker.LinearLocator

```
LinearLocator(numticks=None, presets=None)
```

Place ticks at evenly spaced values.

The first time this function is called it will try to set the number of ticks to make a nice tick partitioning. Thereafter, the number of ticks will be fixed so that interactive navigation will be nice

matplotlib.ticker.Locator

```
Locator()
```

Determine tick locations.

Note that the same locator should not be used across multiple `~matplotlib.axis.Axis`` because the locator stores references to the Axis data and view limits.

matplotlib.ticker.LogFormatter

```
LogFormatter(base=10.0, labelOnlyBase=False, minor_thresholds=None, linthresh=None)
```

Base class for formatting ticks on a log or symlog scale.

It may be instantiated directly, or subclassed.

Parameters

base : float, default: 10.

Base of the logarithm used in all calculations.

labelOnlyBase : bool, default: False

If True, label ticks only at integer powers of base.

This is normally True for major ticks and False for minor ticks.

minor_thresholds : (subset, all), default: (1, 0.4)

If labelOnlyBase is False, these two numbers control the labeling of ticks that are not at integer powers of base; normally these are the minor ticks. The controlling parameter is the log of the axis data range. In the typical case where base is 10 it is the number of decades spanned by the axis, so we can call it 'numdec'. If ```numdec <= all```, all minor ticks will be labeled. If ```all < numdec <= subset```, then only a subset of minor ticks will be labeled, so as to avoid crowding. If ```numdec > subset``` then no minor ticks will be labeled.

linthresh : None or float, default: None

If a symmetric log scale is in use, its ```linthresh``` parameter must be supplied here.

Notes

The ``set_locs`` method must be called to enable the subsetting logic controlled by the ```minor_thresholds``` parameter.

In some cases such as the colorbar, there is no distinction between major and minor ticks; the tick locations might be set manually, or by a locator that puts ticks at integer powers of base and at intermediate locations. For this situation, disable the `minor_thresholds` logic by using ```minor_thresholds=(np.inf, np.inf)```, so that all ticks will be labeled.

To disable labeling of minor ticks when 'labelOnlyBase' is False, use ```minor_thresholds=(0, 0)```. This is the default for the "classic" style.

Examples

To label a subset of minor ticks when the view limits span up to 2 decades, and all of the ticks when zoomed in to 0.5 decades or less, use ```minor_thresholds=(2, 0.5)```.

To label all minor ticks when the view limits span up to 1.5 decades, use ```minor_thresholds=(1.5, 1.5)```.

matplotlib.ticker.LogFormatterExponent

```
LogFormatterExponent(base=10.0, labelOnlyBase=False, minor_thresholds=None,
linthresh=None)
```

Format values for log axis using ```exponent = log_base(value)```.

matplotlib.ticker.LogFormatterMathtext

```
LogFormatterMathtext(base=10.0, labelOnlyBase=False, minor_thresholds=None,
linthresh=None)
```

Format values for log axis using ```exponent = log_base(value)```.

matplotlib.ticker.LogFormatterSciNotation

```
LogFormatterSciNotation(base=10.0, labelOnlyBase=False, minor_thresholds=None,
linthresh=None)
```

Format values following scientific notation in a logarithmic axis.

matplotlib.ticker.LogLocator

```
LogLocator(base=10.0, subs=(1.0,), *, numticks=None)
```

Place logarithmically spaced ticks.

Places ticks at the values ```subs[j] * base**i```.

matplotlib.ticker.LogitFormatter

```
LogitFormatter(*, use_overline=False, one_half='\\frac{1}{2}', minor=False,
minor_threshold=25, minor_number=6)
```

Probability formatter (using Math text).

matplotlib.ticker.LogitLocator

```
LogitLocator(minor=False, *, nbins='auto')
```

Place ticks spaced evenly on a logit scale.

matplotlib.ticker.MaxNLocator

```
MaxNLocator(nbins=None, **kwargs)
```

Place evenly spaced ticks, with a cap on the total number of ticks.

Finds nice tick locations with no more than $\text{nbins} + 1$ ticks being within the view limits. Locations beyond the limits are added to support autoscaling.

matplotlib.ticker.MultipleLocator

```
MultipleLocator(base=1.0, offset=0.0)
```

Place ticks at every integer multiple of a base plus an offset.

matplotlib.ticker.NullFormatter

```
NullFormatter()
```

Always return the empty string.

matplotlib.ticker.NullLocator

```
NullLocator()
```

No ticks

matplotlib.ticker.PercentFormatter

```
PercentFormatter(xmax=100, decimals=None, symbol='%', is_latex=False)
```

Format numbers as a percentage.

Parameters

xmax : float

Determines how the number is converted into a percentage.

xmax is the data value that corresponds to 100%.

Percentages are computed as $x / \text{xmax} * 100$. So if the data is already scaled to be percentages, **xmax** will be 100. Another common situation is where **xmax** is 1.0.

decimals : None or int

The number of decimal places to place after the point.

If **None** (the default), the number will be computed automatically.

symbol : str or None

A string that will be appended to the label. It may be

None or empty to indicate that no symbol should be used. LaTeX

special characters are escaped in `*symbol*` whenever latex mode is enabled, unless `*is_latex*` is `*True*`.

`is_latex` : bool

If `*False*`, reserved LaTeX characters in `*symbol*` will be escaped.

matplotlib.ticker.ScalarFormatter

```
ScalarFormatter(useOffset=None, useMathText=None, useLocale=None, *, usetex=None)
```

Format tick values as a number.

Parameters

`useOffset` : bool or float, default: `:rc:`axes.formatter.useoffset``

Whether to use offset notation. See ``set_useOffset``.

`useMathText` : bool, default: `:rc:`axes.formatter.use_mathtext``

Whether to use fancy math formatting. See ``set_useMathText``.

`useLocale` : bool, default: `:rc:`axes.formatter.use_locale``.

Whether to use locale settings for decimal sign and positive sign. See ``set_useLocale``.

`usetex` : bool, default: `:rc:`text.usetex``

To enable/disable the use of TeX's math mode for rendering the numbers in the formatter.

.. versionadded:: 3.10

Notes

In addition to the parameters above, the formatting of scientific vs. floating point representation can be configured via ``set_scientific`` and ``set_powerlimits``).

****Offset notation and scientific notation****

Offset notation and scientific notation look quite similar at first sight. Both split some information from the formatted tick values and display it at the end of the axis.

- The scientific notation splits up the order of magnitude, i.e. a multiplicative scaling factor, e.g. ``1e6``.

- The offset notation separates an additive constant, e.g. ``+1e6``. The offset notation label is always prefixed with a ``+`` or ``-`` sign and is thus distinguishable from the order of magnitude label.

The following plot with x limits ``1_000_000`` to ``1_000_010`` illustrates the different formatting. Note the labels at the right edge of the x axis.

.. plot::

```
lim = (1_000_000, 1_000_010)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, gridspec_kw={'hspace': 2})
```

```
ax1.set(title='offset notation', xlim=lim)
ax2.set(title='scientific notation', xlim=lim)
ax2.xaxis.get_major_formatter().set_useOffset(False)
ax3.set(title='floating-point notation', xlim=lim)
ax3.xaxis.get_major_formatter().set_useOffset(False)
ax3.xaxis.get_major_formatter().set_scientific(False)
```

matplotlib.ticker.StrMethodFormatter

```
StrMethodFormatter(fmt)
```

Use a new-style format string (as used by ``str.format``) to format the tick.

The field used for the tick value must be labeled `*x*` and the field used for the tick position must be labeled `*pos*`.

The formatter will respect `:rc:`axes.unicode_minus`` when formatting negative numeric values.

It is typically unnecessary to explicitly construct ``StrMethodFormatter`` objects, as ``~.Axis.set_major_formatter`` directly accepts the format string itself.

matplotlib.ticker.SymmetricalLogLocator

```
SymmetricalLogLocator(transform=None, subs=None, linthresh=None, base=None)
```

Place ticks spaced linearly near zero and spaced logarithmically beyond a threshold.

matplotlib.ticker.TickHelper

```
TickHelper()
```

No description available.

matplotlib.ticker.scale_range

```
scale_range(vmin, vmax, n=1, threshold=100)
```

No description available.

matplotlib.transforms

```
transforms(...)
```

Matplotlib includes a framework for arbitrary geometric transformations that is used to determine the final position of all elements drawn on the canvas.

Transforms are composed into trees of ``TransformNode`` objects whose actual value depends on their children. When the contents of children change, their parents are automatically invalidated. The next time an invalidated transform is accessed, it is recomputed to reflect those changes. This invalidation/caching approach prevents unnecessary recomputations of transforms, and contributes to better

interactive performance.

For example, here is a graph of the transform tree used to plot data to the figure:

.. graphviz:: /api/transforms.dot

:alt: Diagram of transform tree from data to figure coordinates.

The framework can be used for both affine and non-affine transformations. However, for speed, we want to use the backend renderers to perform affine transformations whenever possible. Therefore, it is possible to perform just the affine or non-affine part of a transformation on a set of data. The affine is always assumed to occur after the non-affine. For any transform::

full transform == non-affine part + affine part

The backends are not expected to handle non-affine transformations themselves.

See the tutorial :ref:`transforms_tutorial` for examples of how to use transforms.

matplotlib.transforms.Affine2D

```
Affine2D(matrix=None, **kwargs)
```

A mutable 2D affine transformation.

matplotlib.transforms.Affine2DBase

```
Affine2DBase(*args, **kwargs)
```

The base class of all 2D affine transformations.

2D affine transformations are performed using a 3x3 numpy array::

```
a c e
b d f
0 0 1
```

This class provides the read-only interface. For a mutable 2D affine transformation, use `Affine2D`.

