

MARKDOWN SYNTAX



Markdown is a way to style text on the web. You control the display of the document; formatting words as bold or italic, adding images, and creating lists are just a few of the things we can do with Markdown. Mostly, Markdown is just regular text with a few non-alphabetic characters thrown in, like `#` or `*`.

HEADERS

```
# This is an <h1> tag
## This is an <h2> tag
##### This is an <h6> tag
```

EMPHASIS

```
*This text will be italic*
_This will also be italic_

**This text will be bold**
__This will also be bold__

*You can combine them*
```

BLOCKQUOTES

As Grace Hopper said:

```
> I've always been more interested
> in the future than in the past.
```

As Grace Hopper said:

```
| I've always been more interested
| in the future than in the past.
```

LISTS

Unordered

```
* Item 1
* Item 2
  * Item 2a
  * Item 2b
```

Ordered

```
1. Item 1
2. Item 2
3. Item 3
  * Item 3a
  * Item 3b
```

IMAGES

```
![GitHub Logo](/images/logo.png)
```

Format: `![Alt Text](url)`

LINKS

```
http://github.com - automatic!
```

```
[GitHub](http://github.com)
```

BACKSLASH ESCAPES

Markdown allows you to use backslash escapes to generate literal characters which would otherwise have special meaning in Markdown's formatting syntax.

```
\*literal asterisks\*
```

```
*literal asterisks*
```

Markdown provides backslash escapes for the following characters:

<code>\</code> backslash	<code>()</code> parentheses
<code>`</code> backtick	<code>#</code> hash mark
<code>*</code> asterisk	<code>+</code> plus sign
<code>_</code> underscore	<code>-</code> minus sign (hyphen)
<code>{}</code> curly braces	<code>.</code> dot
<code>[]</code> square brackets	<code>!</code> exclamation mark

GITHUB FLAVORED MARKDOWN



GitHub.com uses its own version of the Markdown syntax, GFM, that provides an additional set of useful features, many of which make it easier to work with content on GitHub.com.

USERNAME @MENTIONS

Typing an `@` symbol, followed by a username, will notify that person to come and view the comment. This is called an “@mention”, because you’re mentioning the individual. You can also @mention teams within an organization.

ISSUE REFERENCES

Any number that refers to an Issue or Pull Request will be automatically converted into a link.

```
#1
github-flavored-markdown#1
defunkt/github-flavored-markdown#1
```

EMOJI

To see a list of every image we support, check out www.emoji-cheat-sheet.com

```
GitHub supports emoji!
:+1: :sparkles: :camel: :tada:
:rocket: :metal: :octocat:
```

GitHub supports emoji!



FENCED CODE BLOCKS

Markdown converts text with four leading spaces into a code block; with GFM you can wrap your code with ````` to create a code block without the leading spaces. Add an optional language identifier and your code will get syntax highlighting.

```
```javascript
function test() {
 console.log("look ma', no spaces");
}
```
```

```
function test() {
  console.log("look ma', no spaces");
}
```

TASK LISTS

- [x] this is a complete item
- [] this is an incomplete item
- [x] @mentions, #refs, [links](), **formatting**, and tags supported
- [x] list syntax required (any unordered or ordered list supported)

- ☒ this is a complete item
- ☐ this is an incomplete item
- ☒ @mentions, #refs, links, formatting, and tags supported
- ☒ list syntax required (any unordered or ordered list supported)

TABLES

You can create tables by assembling a list of words and dividing them with hyphens `-` (for the first row), and then separating each column with a pipe `|`:

```
First Header	Second Header
Content cell 1 | Content cell 2
Content column 1 | Content column 2
```

| First Header | Second Header |
|------------------|------------------|
| Content cell 1 | Content cell 2 |
| Content column 1 | Content column 2 |

Python For Data Science Cheat Sheet

Matplotlib

Learn Python Interactively at [www.DataCamp.com](https://www.datacamp.com)



Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.