Subclasses of this class will generally only need to override a constructor and `~.Transform.get_matrix` that generates a custom 3x3 matrix.

matplotlib.transforms.AffineBase

```
AffineBase(*args, **kwargs)
```

The base class of all affine transformations of any number of dimensions.

matplotlib.transforms.AffineDeltaTransform

```
AffineDeltaTransform(transform, **kwargs)
```

A transform wrapper for transforming displacements between pairs of points.

This class is intended to be used to transform displacements ("position deltas") between pairs of points (e.g., as the `offset_transform` of `.Collection`'s): given a transform `t` such that `t = AffineDeltaTransform(t) + offset`, `AffineDeltaTransform` satisfies `AffineDeltaTransform(a - b) == AffineDeltaTransform(a) - AffineDeltaTransform(b)`.

This is implemented by forcing the offset components of the transform matrix to zero.

This class is experimental as of 3.3, and the API may change.

matplotlib.transforms.Bbox

```
Bbox(points, **kwargs)
```

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" `[[xmin, ymin], [xmax, ymax]]`.

```
>>> Bbox([[1, 1], [3, 7]])  
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" `(xmin, ymin, xmax, ymax)`

```
>>> Bbox.from_extents(1, 1, 3, 7)  
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" `(xmin, ymin, width, height)`.

```
>>> Bbox.from_bounds(1, 1, 2, 6)  
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()  
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()  
>>> box.update_from_data_xy([[1, 1]])  
>>> box
```

```
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting ``ignore=True`` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify ``ignore`` explicitly. If not, the default value of ``ignore`` can be changed at any time by code with access to your Bbox, for example using the method `~.Bbox.ignore`.

****Properties of the ``null`` bbox****

.. note::

The current behavior of `Bbox.null()` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[-inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[-inf, -inf], [inf, inf]])
```

matplotlib.transforms.BboxBase

```
BboxBase(shorthand_name=None)
```

The base class of all bounding boxes.

This class is immutable; `Bbox` is a mutable subclass.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

matplotlib.transforms.BboxTransform

```
BboxTransform(boxin, boxout, **kwargs)
```

``BboxTransform`` linearly transforms points from one ``Bbox`` to another.

matplotlib.transforms.BboxTransformFrom

```
BboxTransformFrom(boxin, **kwargs)
```

``BboxTransformFrom`` linearly transforms points from a given ``Bbox`` to the unit bounding box.

matplotlib.transforms.BboxTransformTo

```
BboxTransformTo(boxout, **kwargs)
```

``BboxTransformTo`` is a transformation that linearly transforms points from the unit bounding box to a given ``Bbox``.

matplotlib.transforms.BboxTransformToMaxOnly

```
BboxTransformToMaxOnly(boxout, **kwargs)
```

[*Deprecated*] ``BboxTransformToMaxOnly`` is a transformation that linearly transforms points from the unit bounding box to a given ``Bbox`` with a fixed upper left of (0, 0).

Notes

.. deprecated:: 3.9

matplotlib.transforms.BlendedAffine2D

```
BlendedAffine2D(x_transform, y_transform, **kwargs)
```

A "blended" transform uses one transform for the `*x*`-direction, and another transform for the `*y*`-direction.

This version is an optimization for the case where both child transforms are of type ``Affine2DBase``.

matplotlib.transforms.BlendedGenericTransform

```
BlendedGenericTransform(x_transform, y_transform, **kwargs)
```

A "blended" transform uses one transform for the `*x*`-direction, and another transform for the `*y*`-direction.

This "generic" version can handle any given child transform in the `*x*`- and `*y*`-directions.

matplotlib.transforms.CompositeAffine2D

```
CompositeAffine2D(a, b, **kwargs)
```

A composite transform formed by applying transform **a** then transform **b**.

This version is an optimization that handles the case where both **a** and **b** are 2D affines.

matplotlib.transforms.CompositeGenericTransform

```
CompositeGenericTransform(a, b, **kwargs)
```

A composite transform formed by applying transform **a** then transform **b**.

This "generic" version can handle any two arbitrary transformations.

matplotlib.transforms.IdentityTransform

```
IdentityTransform(*args, **kwargs)
```

A special class that does one thing, the identity transform, in a fast way.

matplotlib.transforms.LockableBbox

```
LockableBbox(bbox, x0=None, y0=None, x1=None, y1=None, **kwargs)
```

A ``Bbox`` where some elements may be locked at certain values.

When the child bounding box changes, the bounds of this bbox will update accordingly with the exception of the locked elements.

matplotlib.transforms.Path

```
Path(vertices, codes=None, _interpolation_steps=1, closed=False, readonly=False)
```

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- **vertices**: an (N, 2) float array of vertices
- **codes**: an N-length ``numpy.uint8`` array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three ``CURVE4`` codes.

The code types are:

- ``STOP`` : 1 vertex (ignored)

A marker for the end of the entire path (currently not required and ignored)

- ``MOVETO`` : 1 vertex

Pick up the pen and move to the given vertex.

- ``LINETO`` : 1 vertex

Draw a line from the current position to the given vertex.

- ``CURVE3`` : 1 control point, 1 endpoint

Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ``CURVE4`` : 2 control points, 1 endpoint

Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ``CLOSEPOLY`` : 1 vertex (ignored)

Draw a line segment to the start point of the current polyline.

If `*codes*` is None, it is interpreted as a ``MOVETO`` followed by a series of ``LINETO``.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use ``iter_segments`` or ``cleaned`` to get the vertex/code pairs. This helps, in particular, to consistently handle the case of `*codes*` being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'

.. note::

The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

matplotlib.transforms.ScaledTranslation

```
ScaledTranslation(xt, yt, scale_trans, **kwargs)
```

A transformation that translates by `*xt*` and `*yt*`, after `*xt*` and `*yt*` have been transformed by `*scale_trans*`.

matplotlib.transforms.Transform

```
Transform(shorthand_name=None)
```

The base class of all ``TransformNode`` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of ``Affine2D``.

Subclasses of this class should override the following members (at minimum):

- :attr:`input_dims`

- :attr: `output_dims`
- :meth: `transform`
- :meth: `inverted` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr: `is_separable` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr: `has_inverse` (defaults to True if :meth: `inverted` is overridden, False otherwise)

If the transform needs to do something non-standard with `matplotlib.path.Path`` objects, such as adding curves where there were once line segments, it should override:

- :meth: `transform_path`

matplotlib.transforms.TransformNode

```
TransformNode(shorthand_name=None)
```

The base class for anything that participates in the transform tree and needs to invalidate its parents or be invalidated. This includes classes that are not really transforms, such as bounding boxes, since some transforms depend on bounding boxes to compute their values.

matplotlib.transforms.TransformWrapper

```
TransformWrapper(child)
```

A helper class that holds a single child transform and acts equivalently to it.

This is useful if a node of the transform tree must be replaced at run time with a transform of a different type. This class allows that replacement to correctly trigger invalidation.

`TransformWrapper` instances must have the same input and output dimensions during their entire lifetime, so the child transform may only be replaced with another child transform of the same dimensions.

matplotlib.transforms.TransformedBbox

```
TransformedBbox(bbox, transform, **kwargs)
```

A `Bbox` that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.

matplotlib.transforms.TransformedPatchPath

```
TransformedPatchPath(patch)
```

A `TransformedPatchPath` caches a non-affine transformed copy of the `~.patches.Patch`. This cached copy is automatically updated when the non-affine part of the transform or the patch changes.

matplotlib.transforms.TransformPath

```
TransformedPath(path, transform)
```

A `TransformedPath` caches a non-affine transformed copy of the `~.path.Path`. This cached copy is automatically updated when the non-affine part of the transform changes.

.. note::

Paths are considered immutable by this class. Any update to the path's vertices/codes will not trigger a transform recomputation.

matplotlib.transforms.blended_transform_factory

```
blended_transform_factory(x_transform, y_transform)
```

Create a new "blended" transform using `*x_transform*` to transform the `*x*-axis` and `*y_transform*` to transform the `*y*-axis`.

A faster version of the blended transform is returned for the case where both child transforms are affine.

matplotlib.transforms.composite_transform_factory

```
composite_transform_factory(a, b)
```

Create a new composite transform that is the result of applying transform `a` then transform `b`.

Shortcut versions of the blended transform are provided for the case where both child transforms are affine, or one or the other is the identity transform.

Composite transforms may also be created using the `'+'` operator, e.g.::

```
c = a + b
```

matplotlib.transforms.interval_contains

```
interval_contains(interval, val)
```

Check, inclusively, whether an interval includes a given value.

Parameters

`interval` : (float, float)

The endpoints of the interval.

`val` : float

Value to check is within interval.

Returns

bool

Whether *val* is within the *interval*.

matplotlib.transforms.interval_contains_open

```
interval_contains_open(interval, val)
```

Check, excluding endpoints, whether an interval includes a given value.

Parameters

interval : (float, float)

The endpoints of the interval.

val : float

Value to check is within interval.

Returns

bool

Whether *val* is within the *interval*.

matplotlib.transforms.nonsingular

```
nonsingular(vmin, vmax, expander=0.001, tiny=1e-15, increasing=True)
```

Modify the endpoints of a range as needed to avoid singularities.

Parameters

vmin, vmax : float

The initial endpoints.

expander : float, default: 0.001

Fractional amount by which *vmin* and *vmax* are expanded if the original interval is too small, based on *tiny*.

tiny : float, default: 1e-15

Threshold for the ratio of the interval to the maximum absolute value of its endpoints. If the interval is smaller than this, it will be expanded. This value should be around 1e-15 or larger; otherwise the interval will be approaching the double precision resolution limit.

increasing : bool, default: True

If True, swap *vmin*, *vmax* if *vmin* > *vmax*.

Returns

vmin, vmax : float

Endpoints, expanded and/or swapped if necessary.

If either input is inf or NaN, or if both inputs are 0 or very close to zero, it returns -*expander*, *expander*.

matplotlib.transforms.offset_copy

```
offset_copy(trans, fig=None, x=0.0, y=0.0, units='inches')
```

Return a new transform with an added offset.