1 Prepare The Data

Also see Lists & NumPy

1D Data

```
>>> import numpy as np
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))
>>> data2 = 3 * np.random.random((10, 10))
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample_data
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

2 Create Plot

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

Figure

```
>>> fig = plt.figure()
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

Axes

All plotting is done with respect to an Axes. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()
>>> ax1 = fig.add_subplot(221) # row-col-num
>>> ax3 = fig.add_subplot(212)
>>> fig3, axes = plt.subplots(nrows=2,ncols=2)
>>> fig4, axes2 = plt.subplots(ncols=3)
```

3 Plotting Routines

1D Data

| | |
|--|--|
| <pre>>>> lines = ax.plot(x,y) >>> ax.scatter(x,y) >>> axes[0,0].bar([1,2,3],[3,4,5]) >>> axes[1,0].barh([0.5,1,2.5],[0,1,2]) >>> axes[1,1].axhline(0.45) >>> axes[0,1].axvline(0.65) >>> ax.fill(x,y,color='blue') >>> ax.fill_between(x,y,color='yellow')</pre> | <p>Draw points with lines or markers connecting them</p> <p>Draw unconnected points, scaled or colored</p> <p>Plot vertical rectangles (constant width)</p> <p>Plot horizontal rectangles (constant height)</p> <p>Draw a horizontal line across axes</p> <p>Draw a vertical line across axes</p> <p>Draw filled polygons</p> <p>Fill between y-values and 0</p> |
|--|--|

2D Data or Images

| | |
|--|---------------------------|
| <pre>>>> fig, ax = plt.subplots() >>> im = ax.imshow(img, cmap='gist_earth', interpolation='nearest', vmin=-2, vmax=2)</pre> | Colormapped or RGB arrays |
|--|---------------------------|

Vector Fields

| | |
|--|---|
| <pre>>>> axes[0,1].arrow(0,0,0.5,0.5) >>> axes[1,1].quiver(y,z) >>> axes[0,1].streamplot(X,Y,U,V)</pre> | <p>Add an arrow to the axes</p> <p>Plot a 2D field of arrows</p> <p>Plot 2D vector fields</p> |
|--|---|

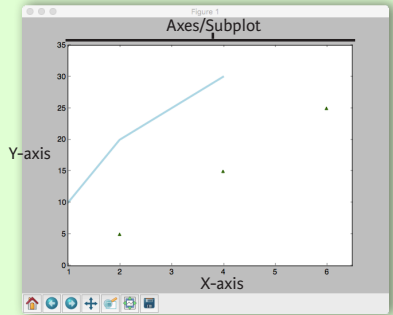
Data Distributions

| | |
|--|--|
| <pre>>>> ax1.hist(y) >>> ax3.boxplot(y) >>> ax3.violinplot(z)</pre> | <p>Plot a histogram</p> <p>Make a box and whisker plot</p> <p>Make a violin plot</p> |
|--|--|

| | |
|--|--|
| <pre>>>> axes2[0].pcolor(data2) >>> axes2[0].pcolormesh(data) >>> CS = plt.contour(Y,X,U) >>> axes2[2].contourf(data1) >>> axes2[2] = ax.clabel(CS)</pre> | <p>Pseudocolor plot of 2D array</p> <p>Pseudocolor plot of 2D array</p> <p>Plot contours</p> <p>Plot filled contours</p> <p>Label a contour plot</p> |
|--|--|

Plot Anatomy & Workflow

Plot Anatomy



Workflow

The basic steps to creating plots with matplotlib are:

- 1 Prepare data
- 2 Create plot
- 3 Plot
- 4 Customize plot
- 5 Save plot
- 6 Show plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4]
>>> y = [10,20,25,30]
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x, y, color='lightblue', linewidth=3)
>>> ax.scatter([2,4,6],
    [5,15,25],
    color='darkgreen',
    marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png')
>>> plt.show()
```

4 Customize Plot

Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
    cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,y,'--',x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

Text & Annotations

```
>>> ax.text(1,
    -2.1,
    'Example Graph',
    style='italic')
>>> ax.annotate("Sine",
    xy=(8, 0),
    xycoords='data',
    xytext=(10.5, 0),
    textcoords='data',
    arrowprops=dict(arrowstyle="->",
        connectionstyle="arc3"),)
```