Parameters

`trans` : ``Transform`` subclass

Any transform, to which offset will be applied.

`fig` : ``~matplotlib.figure.Figure``, default: None

Current figure. It can be None if `*units*` are 'dots'.

`x, y` : float, default: 0.0

The offset to apply.

`units` : {'inches', 'points', 'dots'}, default: 'inches'

Units of the offset.

Returns

``Transform`` subclass

Transform with applied offset.

matplotlib.tri

```
tri(...)
```

Unstructured triangular grid functions.

matplotlib.tri.CubicTriInterpolator

```
CubicTriInterpolator(triangulation, z, kind='min_E', trifinder=None, dz=None)
```

Cubic interpolator on a triangular grid.

In one-dimension - on a segment - a cubic interpolating function is

defined by the values of the function and its derivative at both ends.

This is almost the same in 2D inside a triangle, except that the values of the function and its 2 derivatives have to be defined at each triangle node.

The `CubicTriInterpolator` takes the value of the function at each node - provided by the user - and internally computes the value of the derivatives, resulting in a smooth interpolation.

(As a special feature, the user can also impose the value of the derivatives at each node, but this is not supposed to be the common usage.)

Parameters

`triangulation` : ``~matplotlib.tri.Triangulation``

The triangulation to interpolate over.

`z` : (npoints,) array-like

Array of values, defined at grid points, to interpolate between.

`kind` : {'min_E', 'geom', 'user'}, optional

Choice of the smoothing algorithm, in order to compute the interpolant derivatives (defaults to 'min_E'):

- if 'min_E': (default) The derivatives at each node is computed to minimize a bending energy.
- if 'geom': The derivatives at each node is computed as a weighted average of relevant triangle normals. To be used for speed optimization (large grids).
- if 'user': The user provides the argument *dz*, no computation is hence needed.

trifinder : `~matplotlib.tri.TriFinder``, optional

If not specified, the Triangulation's default TriFinder will be used by calling `~.Triangulation.get_trifinder``.

dz : tuple of array-likes (dzdx, dzdy), optional

Used only if *kind* ='user'. In this case *dz* must be provided as (dzdx, dzdy) where dzdx, dzdy are arrays of the same shape as *z* and are the interpolant first derivatives at the *triangulation* points.

Methods

`__call__`` (x, y) : Returns interpolated values at (x, y) points.

`gradient`` (x, y) : Returns interpolated derivatives at (x, y) points.

Notes

This note is a bit technical and details how the cubic interpolation is computed.

The interpolation is based on a Clough-Tocher subdivision scheme of the *triangulation* mesh (to make it clearer, each triangle of the grid will be divided in 3 child-triangles, and on each child triangle the interpolated function is a cubic polynomial of the 2 coordinates). This technique originates from FEM (Finite Element Method) analysis; the element used is a reduced Hsieh-Clough-Tocher (HCT) element. Its shape functions are described in [1].

The assembled function is guaranteed to be C1-smooth, i.e. it is continuous and its first derivatives are also continuous (this is easy to show inside the triangles but is also true when crossing the edges).

In the default case (*kind* ='min_E'), the interpolant minimizes a curvature energy on the functional space generated by the HCT element shape functions - with imposed values but arbitrary derivatives at each node. The minimized functional is the integral of the so-called total curvature (implementation based on an algorithm from [2] - PCG sparse solver):

.. math::

$$E(z) = \frac{1}{2} \int_{\Omega} \left(\left(\frac{\partial^2 z}{\partial x^2} \right)^2 + \left(\frac{\partial^2 z}{\partial y^2} \right)^2 + 2 \left(\frac{\partial^2 z}{\partial x \partial y} \right)^2 \right) dx, dy$$

If the case *kind* ='geom' is chosen by the user, a simple geometric

approximation is used (weighted average of the triangle normal vectors), which could improve speed on very large grids.

References

-
- .. [1] Michel Bernadou, Kamal Hassan, "Basis functions for general Hsieh-Clough-Tocher triangles, complete or reduced.", International Journal for Numerical Methods in Engineering, 17(5):784 - 789. 2.01.
- .. [2] C.T. Kelley, "Iterative Methods for Optimization".

matplotlib.tri.LinearTriInterpolator

```
LinearTriInterpolator(triangulation, z, trifinder=None)
```

Linear interpolator on a triangular grid.

Each triangle is represented by a plane so that an interpolated value at point (x, y) lies on the plane of the triangle containing (x, y). Interpolated values are therefore continuous across the triangulation, but their first derivatives are discontinuous at edges between triangles.

Parameters

triangulation : `~matplotlib.tri.Triangulation``
The triangulation to interpolate over.

z : (npoints,) array-like
Array of values, defined at grid points, to interpolate between.

trifinder : `~matplotlib.tri.TriFinder``, optional
If this is not specified, the Triangulation's default TriFinder will be used by calling ``.Triangulation.get_trifinder``.

Methods

`__call__`` (x, y) : Returns interpolated values at (x, y) points.

`gradient`` (x, y) : Returns interpolated derivatives at (x, y) points.

matplotlib.tri.TrapezoidMapTriFinder

```
TrapezoidMapTriFinder(triangulation)
```

`~matplotlib.tri.TriFinder`` class implemented using the trapezoid map algorithm from the book "Computational Geometry, Algorithms and Applications", second edition, by M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf.

The triangulation must be valid, i.e. it must not have duplicate points, triangles formed from colinear points, or overlapping triangles. The algorithm has some tolerance to triangles formed from colinear points, but this should not be relied upon.

matplotlib.tri.TriAnalyzer

```
TriAnalyzer(triangulation)
```

Define basic tools for triangular mesh analysis and improvement.

A `TriAnalyzer` encapsulates a `Triangulation`` object and provides basic tools for mesh analysis and mesh improvement.

Attributes

`scale_factors`

Parameters

`triangulation` : `~matplotlib.tri.Triangulation``

The encapsulated triangulation to analyze.

matplotlib.tri.TriContourSet

```
TriContourSet(ax, *args, **kwargs)
```

Create and store a set of contour lines or filled regions for a triangular grid.

This class is typically not instantiated directly by the user but by `~.Axes.tricontour`` and `~.Axes.tricontourf``.

Attributes

`levels` : array

The values of the contour levels.

`layers` : array

Same as levels for line contours; half-way between levels for filled contours. See `ContourSet._process_colors``.

matplotlib.tri.TriFinder

```
TriFinder(triangulation)
```

Abstract base class for classes used to find the triangles of a `Triangulation` in which (x, y) points lie.

Rather than instantiate an object of a class derived from `TriFinder`, it is usually better to use the function `Triangulation.get_trifinder``.

Derived classes implement `__call__(x, y)` where x and y are array-like point coordinates of the same shape.

matplotlib.tri.TriInterpolator

```
TriInterpolator(triangulation, z, trifinder=None)
```

Abstract base class for classes used to interpolate on a triangular grid.

Derived classes implement the following methods:

- ``__call__(x, y)``,

where x, y are array-like point coordinates of the same shape, and that returns a masked array of the same shape containing the interpolated z-values.

- ``gradient(x, y)``,

where x, y are array-like point coordinates of the same shape, and that returns a list of 2 masked arrays of the same shape containing the 2 derivatives of the interpolator (derivatives of interpolated z values with respect to x and y).

matplotlib.tri.TriRefiner

`TriRefiner(triangulation)`

Abstract base class for classes implementing mesh refinement.

A TriRefiner encapsulates a Triangulation object and provides tools for mesh refinement and interpolation.

Derived classes must implement:

- ``refine_triangulation(return_tri_index=False, **kwargs)``, where the optional keyword arguments *kwargs* are defined in each TriRefiner concrete implementation, and which returns:

- a refined triangulation,
- optionally (depending on *return_tri_index*), for each point of the refined triangulation: the index of the initial triangulation triangle to which it belongs.

- ``refine_field(z, triinterpolator=None, **kwargs)``, where:

- *z* array of field values (to refine) defined at the base triangulation nodes,
- *triinterpolator* is an optional `~matplotlib.tri.TriInterpolator`,
- the other optional keyword arguments *kwargs* are defined in each TriRefiner concrete implementation;

and which returns (as a tuple) a refined triangular mesh and the interpolated values of the field at the refined triangulation nodes.

matplotlib.tri.Triangulation

`Triangulation(x, y, triangles=None, mask=None)`

An unstructured triangular grid consisting of npoints points and ntri triangles. The triangles can either be specified by the user or automatically generated using a Delaunay triangulation.

Parameters

x, y : (npoints,) array-like
Coordinates of grid points.

triangles : (ntri, 3) array-like of int, optional
For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.
mask : (ntri,) array-like of bool, optional
Which triangles are masked out.

Attributes

triangles : (ntri, 3) array of int
For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If you want to take the *mask* into account, use ``get_masked_triangles`` instead.
mask : (ntri, 3) array of bool or None
Masked out triangles.
is_delaunay : bool
Whether the Triangulation is a calculated Delaunay triangulation (where *triangles* was not specified) or not.

Notes

For a Triangulation to be valid it must not have duplicate points, triangles formed from colinear points, or overlapping triangles.

matplotlib.tri.UniformTriRefiner

```
UniformTriRefiner(triangulation)
```

Uniform mesh refinement by recursive subdivisions.

Parameters

triangulation : ``~matplotlib.tri.Triangulation``
The encapsulated triangulation (to be refined)

matplotlib.tri.tricontour

```
tricontour(ax, *args, **kwargs)
```

Draw contour lines on an unstructured triangular grid.

Call signatures::

```
tricontour(triangulation, z, [levels], ...)  
tricontour(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a ``Triangulation`` object as the first parameter, or by passing the points *x*, *y* and optionally the *triangles* and a *mask*. See ``Triangulation`` for an explanation of these parameters. If neither of *triangulation* or *triangles* are given, the triangulation is calculated on the fly.

It is possible to pass *triangles* positionally, i.e.
``tricontour(x, y, triangles, z, ...)``. However, this is discouraged. For more

clarity, pass `*triangles*` via keyword argument.

Parameters

`triangulation` : ``~.Triangulation``, optional

An already created triangular grid.

`x`, `y`, `triangles`, `mask`

Parameters defining the triangular grid. See ``~.Triangulation``.

This is mutually exclusive with specifying `*triangulation*`.

`z` : array-like

The height values over which the contour is drawn. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

.. note::

All values in `*z*` must be finite. Hence, nan and inf values must either be removed or ``~.Triangulation.set_mask`` be used.

`levels` : int or array-like, optional

Determines the number and positions of the contour lines / regions.

If an int `*n*`, use ``~matplotlib.ticker.MaxNLocator``, which tries to automatically choose no more than `*n+1*` "nice" contour levels between between minimum and maximum numeric values of `*Z*`.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

``~matplotlib.tri.TriContourSet``

Other Parameters

`colors` : `:mpltype:`color`` or list of `:mpltype:`color``, optional

The colors of the levels, i.e., the contour lines.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. ```red``` instead of ```[red]``` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value `*None*`), the colormap specified by `*cmap*` will be used.

`alpha` : float, default: 1

The alpha blending value, between 0 (transparent) and 1 (opaque).

`cmap` : str or ``~matplotlib.colors.Colormap``, default: `:rc:`image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*colors*` is set.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `.Normalize`` or one of its subclasses (see `:ref:`colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin, vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a ``str` *norm*` name together with `*vmin*/vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`origin` : {`*None*`, 'upper', 'lower', 'image'}, default: None

Determines the orientation and exact position of `*z*` by specifying the position of `z[0, 0]``. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: `z[0, 0]`` is at X=0, Y=0 in the lower left corner.
- 'lower': `z[0, 0]`` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `z[0, 0]`` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `:rc:`image.origin``.

`extent` : (x0, x1, y0, y1), optional

If `*origin*` is not `*None*`, then `*extent*` is interpreted as in `.imshow``: it gives the outer pixel boundaries. In this case, the position of `z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then `(*x0*, *y0*)` is the position of `z[0, 0]`, and `(*x1*, *y1*)` is the position of `z[-1, -1]`.

This argument is ignored if `*X*` and `*Y*` are specified in the call to `contour`.

`locator` : `ticker.Locator` subclass, optional

The locator is used to determine the contour levels if they are not given explicitly via `*levels*`.

Defaults to `~.ticker.MaxNLocator``.

`extend` : {'neither', 'both', 'min', 'max'}, default: 'neither'

Determines the ``tricontour``-coloring of values that are outside the **levels** range.

If 'neither', values outside the **levels** range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the **levels** range.

Values below ``min(levels)`` and above ``max(levels)`` are mapped to the under/over values of the `.Colormap`. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using `.Colormap.set_under` and `.Colormap.set_over`.

.. note::

An existing `.TriContourSet` does not get notified if properties of its colormap are changed. Therefore, an explicit call to `.ContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the `.TriContourSet` because it internally calls `.ContourSet.changed()`.

xunits, yunits : registered units, optional
Override axis units by specifying an instance of a
:class:`matplotlib.units.ConversionInterface`.

antialiased : bool, optional
Enable antialiasing, overriding the defaults. For
filled contours, the default is **True**. For line contours,
it is taken from `:rc:`lines.antialiased``.

linewidths : float or array-like, default: `:rc:`contour.linewidth``
The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with
the linewidths in the order specified.

If None, this falls back to `:rc:`lines.linewidth``.

linestyles : {**None**, 'solid', 'dashed', 'dashdot', 'dotted'}, optional
If **linestyles** is **None**, the default is 'solid' unless the lines are
monochrome. In that case, negative contours will take their linestyle
from `:rc:`contour.negative_linestyle`` setting.

linestyles can also be an iterable of the above strings specifying a
set of linestyles to be used. If this iterable is shorter than the
number of contour levels it will be repeated as necessary.

matplotlib.tri.tricontourf

```
tricontourf(ax, *args, **kwargs)
```

Draw contour regions on an unstructured triangular grid.

Call signatures::

```
tricontourf(triangulation, z, [levels], ...)
tricontourf(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a `Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. See `Triangulation`` for an explanation of these parameters. If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly.

It is possible to pass `*triangles*` positionally, i.e. `tricontourf(x, y, triangles, z, ...)``. However, this is discouraged. For more clarity, pass `*triangles*` via keyword argument.

Parameters

`triangulation` : `Triangulation``, optional
An already created triangular grid.

`x`, `y`, `triangles`, `mask`
Parameters defining the triangular grid. See `Triangulation``.
This is mutually exclusive with specifying `*triangulation*`.

`z` : array-like
The height values over which the contour is drawn. Color-mapping is controlled by `*cmap*`, `*norm*`, `*vmin*`, and `*vmax*`.

.. note::
All values in `*z*` must be finite. Hence, nan and inf values must either be removed or `Triangulation.set_mask`` be used.

`levels` : int or array-like, optional
Determines the number and positions of the contour lines / regions.

If an int `*n*`, use `~matplotlib.ticker.MaxNLocator``, which tries to automatically choose no more than `*n+1*` "nice" contour levels between between minimum and maximum numeric values of `*Z*`.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

Returns

`~matplotlib.tri.TriContourSet``

Other Parameters

`colors` : `:mpltype:`color`` or list of `:mpltype:`color``, optional
The colors of the levels, i.e., the contour regions.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. ```red``` instead of ```[red]``` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value `*None*`), the colormap specified by `*cmap*` will be used.

`alpha` : float, default: 1

The alpha blending value, between 0 (transparent) and 1 (opaque).

`cmap` : str or `~matplotlib.colors.Colormap``, default: `:rc:~image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if `*colors*` is set.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `~matplotlib.colors.Normalize`` or one of its subclasses (see `:ref:~colormapnorms``).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names(``. In that case, a suitable `~matplotlib.colors.Normalize`` subclass is dynamically generated and instantiated.

This parameter is ignored if `*colors*` is set.

`vmin`, `vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a `~matplotlib.colors.Normalize`` name together with `*vmin*/vmax*` is acceptable).

If `*vmin*` or `*vmax*` are not given, the default color scaling is based on `*levels*`.

This parameter is ignored if `*colors*` is set.

`origin` : {`*None*`, 'upper', 'lower', 'image'}, default: None

Determines the orientation and exact position of `*z*` by specifying the position of ```z[0, 0]```. This is only relevant, if `*X*`, `*Y*` are not given.

- `*None*`: ```z[0, 0]``` is at X=0, Y=0 in the lower left corner.
- 'lower': ```z[0, 0]``` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': ```z[0, 0]``` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `:rc:~image.origin``.

`extent` : (x0, x1, y0, y1), optional

If `*origin*` is not `*None*`, then `*extent*` is interpreted as in `~matplotlib.pyplot.imshow``: it

gives the outer pixel boundaries. In this case, the position of `z[0, 0]` is the center of the pixel, not a corner. If `*origin*` is `*None*`, then `(*x0*, *y0*)` is the position of `z[0, 0]`, and `(*x1*, *y1*)` is the position of `z[-1, -1]`.

This argument is ignored if `*X*` and `*Y*` are specified in the call to `contour`.

`locator` : `ticker.Locator` subclass, optional

The locator is used to determine the contour levels if they are not given explicitly via `*levels*`.

Defaults to `~.ticker.MaxNLocator``.

`extend` : `{'neither', 'both', 'min', 'max'}`, default: `'neither'`

Determines the ```tricontourf```-coloring of values that are outside the `*levels*` range.

If `'neither'`, values outside the `*levels*` range are not colored. If `'min'`, `'max'` or `'both'`, color the values below, above or below and above the `*levels*` range.

Values below ```min(levels)``` and above ```max(levels)``` are mapped to the under/over values of the ``.Colormap``. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using ``.Colormap.set_under`` and ``.Colormap.set_over``.

.. note::

An existing ``.TriContourSet`` does not get notified if properties of its colormap are changed. Therefore, an explicit call to ``.ContourSet.changed()``` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the ``.TriContourSet`` because it internally calls ``.ContourSet.changed()```.

`xunits, yunits` : registered units, optional

Override axis units by specifying an instance of a `:class:`matplotlib.units.ConversionInterface``.

`antialiased` : bool, optional

Enable antialiasing, overriding the defaults. For filled contours, the default is `*True*`. For line contours, it is taken from `:rc:`lines.antialiased``.

`hatches` : list[str], optional

A list of crosshatch patterns to use on the filled areas. If None, no hatching will be added to the contour.

Notes

``.tricontourf`` fills intervals that are closed at the top; that is, for boundaries `*z1*` and `*z2*`, the filled region is::

$z1 < Z \leq z2$

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

matplotlib.tri.tripcolor

```
tripcolor(ax, *args, alpha=1.0, norm=None, cmap=None, vmin=None, vmax=None,
          shading='flat', facecolors=None, **kwargs)
```

Create a pseudocolor plot of an unstructured triangular grid.