Mathtext

```
>>> plt.title(r'$\sigma_i=15$', fontsize=20)
```

Limits, Legends & Layouts

Limits & Autoscaling

```
>>> ax.margins(x=0.0,y=0.1)
>>> ax.axis('equal')
>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5])
>>> ax.set_xlim(0,10.5)
```

Legends

```
>>> ax.set(title='An Example Axes',
    ylabel='Y-Axis',
    xlabel='X-Axis')
>>> ax.legend(loc='best')
```

Ticks

```
>>> ax.xaxis.set(ticks=range(1,5),
    ticklabels=[3,100,-12,"foo"])
>>> ax.tick_params(axis='y',
    direction='inout',
    length=10)
```

Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5,
    hspace=0.3,
    left=0.125,
    right=0.9,
    top=0.9,
    bottom=0.1)
```

```
>>> fig.tight_layout()
```

Axis Spines

```
>>> ax1.spines['top'].set_visible(False)
>>> ax1.spines['bottom'].set_position(('outward',10))
```

Add padding to a plot

Set the aspect ratio of the plot to 1

Set limits for x-and y-axis

Set limits for x-axis

Set a title and x-and y-axis labels

No overlapping plot elements

Manually set x-ticks

Make y-ticks longer and go in and out

Adjust the spacing between subplots

Fit subplot(s) in to the figure area

Make the top axis line for a plot invisible

Move the bottom axis line outward

5 Save Plot

Save figures

```
>>> plt.savefig('foo.png')
```

Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

6 Show Plot

```
>>> plt.show()
```

Close & Clear

```
>>> plt.cla()
>>> plt.clf()
>>> plt.close()
```

Clear an axis

Clear the entire figure

Close a window



Matplotlib for beginners

Matplotlib is a library for making 2D plots in Python. It is designed with the philosophy that you should be able to create simple plots with just a few commands:

1 Initialize

```
import numpy as np
import matplotlib.pyplot as plt
```

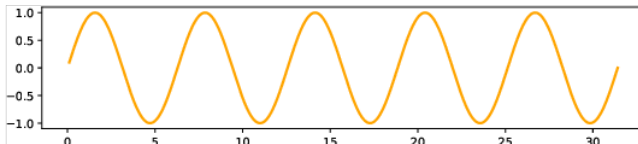
2 Prepare

```
X = np.linspace(0, 4*np.pi, 1000)
Y = np.sin(X)
```

3 Render

```
fig, ax = plt.subplots()
ax.plot(X, Y)
fig.show()
```

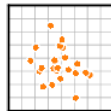
4 Observe



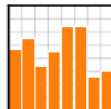
Choose

Matplotlib offers several kind of plots (see Gallery):

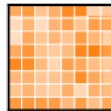
```
X = np.random.uniform(0, 1, 100)
Y = np.random.uniform(0, 1, 100)
ax.scatter(X, Y)
```



```
X = np.arange(10)
Y = np.random.uniform(1, 10, 10)
ax.bar(X, Y)
```



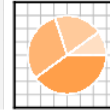
```
Z = np.random.uniform(0, 1, (8,8))
ax.imshow(Z)
```



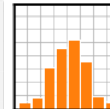
```
Z = np.random.uniform(0, 1, (8,8))
ax.contourf(Z)
```



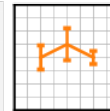
```
Z = np.random.uniform(0, 1, 4)
ax.pie(Z)
```



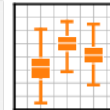
```
Z = np.random.normal(0, 1, 100)
ax.hist(Z)
```



```
X = np.arange(5)
Y = np.random.uniform(0, 1, 5)
ax.errorbar(X, Y, Y/4)
```



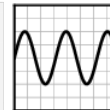
```
Z = np.random.normal(0, 1, (100,3))
ax.boxplot(Z)
```



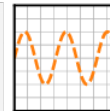
Tweak

You can modify pretty much anything in a plot, including limits, colors, markers, line width and styles, ticks and ticks labels, titles, etc.

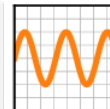
```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, color="black")
```



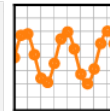
```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, linestyle="--")
```