Call signatures::

```
tripcolor(triangulation, c, *, ...)
tripcolor(x, y, c, *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a `.Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. See `.Triangulation`` for an explanation of these parameters.

It is possible to pass the triangles positionally, i.e. `tripcolor(x, y, triangles, c, ...)``. However, this is discouraged. For more clarity, pass `*triangles*` via keyword argument.

If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly. In this case, it does not make sense to provide colors at the triangle faces via `*c*` or `*facecolors*` because there are multiple possible triangulations for a group of points and you don't know which triangles will be constructed.

Parameters

triangulation : `.Triangulation``

An already created triangular grid.

x, y, triangles, mask

Parameters defining the triangular grid. See `.Triangulation``.

This is mutually exclusive with specifying `*triangulation*`.

c : array-like

The color values, either for the points or for the triangles. Which one is automatically inferred from the length of `*c*`, i.e. does it match the number of points or the number of triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the keyword argument `facecolors=c`` instead of just `c``.

This parameter is position-only.

facecolors : array-like, optional

Can be used alternatively to `*c*` to specify colors at the triangle faces. This parameter takes precedence over `*c*`.

shading : {'flat', 'gouraud'}, default: 'flat'

If 'flat' and the color values `*c*` are defined at points, the color values used for each triangle are from the mean c of the triangle's three points. If `*shading*` is 'gouraud' then color values must be defined at points.

cmap : str or `~matplotlib.colors.Colormap``, default: `:rc:~image.cmap``

The Colormap instance or registered colormap name used to map scalar data to colors.

`norm` : str or `~matplotlib.colors.Normalize``, optional

The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using `*cmap*`. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `~matplotlib.colors.Normalize`` or one of its subclasses (see :ref:`colormapnorms`).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `~matplotlib.scale.get_scale_names()``. In that case, a suitable `~matplotlib.colors.Normalize`` subclass is dynamically generated and instantiated.

`vmin, vmax` : float, optional

When using scalar data and no explicit `*norm*`, `*vmin*` and `*vmax*` define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use `*vmin*/vmax*` when a `*norm*` instance is given (but using a `~matplotlib.colors.Normalize`` name together with `*vmin*/vmax*` is acceptable).

`colorizer` : `~matplotlib.colorbar.Colorizer`` or None, default: None

The Colorizer object used to map color to data. If None, a Colorizer object is created from a `*norm*` and `*cmap*`.

Returns

`~matplotlib.collections.PolyCollection`` or `~matplotlib.collections.TriMesh``
The result depends on `*shading*`: For ```shading='flat'``` the result is a `~matplotlib.collections.PolyCollection``, for ```shading='gouraud'``` the result is a `~matplotlib.collections.TriMesh``.

Other Parameters

`**kwargs` : `~matplotlib.collections.Collection`` properties

Properties:

`agg_filter`: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image

`alpha`: array-like or float or None

`animated`: bool

`antialiased` or `aa` or `antialiaseds`: bool or list of bools

`array`: array-like or None

`capstyle`: `~matplotlib.collections.Collection`.CapStyle`` or {'butt', 'projecting', 'round'}

`clim`: (vmin: float, vmax: float)

`clip_box`: `~matplotlib.transforms.BboxBase`` or None

`clip_on`: bool

`clip_path`: Patch or (Path, Transform) or None

`cmap`: `~matplotlib.colors.Colormap`` or str or None

`color`: :mpltype:`color` or list of RGBA tuples

`edgecolor` or `ec` or `edgcolors`: :mpltype:`color` or list of :mpltype:`color` or 'face'

`facecolor` or `facecolors` or `fc`: :mpltype:`color` or list of :mpltype:`color`

`figure`: `~matplotlib.figure.Figure`` or `~matplotlib.figure.SubFigure``

gid: str
 hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
 hatch_linewidth: unknown
 in_layout: bool
 joinstyle: ``.JoinStyle`` or {'miter', 'round', 'bevel'}
 label: object
 linestyle or dashes or linestyles or ls: str or tuple or list thereof
 linewidth or linewidths or lw: float or list of floats
 mouseover: bool
 norm: ``.Normalize`` or str or None
 offset_transform or transOffset: ``.Transform``
 offsets: (N, 2) or (2,) array-like
 path_effects: list of ``.AbstractPathEffect``
 paths: unknown
 picker: None or bool or float or callable
 pickradius: float
 rasterized: bool
 sketch_params: (scale: float, length: float, randomness: float)
 snap: bool or None
 transform: ``.~matplotlib.transforms.Transform``
 url: str
 urls: list of str or None
 visible: bool
 zorder: float

matplotlib.tri.triplot

```
triplot(ax, *args, **kwargs)
```

Draw an unstructured triangular grid as lines and/or markers.

Call signatures::

```

triplot(triangulation, ...)
triplot(x, y, [triangles], *, [mask=mask], ...)
```

The triangular grid can be specified either by passing a ``.Triangulation`` object as the first parameter, or by passing the points `*x*`, `*y*` and optionally the `*triangles*` and a `*mask*`. If neither of `*triangulation*` or `*triangles*` are given, the triangulation is calculated on the fly.

Parameters

 triangulation : ``.Triangulation``
 An already created triangular grid.
 x, y, triangles, mask
 Parameters defining the triangular grid. See ``.Triangulation``.
 This is mutually exclusive with specifying `*triangulation*`.
 other_parameters
 All other args and kwargs are forwarded to ``.~.Axes.plot``.

Returns

 lines : ``.~matplotlib.lines.Line2D``

The drawn triangles edges.
markers : ``~matplotlib.lines.Line2D`
The drawn marker nodes.

matplotlib.typing

`typing(...)`

Typing support for Matplotlib

This module contains Type aliases which are useful for Matplotlib and potentially downstream libraries.

.. admonition:: Provisional status of typing

The ``typing`` module and type stub files are considered provisional and may change at any time without a deprecation period.

matplotlib.typing.Artist

`Artist()`

Abstract base class for objects that render into a FigureCanvas.

Typically, all visible elements in a figure are subclasses of Artist.

matplotlib.typing.Bbox

`Bbox(points, **kwargs)`

A mutable bounding box.

Examples

****Create from known bounds****

The default constructor takes the boundary "points" ``[[xmin, ymin], [xmax, ymax]]``.

```
>>> Bbox([[1, 1], [3, 7]])  
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" ``(xmin, ymin, xmax, ymax)``

```
>>> Bbox.from_extents(1, 1, 3, 7)  
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" ``(xmin, ymin, width, height)``.

```
>>> Bbox.from_bounds(1, 1, 2, 6)  
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

****Create from collections of points****

The "empty" object for accumulating Bboxes is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting ``ignore=True`` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

.. warning::

It is recommended to always specify ``ignore`` explicitly. If not, the default value of ``ignore`` can be changed at any time by code with access to your Bbox, for example using the method `~.Bbox.ignore`.

****Properties of the ``null`` bbox****

.. note::

The current behavior of `Bbox.null()` may be surprising as it does not have all of the properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[ -inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[ -inf, -inf], [inf, inf]])
```

matplotlib.typing.CapStyle

```
CapStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the two endpoints (caps) of an unclosed line are drawn.

How to draw the start and end points of lines that represent a closed curve (i.e. that end in a `~.path.Path.CLOSEPOLY`) is controlled by the line's `JoinStyle`. For all other lines, how the start and end points are drawn is controlled by the `*CapStyle*`.

For a visual impression of each `*CapStyle*`, view these docs online [<CapStyle>](#) or run `CapStyle.demo`.

By default, `~.backend_bases.GraphicsContextBase`` draws a stroked line as squared off at its endpoints.

****Supported values:****

.. rst-class:: value-list

'butt'

the line is squared off at its endpoint.

'projecting'

the line is squared off as in `*butt*`, but the filled in area extends beyond the endpoint a distance of ```linewidth/2```.

'round'

like `*butt*`, but a semicircular cap is added to the end of the line, of radius ```linewidth/2```.

.. plot::

:alt: Demo of possible CapStyle's

```
from matplotlib._enums import CapStyle
CapStyle.demo()
```

matplotlib.typing.JoinStyle

```
JoinStyle(value, names=None, *, module=None, qualname=None, type=None, start=1,
boundary=None)
```

Define how the connection between two line segments is drawn.

For a visual impression of each `*JoinStyle*`, view these docs online [<JoinStyle>](#), or run `JoinStyle.demo`.

Lines in Matplotlib are typically defined by a 1D `~.path.Path`` and a finite ```linewidth```, where the underlying 1D `~.path.Path`` represents the center of the stroked line.

By default, `~.backend_bases.GraphicsContextBase`` defines the boundaries of a stroked line to simply be every point within some radius, ```linewidth/2```, away from any point of the center line. However, this results in corners appearing "rounded", which may not be the desired behavior if you are drawing, for example, a polygon or pointed star.

****Supported values:****

.. rst-class:: value-list

'miter'

the "arrow-tip" style. Each boundary of the filled-in area will extend in a straight line parallel to the tangent vector of the centerline at the point it meets the corner, until they meet in a sharp point.

'round'

stokes every point within a radius of ``linewidth/2`` of the center lines.

'bevel'

the "squared-off" style. It can be thought of as a rounded corner where the "circular" part of the corner has been cut off.

.. note::

Very long miter tips are cut off (to form a *bevel*) after a backend-dependent limit called the "miter limit", which specifies the maximum allowed ratio of miter length to line width. For example, the PDF backend uses the default value of 10 specified by the PDF standard, while the SVG backend does not even specify the miter limit, resulting in a default value of 4 per the SVG specification. Matplotlib does not currently allow the user to adjust this parameter.

A more detailed description of the effect of a miter limit can be found in the `Mozilla Developer Docs

<<https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-miterlimit>>`_

.. plot::

:alt: Demo of possible JoinStyle's

```
from matplotlib._enums import JoinStyle
JoinStyle.demo()
```

matplotlib.typing.MarkerStyle

```
MarkerStyle(marker, fillstyle=None, transform=None, capstyle=None, joinstyle=None)
```

A class representing marker types.

Instances are immutable. If you need to change anything, create a new instance.

Attributes

markers : dict

All known markers.

filled_markers : tuple

All known filled markers. This is a subset of *markers*.

fillstyles : tuple

The supported fillstyles.

matplotlib.typing.RendererBase

```
RendererBase()
```

An abstract base class to handle drawing/rendering operations.