```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, linewidth=5)
```



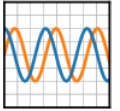
```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, marker="o")
```



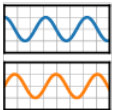
Organize

You can plot several data on the the same figure, but you can also split a figure in several subplots (named Axes):

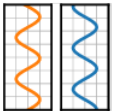
```
X = np.linspace(0, 10, 100)
Y1, Y2 = np.sin(X), np.cos(X)
ax.plot(X, Y1, X, Y2)
```



```
fig, (ax1, ax2) = plt.subplots((2,1))
ax1.plot(X, Y1, color="C1")
ax2.plot(X, Y2, color="C0")
```

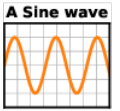


```
fig, (ax1, ax2) = plt.subplots((1,2))
ax1.plot(Y1, X, color="C1")
ax2.plot(Y2, X, color="C0")
```

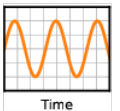


Label (everything)

```
ax.plot(X, Y)
fig.suptitle(None)
ax.set_title("A Sine wave")
```



```
ax.plot(X, Y)
ax.set_ylabel(None)
ax.set_xlabel("Time")
```



Explore

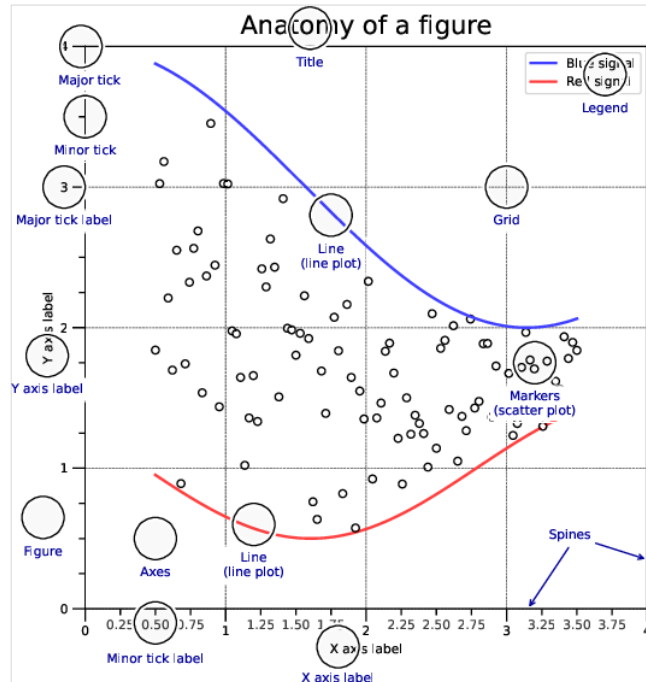
Figures are shown with a graphical user interface that allows to zoom and pan the figure, to navigate between the different views and to show the value under the mouse.

Save (bitmap or vector format)

```
fig.savefig("my-first-figure.png", dpi=300)
fig.savefig("my-first-figure.pdf")
```


Matplotlib for intermediate users

A matplotlib figure is composed of a hierarchy of elements that forms the actual figure. Each element can be modified.



Figure, axes & spines

```
fig, axs = plt.subplots((3,3))
axs[0,0].set_facecolor("#dddddff")
axs[2,2].set_facecolor("#ffffdd")
```



```
gs = fig.add_gridspec(3, 3)
ax = fig.add_subplot(gs[0, :])
ax.set_facecolor("#dddddff")
```

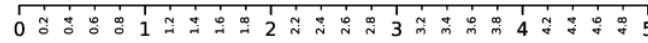


```
fig, ax = plt.subplots()
ax.spines["top"].set_color("None")
ax.spines["right"].set_color("None")
```



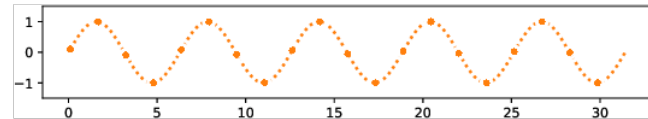
Ticks & labels

```
from mpl.ticker import MultipleLocator as ML
from mpl.ticker import ScalarFormatter as SF
ax.xaxis.set_minor_locator(ML(0.2))
ax.xaxis.set_minor_formatter(SF())
ax.tick_params(axis='x', which='minor', rotation=90)
```



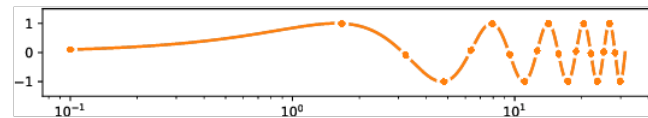
Lines & markers

```
X = np.linspace(0.1, 10*np.pi, 1000)
Y = np.sin(X)
ax.plot(X, Y, "C1o:", markevery=25, mec="1.0")
```



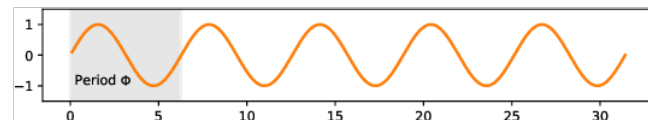
Scales & projections

```
fig, ax = plt.subplots()
ax.set_xscale("log")
ax.plot(X, Y, "C1o-", markevery=25, mec="1.0")
```



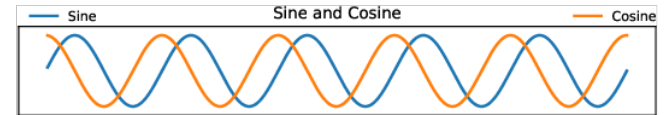
Text & ornaments

```
ax.fill_betweenx([-1,1],[0],[2*np.pi])
ax.text(0, -1, r"Period $\Phi$")
```



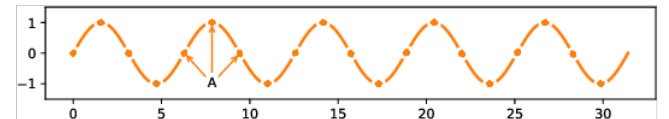
Legend

```
ax.plot(X, np.sin(X), "C0", label="Sine")
ax.plot(X, np.cos(X), "C1", label="Cosine")
ax.legend(bbox_to_anchor=(0,1,1,.1), ncol=2,
         mode="expand", loc="lower left")
```



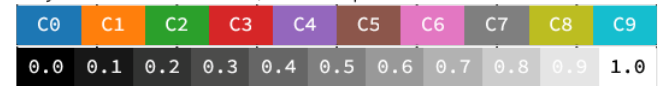
Annotation

```
ax.annotate("A", (X[250],Y[250]),(X[250],-1),
          ha="center", va="center",arrowprops =
          {"arrowstyle": "->", "color": "C1"})
```



Colors

Any color can be used, but Matplotlib offers sets of colors:



Size & DPI

Consider a square figure to be included in a two-columns A4 paper with 2cm margins on each side and a column separation of 1cm. The width of a figure is $(21 - 2 \times 2 - 1) / 2 = 8$ cm. One inch being 2.54cm, figure size should be 3.15x3.15 in.

```
fig = plt.figure(figsize=(3.15,3.15), dpi=50)
plt.savefig("figure.pdf", dpi=600)
```

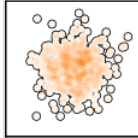
Matplotlib 3.4.2 handout for intermediate users. Copyright (c) 2021 Matplotlib Development Team. Released under a CC-BY 4.0 International License. Supported by NumFOCUS.

Matplotlib tips & tricks

Transparency

Scatter plots can be enhanced by using transparency (alpha) in order to show area with higher density. Multiple scatter plots can be used to delineate a frontier.