The following methods must be implemented in the backend for full functionality (though just implementing `draw_path` alone would give a highly capable backend):

- * `draw_path`
- * `draw_image`
- * `draw_gouraud_triangles`

The following methods *should* be implemented in the backend for optimization reasons:

- * `draw_text`
- * `draw_markers`
- * `draw_path_collection`
- * `draw_quad_mesh`

matplotlib.typing.Transform

```
Transform(shorthand_name=None)
```

The base class of all `TransformNode` instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of `Affine2D`.

Subclasses of this class should override the following members (at minimum):

- :attr: `input_dims`
- :attr: `output_dims`
- :meth: `transform`
- :meth: `inverted` (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- :attr: `is_separable` (defaults to True for 1D -> 1D transforms, False otherwise)
- :attr: `has_inverse` (defaults to True if `meth: inverted` is overridden, False otherwise)

If the transform needs to do something non-standard with `matplotlib.path.Path` objects, such as adding curves where there were once line segments, it should override:

- :meth: `transform_path`

matplotlib.units

```
units(...)
```

The classes here provide support for using custom classes with Matplotlib, e.g., those that do not expose the array interface but know how to convert themselves to arrays. It also supports classes with units and units conversion. Use cases include converters for custom objects, e.g., a list of datetime objects, as well as for objects that are unit aware. We don't assume any particular units implementation; rather a units implementation must register with the Registry converter dictionary and provide a `ConversionInterface`. For example, here is a complete implementation which supports plotting with native datetime objects::

```
import matplotlib.units as units
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import datetime

class DateConverter(units.ConversionInterface):

    @staticmethod
    def convert(value, unit, axis):
        "Convert a datetime value to a scalar or array."
        return dates.date2num(value)

    @staticmethod
    def axisinfo(unit, axis):
        "Return major and minor tick locators and formatters."
        if unit != 'date':
            return None
        majloc = dates.AutoDateLocator()
        majfmt = dates.AutoDateFormatter(majloc)
        return units.AxisInfo(majloc=majloc, majfmt=majfmt, label='date')

    @staticmethod
    def default_units(x, axis):
        "Return the default unit for x or None."
        return 'date'

# Finally we register our object type with the Matplotlib units registry.
units.registry[datetime.date] = DateConverter()
```

matplotlib.units.AxisInfo

```
AxisInfo(majloc=None, minloc=None, majfmt=None, minfmt=None, label=None,
default_limits=None)
```

Information to support default axis labeling, tick labeling, and limits.

An instance of this class must be returned by `ConversionInterface.axisinfo`.

matplotlib.units.ConversionError

```
ConversionError(...)
```

Inappropriate argument type.

matplotlib.units.ConversionInterface

```
ConversionInterface()
```

The minimal interface for a converter to take custom data types (or sequences) and convert them to values Matplotlib can use.

matplotlib.units.DecimalConverter

```
DecimalConverter()
```

Converter for decimal.Decimal data to float.

matplotlib.units.Registry

```
Registry(...)
```

Register types with conversion interface.

matplotlib.use

```
use(backend, *, force=True)
```

Select the backend used for rendering and GUI integration.

If pyplot is already imported, `~matplotlib.pyplot.switch_backend` is used and if the new backend is different than the current backend, all Figures will be closed.

Parameters

backend : str

The backend to switch to. This can either be one of the standard backend names, which are case-insensitive:

- interactive backends:

GTK3Agg, GTK3Cairo, GTK4Agg, GTK4Cairo, MacOSX, nbAgg, notebook, QtAgg, QtCairo, TkAgg, TkCairo, WebAgg, WX, WXAgg, WXCairo, Qt5Agg, Qt5Cairo

- non-interactive backends:

agg, cairo, pdf, pgf, ps, svg, template

or a string of the form: ```module://my.module.name```.

notebook is a synonym for nbAgg.

Switching to an interactive backend is not possible if an unrelated event loop has already been started (e.g., switching to GTK3Agg if a TkAgg window has already been opened). Switching to a non-interactive backend is always possible.

force : bool, default: True

If True (the default), raise an `ImportError` if the backend cannot be

set up (either because it fails to import, or because an incompatible GUI interactive framework is already running); if False, silently ignore the failure.

See Also

:ref:`backends`

matplotlib.get_backend

matplotlib.pyplot.switch_backend

matplotlib.validate_backend

```
validate_backend(s)
```

[*Deprecated*]

Notes

.. deprecated:: 3.10

Use matplotlib.rcsetup.validate_backend instead.\

matplotlib.widgets

```
widgets(...)
```

GUI neutral widgets

=====

Widgets that are designed to work for any of the GUI backends.

All of these widgets require you to predefine an `~.axes.Axes`` instance and pass that as the first parameter. Matplotlib doesn't try to be too smart with respect to layout -- you will have to figure out how wide and tall you want your Axes to be to accommodate your widget.

matplotlib.widgets.Affine2D

```
Affine2D(matrix=None, **kwargs)
```

A mutable 2D affine transformation.

matplotlib.widgets.AxesWidget

```
AxesWidget(ax)
```

Widget connected to a single `~matplotlib.axes.Axes``.

To guarantee that the widget remains responsive and not garbage-collected, a reference to the object should be maintained by the user.

This is necessary because the callback registry maintains only weak-refs to the functions, which are member functions of the widget. If there are no references to the widget object it may be garbage collected which will disconnect the callbacks.

Attributes

`ax` : `~matplotlib.axes.Axes``

The parent Axes for the widget.

`canvas` : `~matplotlib.backend_bases.FigureCanvasBase``

The parent figure canvas for the widget.

`active` : bool

If False, the widget does not respond to events.

matplotlib.widgets.Button

```
Button(ax, label, image=None, color='0.85', hovercolor='0.95', *, useblit=True)
```

A GUI neutral button.

For the button to remain responsive you must keep a reference to it.

Call `.on_clicked`` to connect to the button.

Attributes

`ax`

The `~.axes.Axes`` the button renders into.

`label`

A `.Text`` instance.

`color`

The color of the button when not hovering.

`hovercolor`

The color of the button when hovering.

matplotlib.widgets.CheckButtons

```
CheckButtons(ax, labels, actives=None, *, useblit=True, label_props=None, frame_props=None, check_props=None)
```

A GUI neutral set of check buttons.

For the check buttons to remain responsive you must keep a reference to this object.

Connect to the CheckButtons with the `.on_clicked`` method.

Attributes

`ax` : `~matplotlib.axes.Axes``

The parent Axes for the widget.

`labels` : list of `~matplotlib.text.Text``

The text label objects of the check buttons.

matplotlib.widgets.Cursor

```
Cursor(ax, *, horizOn=True, vertOn=True, useblit=False, **lineprops)
```

A crosshair cursor that spans the Axes and moves with mouse cursor.

For the cursor to remain responsive you must keep a reference to it.

Parameters

ax : `~matplotlib.axes.Axes``
The `~.axes.Axes`` to attach the cursor to.
horizOn : bool, default: True
Whether to draw the horizontal line.
vertOn : bool, default: True
Whether to draw the vertical line.
useblit : bool, default: False
Use blitting for faster drawing if supported by the backend.
See the tutorial :ref:`blitting` for details.

Other Parameters

**lineprops
`~.Line2D`` properties that control the appearance of the lines.
See also `~.Axes.axhline``.

Examples

See :doc:`/gallery/widgets/cursor`.

matplotlib.widgets.Ellipse

```
Ellipse(xy, width, height, *, angle=0, **kwargs)
```

A scale-free ellipse.

matplotlib.widgets.EllipseSelector

```
EllipseSelector(ax, onselect=None, *, minspanx=0, minspany=0, useblit=False,  
props=None, spancoords='data', button=None, grab_range=10, handle_props=None,  
interactive=False, state_modifier_keys=None, drag_from_anywhere=False,  
ignore_event_outside=False, use_data_coordinates=False)
```

Select an elliptical region of an Axes.

For the cursor to remain responsive you must keep a reference to it.

Press and release events triggered at the same coordinates outside the selection will clear the selector, except when
```ignore_event_outside=True```.

#### Parameters

-----  
ax : `~matplotlib.axes.Axes``  
The parent Axes for the widget.  
  
onselect : function, optional  
A callback function that is called after a release event and the selection is created, changed or removed.  
It must have the signature::

`def onselect(eclick: MouseEvent, erelease: MouseEvent)`

where `*eclick*` and `*erelease*` are the mouse click and release ``MouseEvent`s` that start and complete the selection.

`minspanx` : float, default: 0

Selections with an x-span less than or equal to `*minspanx*` are removed (when already existing) or cancelled.

`minspany` : float, default: 0

Selections with an y-span less than or equal to `*minspanx*` are removed (when already existing) or cancelled.

`useblit` : bool, default: False

Whether to use blitting for faster drawing (if supported by the backend). See the tutorial :ref:`blitting` for details.

`props` : dict, optional

Properties with which the ellipse is drawn. See ``Patch`` for valid properties.

Default:

```
``dict(facecolor='red', edgecolor='black', alpha=0.2, fill=True)``
```

`spancoords` : {"data", "pixels"}, default: "data"

Whether to interpret `*minspanx*` and `*minspany*` in data or in pixel coordinates.

`button` : ``MouseButton``, list of ``MouseButton``, default: all buttons  
Button(s) that trigger rectangle selection.

`grab_range` : float, default: 10

Distance in pixels within which the interactive tool handles can be activated.

`handle_props` : dict, optional

Properties with which the interactive handles (marker artists) are drawn. See the marker arguments in ``Line2D`` for valid properties. Default values are defined in ```mpl.rcParams``` except for the default value of ```markeredgecolor``` which will be the same as the ```edgecolor``` property in `*props*`.

`interactive` : bool, default: False

Whether to draw a set of handles that allow interaction with the widget after it is drawn.

`state_modifier_keys` : dict, optional

Keyboard modifiers which affect the widget's behavior. Values amend the defaults, which are:

- "move": Move the existing shape, default: no modifier.
- "clear": Clear the current shape, default: "escape".
- "square": Make the shape square, default: "shift".
- "center": change the shape around its center, default: "ctrl".

- "rotate": Rotate the shape around its center between -45° and 45°, default: "r".

"square" and "center" can be combined. The square shape can be defined in data or display coordinates as determined by the ``use\_data\_coordinates`` argument specified when creating the selector.

drag\_from\_anywhere : bool, default: False  
If ``True``, the widget can be moved by clicking anywhere within its bounds.

ignore\_event\_outside : bool, default: False  
If ``True``, the event triggered outside the span selector will be ignored.

use\_data\_coordinates : bool, default: False  
If ``True``, the "square" shape of the selector is defined in data coordinates instead of display coordinates.

#### Examples

-----  
:doc:`gallery/widgets/rectangle\_selector`

## matplotlib.widgets.Lasso