```
X = np.random.normal(-1, 1, 500)
Y = np.random.normal(-1, 1, 500)
ax.scatter(X, Y, 50, "0.0", lw=2) # optional
ax.scatter(X, Y, 50, "1.0", lw=0) # optional
ax.scatter(X, Y, 40, "C1", lw=0, alpha=0.1)
```



Rasterization

If your figure has many graphical elements, such as a huge scatter, you can rasterize them to save memory and keep other elements in vector format.

```
X = np.random.normal(-1, 1, 10_000)
Y = np.random.normal(-1, 1, 10_000)
ax.scatter(X, Y, rasterized=True)
fig.savefig("rasterized-figure.pdf", dpi=600)
```

Offline rendering

Use the Agg backend to render a figure directly in an array.

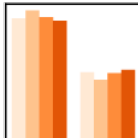
```
from matplotlib.backends.backend_agg import FigureCanvas
canvas = FigureCanvas(Figure())
... # draw som stuff
canvas.draw()
Z = np.array(canvas.renderer.buffer_rgba())
```

Range of continuous colors

You can use colormap to pick from a range of continuous colors.

```
X = np.random.randn(1000, 4)
cmap = plt.get_cmap("Oranges")
colors = cmap([0.2, 0.4, 0.6, 0.8])

ax.hist(X, 2, histtype='bar', color=colors)
```



Text outline

Use text outline to make text more visible.

```
import matplotlib.path_effects as fx
text = ax.text(0.5, 0.1, "Label")
text.set_path_effects([
    fx.Stroke(linewidth=3, foreground='1.0'),
    fx.Normal()])
```



Multiline plot

You can plot several lines at once using None as separator.

```
X, Y = [], []
for x in np.linspace(0, 10*np.pi, 100):
    X.extend([x, x, None]), Y.extend([0, sin(x), None])
ax.plot(X, Y, "black")
```



Dotted lines

To have rounded dotted lines, use a custom lines tyle and modify dash_capstyle.

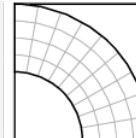
```
ax.plot([0,1], [0,0], "C1",
        linestyle = (0, (0.01, 1)), dash_capstyle="round")
ax.plot([0,1], [1,1], "C1",
        linestyle = (0, (0.01, 2)), dash_capstyle="round")
```



Combining axes

You can use overlaid axes with different projections.

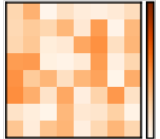
```
ax1 = fig.add_axes([0,0,1,1],
                    label="cartesian")
ax2 = fig.add_axes([0,0,1,1],
                    label="polar",
                    projection="polar")
```



Colorbar adjustment

You can adjust a colorbar's size when adding it.

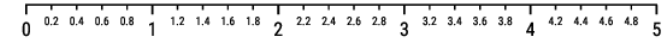
```
im = ax.imshow(Z)
cb = plt.colorbar(im,
                  fraction=0.046, pad=0.04)
cb.set_ticks([])
```



Taking advantage of typography

You can use a condensed font such as Roboto Condensed to save space on tick labels.

```
for tick in ax.get_xticklabels(which='both'):
    tick.set_fontname("Roboto Condensed")
```



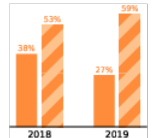
Getting rid of margins

Once your figure is finished, you can call `tight_layout()` to remove white margins. If there are remaining margins, you can use the `pdfcrop` utility (comes with TeX live).

Hatching

You can achieve a nice visual effect with thick hatch patterns.

```
cmap = plt.get_cmap("Oranges")
plt.rcParams['hatch.color'] = cmap(0.2)
plt.rcParams['hatch.linewidth'] = 8
ax.bar(X, Y, color=cmap(0.6), hatch="/" )
```



Read the documentation

Matplotlib comes with an extensive documentation explaining the details of each command and is generally accompanied by examples. Together with the huge online gallery, this documentation is a gold-mine.

Matplotlib 3.4.2 handout for tips & tricks. Copyright (c) 2021 Matplotlib Development Team. Released under a CC-BY 4.0 International License. Supported by NumFOCUS.

Quick start

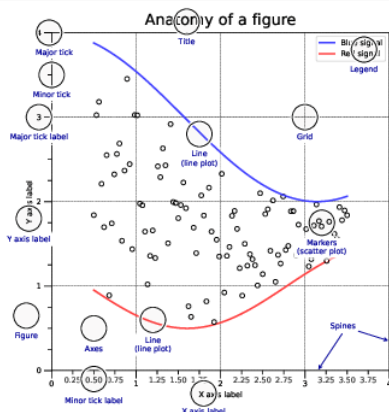
```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

```
X = np.linspace(0, 2*np.pi, 100)
Y = np.cos(X)
```

```
fig, ax = plt.subplots()
ax.plot(X, Y, color='C1')
```

```
fig.savefig("figure.pdf")
fig.show()
```

Anatomy of a figure



Subplots layout

```
fig, axes = plt.subplots(3,3)

G = gridspec(rows,cols,...)
ax = G[0,:]
```

```
ax.inset_axes(extent)
```

```
d=make_axes_locatable(ax)
ax=d.new_horizontal('10%')
```

Getting help

matplotlib.org
github.com/matplotlib/matplotlib/issues
discourse.matplotlib.org
stackoverflow.com/questions/tagged/matplotlib
gitter.im/matplotlib
twitter.com/matplotlib
[Matplotlib users mailing list](https://matplotlib.org/users)

Basic plots

```
plt.plot([X], Y, [fmt], ...)
X, Y, fmt, color, marker, linestyle
```

```
scatter(X, Y, ...)
X, Y, [s]izes, [c]olors, marker, cmap
```

```
bar[h](x,height,...)
x, height, width, bottom, align, color
```

```
imshow(Z, [cmap], ...)
Z, cmap, interpolation, extent, origin
```

```
contour[f]([X], [Y], Z,...)
X, Y, Z, levels, colors, extent, origin
```

```
quiver([X], [Y], U, V,...)
X, Y, U, V, C, units, angles
```

```
pie(X, [explode], ...)
Z, explode, labels, colors, radius
```

```
text(x,y,text,...)
x, y, text, va, ha, size, weight, transform
```

```
fill[_between][x]( ... )
X, Y1, Y2, color, where
```

Advanced plots

```
step(X, Y, [fmt], ...)
X, Y, fmt, color, marker, where
```

```
boxplot(X,...)
X, notch, sym, bootstrap, widths
```

```
errorbar(X, Y, xerr, yerr, ...)
X, Y, xerr, yerr, fmt
```

```
hist(X, bins, ...)
X, bins, range, density, weights
```

```
violinplot(D,...)
D, positions, widths, vert
```

```
barbs([X], [Y], U, V, ...)
X, Y, U, V, C, length, pivot, sizes
```

```
eventplot(positions,...)
positions, orientation, lineoffsets
```

```
hexbin(X, Y, C,...)
X, Y, C, gridsizes, bins
```

```
xcorr(X, Y,...)
X, Y, normed, detrend
```

Scales

```
ax.set_[xy]scale(scale,...)
ax.[xy]axis.set_[minor|major]_locator(locator)
```

linear any values

symlog any values

log values > 0

logit 0 < values < 1

Projections

```
subplot(..., projection=p)
p='polar'
```

p='3d'

p=Orthographic()

from cartopy.crs import Cartographic

Lines

```
linestyle or ls
```

```
capstyle or dash_capstyle
```

"butt" "round" "projecting"

Markers

marker every

10 [0, -1] (25, 5) [0, 25, -1]

Colors

```
C0 C1 C2 C3 C4 C5 C6 C7 C8 C9
```

Darkred Firebrick Crimson Indigo Saffron

(1,0,0) (1,0,0,0.75) (1,0,0,0.5) (1,0,0,0.25)

#FF0000 #FF0000 #FF0000 #FF0000

0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

'Cn'

'x'

'name'

(R,G,B,[A])

'#RRGGBB[AA]'

'x.y'