```
Lasso(ax, xy, callback, *, useblit=True, props=None)
```

Selection curve of an arbitrary shape.

The selected path can be used in conjunction with  
`~matplotlib.path.Path.contains\_point` to select data points from an image.

Unlike `LassoSelector`, this must be initialized with a starting point \*xy\*, and the `Lasso` events are destroyed upon release.

#### Parameters

-----  
ax : `~matplotlib.axes.Axes`

The parent Axes for the widget.

xy : (float, float)

Coordinates of the start of the lasso.

callback : callable

Whenever the lasso is released, the \*callback\* function is called and passed the vertices of the selected path.

useblit : bool, default: True

Whether to use blitting for faster drawing (if supported by the backend). See the tutorial :ref:`blitting` for details.

props: dict, optional

Lasso line properties. See `Line2D` for valid properties.

Default \*props\* are::

```
{'linestyle' : '-', 'color' : 'black', 'lw' : 2}
```

.. versionadded:: 3.9

## matplotlib.widgets.LassoSelector

```
LassoSelector(ax, onselect=None, *, useblit=True, props=None, button=None)
```

Selection curve of an arbitrary shape.

For the selector to remain responsive you must keep a reference to it.

The selected path can be used in conjunction with `~.Path.contains_point` to select data points from an image.

In contrast to `Lasso`, `LassoSelector` is written with an interface similar to `RectangleSelector` and `SpanSelector`, and will continue to interact with the Axes until disconnected.

Example usage::

```
ax = plt.subplot()
ax.plot(x, y)

def onselect(verts):
 print(verts)
lasso = LassoSelector(ax, onselect)
```

### Parameters

-----

`ax` : `~matplotlib.axes.Axes`  
The parent Axes for the widget.

`onselect` : function, optional  
Whenever the lasso is released, the `*onselect*` function is called and passed the vertices of the selected path.

`useblit` : bool, default: True  
Whether to use blitting for faster drawing (if supported by the backend). See the tutorial :ref:`blitting` for details.

`props` : dict, optional  
Properties with which the line is drawn, see `~.Line2D` for valid properties. Default values are defined in ```mpl.rcParams```.

`button` : `~.MouseButton` or list of `~.MouseButton`, optional  
The mouse buttons used for rectangle selection. Default is ```None```, which corresponds to all buttons.

## matplotlib.widgets.Line2D

```
Line2D(xdata, ydata, *, linewidth=None, linestyle=None, color=None, gapcolor=None,
marker=None, markersize=None, markeredgewidth=None, markeredgewidth=None,
markerfacecolor=None, markerfacecoloralt='none', fillstyle=None, antialiased=None,
dash_capstyle=None, solid_capstyle=None, dash_joinstyle=None, solid_joinstyle=None,
pickradius=5, drawstyle=None, markevery=None, **kwargs)
```

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the

drawing of the solid line is influenced by the drawstyle, e.g., one can create "stepped" lines in various styles.

## matplotlib.widgets.LockDraw

```
LockDraw()
```

Some widgets, like the cursor, draw onto the canvas, and this is not desirable under all circumstances, like when the toolbar is in zoom-to-rect mode and drawing a rectangle. To avoid this, a widget can acquire a canvas' lock with `canvas.widgetlock(widget)` before drawing on the canvas; this will prevent other widgets from doing so at the same time (if they also try to acquire the lock first).

## matplotlib.widgets.MultiCursor

```
MultiCursor(canvas, axes, *, useblit=True, horizOn=False, vertOn=True, **lineprops)
```

Provide a vertical (default) and/or horizontal line cursor shared between multiple Axes.

For the cursor to remain responsive you must keep a reference to it.

### Parameters

-----

canvas : object

This parameter is entirely unused and only kept for back-compatibility.

axes : list of `~matplotlib.axes.Axes`

The `~.axes.Axes` to attach the cursor to.

useblit : bool, default: True

Use blitting for faster drawing if supported by the backend.

See the tutorial :ref:`blitting` for details.

horizOn : bool, default: False

Whether to draw the horizontal line.

vertOn : bool, default: True

Whether to draw the vertical line.

### Other Parameters

-----

**lineprops**

`~.Line2D` properties that control the appearance of the lines.

See also `~.Axes.axhline`.

### Examples

-----

See :doc:`/gallery/widgets/multicursor`.

## matplotlib.widgets.Polygon

```
Polygon(xy, *, closed=True, **kwargs)
```

A general polygon patch.

## matplotlib.widgets.PolygonSelector

```
PolygonSelector(ax, onselect=None, *, useblit=False, props=None, handle_props=None,
grab_range=10, draw_bounding_box=False, box_handle_props=None, box_props=None)
```

Select a polygon region of an Axes.

Place vertices with each mouse click, and make the selection by completing the polygon (clicking on the first vertex). Once drawn individual vertices can be moved by clicking and dragging with the left mouse button, or removed by clicking the right mouse button.

In addition, the following modifier keys can be used:

- Hold `*ctrl*` and click and drag a vertex to reposition it before the polygon has been completed.
- Hold the `*shift*` key and click and drag anywhere in the Axes to move all vertices.
- Press the `*esc*` key to start a new polygon.

For the selector to remain responsive you must keep a reference to it.

### Parameters

-----

`ax` : `~matplotlib.axes.Axes``

The parent Axes for the widget.

`onselect` : function, optional

When a polygon is completed or modified after completion, the `*onselect*` function is called and passed a list of the vertices as ``(xdata, ydata)`` tuples.

`useblit` : bool, default: False

Whether to use blitting for faster drawing (if supported by the backend). See the tutorial :ref:`blitting` for details.

`props` : dict, optional

Properties with which the line is drawn, see `~.Line2D`` for valid properties. Default::

```
dict(color='k', linestyle='-', linewidth=2, alpha=0.5)
```

`handle_props` : dict, optional

Artist properties for the markers drawn at the vertices of the polygon. See the marker arguments in `~.Line2D`` for valid properties. Default values are defined in ```mpl.rcParams``` except for the default value of ```markeredgecolor``` which will be the same as the ```color``` property in `*props*`.

`grab_range` : float, default: 10

A vertex is selected (to complete the polygon or to move a vertex) if

the mouse click is within `*grab_range*` pixels of the vertex.

`draw_bounding_box` : bool, optional

If ``True``, a bounding box will be drawn around the polygon selector once it is complete. This box can be used to move and resize the selector.

`box_handle_props` : dict, optional

Properties to set for the box handles. See the documentation for the `*handle_props*` argument to ``RectangleSelector`` for more info.

`box_props` : dict, optional

Properties to set for the box. See the documentation for the `*props*` argument to ``RectangleSelector`` for more info.

#### Examples

-----  
:doc:`/gallery/widgets/polygon\_selector\_simple`  
:doc:`/gallery/widgets/polygon\_selector\_demo`

#### Notes

-----  
If only one point remains after removing points, the selector reverts to an incomplete state and you can start drawing a new polygon from the existing point.

## matplotlib.widgets.RadioButton

```
RadioButton(ax, labels, active=0, activecolor=None, *, useblit=True,
 label_props=None, radio_props=None)
```

A GUI neutral radio button.

For the buttons to remain responsive you must keep a reference to this object.

Connect to the `RadioButton`s with the ``.on_clicked`` method.

#### Attributes

-----  
`ax` : `~matplotlib.axes.Axes``  
The parent Axes for the widget.  
`activecolor` : `:mpltype:`color``  
The color of the selected button.  
`labels` : list of ``.Text``  
The button labels.  
`value_selected` : str  
The label text of the currently selected button.  
`index_selected` : int  
The index of the selected button.

## matplotlib.widgets.RangeSlider

```
RangeSlider(ax, label, valmin, valmax, *, valinit=None, valfmt=None, closedmin=True,
closedmax=True, dragging=True, valstep=None, orientation='horizontal',
track_color='lightgrey', handle_style=None, **kwargs)
```

A slider representing a range of floating point values. Defines the min and max of the range via the *\*val\** attribute as a tuple of (min, max).

Create a slider that defines a range contained within [*\*valmin\**, *\*valmax\**] in Axes *\*ax\**. For the slider to remain responsive you must maintain a reference to it. Call `:meth:`on_changed`` to connect to the slider event.

Attributes

-----

*val* : tuple of float  
Slider value.

## matplotlib.widgets.Rectangle

```
Rectangle(xy, width, height, *, angle=0.0, rotation_point='xy', **kwargs)
```

A rectangle defined via an anchor point *\*xy\** and its *\*width\** and *\*height\**.

The rectangle extends from ```xy[0]``` to ```xy[0] + width``` in x-direction and from ```xy[1]``` to ```xy[1] + height``` in y-direction. ::

```
: +-----+
: |
: height |
: |
: (xy)--- width ----+
```

One may picture *\*xy\** as the bottom left corner, but which corner *\*xy\** is actually depends on the direction of the axis and the sign of *\*width\** and *\*height\**; e.g. *\*xy\** would be the bottom right corner if the x-axis was inverted or if *\*width\** was negative.

## matplotlib.widgets.RectangleSelector

```
RectangleSelector(ax, onselect=None, *, minspanx=0, minspany=0, useblit=False,
props=None, spancoords='data', button=None, grab_range=10, handle_props=None,
interactive=False, state_modifier_keys=None, drag_from_anywhere=False,
ignore_event_outside=False, use_data_coordinates=False)
```

Select a rectangular region of an Axes.

For the cursor to remain responsive you must keep a reference to it.

Press and release events triggered at the same coordinates outside the selection will clear the selector, except when ```ignore_event_outside=True```.

Parameters

-----

*ax* : `~matplotlib.axes.Axes``  
The parent Axes for the widget.

onselect : function, optional

A callback function that is called after a release event and the selection is created, changed or removed.

It must have the signature::

```
def onselect(eclick: MouseEvent, erelease: MouseEvent)
```

where *\*eclick\** and *\*erelease\** are the mouse click and release

``MouseEvent`s` that start and complete the selection.

minspanx : float, default: 0

Selections with an x-span less than or equal to *\*minspanx\** are removed (when already existing) or cancelled.

minspany : float, default: 0

Selections with an y-span less than or equal to *\*minspany\** are removed (when already existing) or cancelled.

useblit : bool, default: False

Whether to use blitting for faster drawing (if supported by the backend). See the tutorial :ref:`blitting` for details.

props : dict, optional

Properties with which the rectangle is drawn. See ``Patch`` for valid properties.

Default:

```
``dict(facecolor='red', edgecolor='black', alpha=0.2, fill=True)``
```

spancoords : {"data", "pixels"}, default: "data"

Whether to interpret *\*minspanx\** and *\*minspany\** in data or in pixel coordinates.

button : ``MouseButton``, list of ``MouseButton``, default: all buttons  
Button(s) that trigger rectangle selection.

grab\_range : float, default: 10

Distance in pixels within which the interactive tool handles can be activated.

handle\_props : dict, optional

Properties with which the interactive handles (marker artists) are drawn. See the marker arguments in ``Line2D`` for valid properties. Default values are defined in ```mpl.rcParams``` except for the default value of ```markeredgecolor``` which will be the same as the ```edgecolor``` property in *\*props\**.

interactive : bool, default: False

Whether to draw a set of handles that allow interaction with the widget after it is drawn.

state\_modifier\_keys : dict, optional

Keyboard modifiers which affect the widget's behavior. Values

amend the defaults, which are:

- "move": Move the existing shape, default: no modifier.
- "clear": Clear the current shape, default: "escape".
- "square": Make the shape square, default: "shift".
- "center": change the shape around its center, default: "ctrl".
- "rotate": Rotate the shape around its center between -45° and 45°, default: "r".

"square" and "center" can be combined. The square shape can be defined in data or display coordinates as determined by the ``use\_data\_coordinates`` argument specified when creating the selector.

drag\_from\_anywhere : bool, default: False  
If ``True``, the widget can be moved by clicking anywhere within its bounds.

ignore\_event\_outside : bool, default: False  
If ``True``, the event triggered outside the span selector will be ignored.

use\_data\_coordinates : bool, default: False  
If ``True``, the "square" shape of the selector is defined in data coordinates instead of display coordinates.

#### Examples

```

>>> import matplotlib.pyplot as plt
>>> import matplotlib.widgets as mwidgets
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3], [10, 50, 100])
>>> def onselect(eclick, erelease):
... print(eclick.xdata, eclick.ydata)
... print(erelease.xdata, erelease.ydata)
>>> props = dict(facecolor='blue', alpha=0.5)
>>> rect = mwidgets.RectangleSelector(ax, onselect, interactive=True,
... props=props)
>>> fig.show()
>>> rect.add_state('square')
```

See also: `:doc:`/gallery/widgets/rectangle_selector``

## matplotlib.widgets.Slider

```
Slider(ax, label, valmin, valmax, *, valinit=0.5, valfmt=None, closedmin=True,
closedmax=True, slidermin=None, slidermax=None, dragging=True, valstep=None,
orientation='horizontal', initcolor='r', track_color='lightgrey', handle_style=None,
**kwargs)
```

A slider representing a floating point range.

Create a slider from `*valmin*` to `*valmax*` in Axes `*ax*`. For the slider to remain responsive you must maintain a reference to it. Call `:meth:`on_changed`` to connect to the slider event.

#### Attributes

-----

val : float  
Slider value.

## matplotlib.widgets.SliderBase

```
SliderBase(ax, orientation, closedmin, closedmax, valmin, valmax, valfmt, dragging, valstep)
```

The base class for constructing Slider widgets. Not intended for direct usage.

For the slider to remain responsive you must maintain a reference to it.

## matplotlib.widgets.SpanSelector

```
SpanSelector(ax, onselect, direction, *, minspan=0, useblit=False, props=None, onmove_callback=None, interactive=False, button=None, handle_props=None, grab_range=10, state_modifier_keys=None, drag_from_anywhere=False, ignore_event_outside=False, snap_values=None)
```

Visually select a min/max range on a single axis and call a function with those values.

To guarantee that the selector remains responsive, keep a reference to it.

In order to turn off the SpanSelector, set ``span\_selector.active`` to False. To turn it back on, set it to True.

Press and release events triggered at the same coordinates outside the selection will clear the selector, except when ``ignore\_event\_outside=True``.

#### Parameters

-----

ax : ~matplotlib.axes.Axes`

onselect : callable with signature ``func(min: float, max: float)``

A callback function that is called after a release event and the selection is created, changed or removed.

direction : {"horizontal", "vertical"}

The direction along which to draw the span selector.

minspan : float, default: 0

If selection is less than or equal to \*minspan\*, the selection is removed (when already existing) or cancelled.

useblit : bool, default: False

If True, use the backend-dependent blitting features for faster canvas updates. See the tutorial :ref:`blitting` for details.

props : dict, default: {'facecolor': 'red', 'alpha': 0.5}

Dictionary of `.Patch`` properties.

`onmove_callback` : callable with signature ``func(min: float, max: float)``, optional  
Called on mouse move while the span is being selected.

`interactive` : bool, default: False  
Whether to draw a set of handles that allow interaction with the widget after it is drawn.

`button` : `.MouseButton`` or list of `.MouseButton``, default: all buttons  
The mouse buttons which activate the span selector.

`handle_props` : dict, default: None  
Properties of the handle lines at the edges of the span. Only used when `*interactive*` is True. See `.Line2D`` for valid properties.

`grab_range` : float, default: 10  
Distance in pixels within which the interactive tool handles can be activated.

`state_modifier_keys` : dict, optional  
Keyboard modifiers which affect the widget's behavior. Values amend the defaults, which are:

- "clear": Clear the current shape, default: "escape".

`drag_from_anywhere` : bool, default: False  
If ``True``, the widget can be moved by clicking anywhere within its bounds.

`ignore_event_outside` : bool, default: False  
If ``True``, the event triggered outside the span selector will be ignored.

`snap_values` : 1D array-like, optional  
Snap the selector edges to the given values.

#### Examples

```

>>> import matplotlib.pyplot as plt
>>> import matplotlib.widgets as mwidgets
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3], [10, 50, 100])
>>> def onselect(vmin, vmax):
... print(vmin, vmax)
>>> span = mwidgets.SpanSelector(ax, onselect, 'horizontal',
... props=dict(facecolor='blue', alpha=0.5))
>>> fig.show()
```

See also: `:doc:`/gallery/widgets/span_selector``

## matplotlib.widgets.SubplotTool

```
SubplotTool(targetfig, toolfig)
```

A tool to adjust the subplot params of a `.Figure``.

## matplotlib.widgets.TextBox

```
TextBox(ax, label, initial='', *, color='.95', hovercolor='1', label_pad=0.01,
 textalignment='left')
```

A GUI neutral text input box.

For the text box to remain responsive you must keep a reference to it.

Call `.on_text_change` to be updated whenever the text changes.

Call `.on_submit` to be updated whenever the user hits enter or leaves the text entry field.

### Attributes

-----

`ax` : `~matplotlib.axes.Axes`

The parent Axes for the widget.

`label` : `~matplotlib.text.Text`

`color` : `:mpltype:`color``

The color of the text box when not hovering.

`hovercolor` : `:mpltype:`color``

The color of the text box when hovering.

## matplotlib.widgets.ToolHandles

```
ToolHandles(ax, x, y, *, marker='o', marker_props=None, useblit=True)
```

Control handles for canvas tools.

### Parameters

-----

`ax` : `~matplotlib.axes.Axes`

Matplotlib Axes where tool handles are displayed.

`x, y` : 1D arrays

Coordinates of control handles.

`marker` : str, default: 'o'

Shape of marker used to display handle. See `~.pyplot.plot`.

`marker_props` : dict, optional

Additional marker properties. See `.Line2D`.

`useblit` : bool, default: True

Whether to use blitting for faster drawing (if supported by the backend). See the tutorial :ref:`blitting` for details.

## matplotlib.widgets.ToolLineHandles

```
ToolLineHandles(ax, positions, direction, *, line_props=None, useblit=True)
```

Control handles for canvas tools.

### Parameters

-----

`ax` : `~matplotlib.axes.Axes`

Matplotlib Axes where tool handles are displayed.

positions : 1D array  
Positions of handles in data coordinates.  
direction : {"horizontal", "vertical"}  
Direction of handles, either 'vertical' or 'horizontal'  
line\_props : dict, optional  
Additional line properties. See `Line2D`.  
useblit : bool, default: True  
Whether to use blitting for faster drawing (if supported by the backend). See the tutorial :ref:`blitting` for details.

## matplotlib.widgets.TransformedPatchPath

`TransformedPatchPath(patch)`

A `TransformedPatchPath` caches a non-affine transformed copy of the `~.patches.Patch`. This cached copy is automatically updated when the non-affine part of the transform or the patch changes.

## matplotlib.widgets.Widget

`Widget()`

Abstract base class for GUI neutral widgets.