Colormaps

```
plt.get_cmap(name)
```

Uniform

viridis magma plasma

Sequential

Greys YlOrBr Wistia

Diverging

Spectral coolwarm RdGy

Qualitative

tab10 tab20

Cyclic

twilight

Tick locators

```
from matplotlib import ticker
ax.[xy]axis.set_[minor|major]_locator(locator)
```

NullLocator()

MultipleLocator(8.5)

FixedLocator([0, 1, 5])

LinearLocator(numticks=3)

IndexLocator(base=0.5, offset=0.25)

AutoLocator()

MaxNLocator(n=4)

LogLocator(base=10, numticks=15)

Tick formatters

```
from matplotlib import ticker
ax.[xy]axis.set_[minor|major]_formatter(formatter)
```

NullFormatter()

FixedFormatter(['', '0', '1', ...])

FuncFormatter(lambda x, pos: "[%2f]" % x)

FormatStrFormatter("%d")

ScalarFormatter()

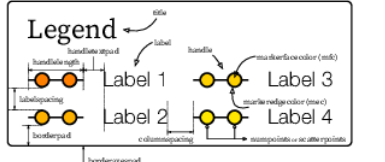
StrMethodFormatter('%x')

PercentFormatter(xmax=5)

Ornaments

```
ax.legend(...)
```

handles, labels, loc, title, frameon



```
ax.colorbar(...)
```

mappable, ax, cax, orientation

```
ax.annotate(...)
```

text, xy, xytext, xycoords, textcoords, arrowprops

Event handling

```
fig, ax = plt.subplots()
def on_click(event):
    print(event)
fig.canvas.mpl_connect(
    'button_press_event', on_click)
```

Animation

```
import matplotlib.animation as mpla
```

T = np.linspace(0, 2*np.pi, 100)

S = np.sin(T)

line, = plt.plot(T, S)

def animate(i):

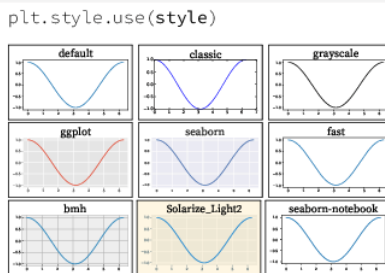
line.set_ydata(np.sin(T+i/50))

anim = mpla.FuncAnimation(

plt.gcf(), animate, interval=5)

plt.show()

Styles



Quick reminder

```
ax.grid()
```

ax.patch.set_alpha(0)

ax.set_[xy]lim(vmin, vmax)

ax.set_[xy]label(label)

ax.set_[xy]ticks(list)

ax.set_[xy]ticklabels(list)

ax.set_[sup]title(title)

ax.tick_params(width=10, ...)

ax.set_axis_[on|off]()

fig.tight_layout()

plt.gcf(), plt.gca()

mpl.rc('axes', linewidth=1, ...)

fig.patch.set_alpha(0)

text=r'\$\frac{-e^{i\pi}}{2^n}\$'

Keyboard shortcuts

ctrl + **s** Save
ctrl + **w** Close plot
r Reset view
f Fullscreen 0/1
f View forward
b View back
p Pan view
o Zoom to rect
x X pan/zoom
y Y pan/zoom
g Minor grid 0/1
G Major grid 0/1
l X axis log/linear
L Y axis log/linear

Ten simple rules

1. Know Your Audience
2. Identify Your Message
3. Adapt the Figure
4. Captions Are Not Optional
5. Do Not Trust the Defaults
6. Use Color Effectively
7. Do Not Misdread the Reader
8. Avoid "Chartjunk"
9. Message Trumps Beauty
10. Get the Right Tool



Python For Data Science



Matplotlib Cheat Sheet

Learn Matplotlib online at www.DataCamp.com

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.

> Prepare The Data

1D Data

```
>>> import numpy as np
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))
>>> data2 = 3 * np.random.random((10, 10))
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample_data
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

> Create Plot

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

Figure

```
>>> fig = plt.figure()
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

Axes

All plotting is done with respect to an Axes. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()
>>> ax1 = fig.add_subplot(221) #row-col-num
>>> ax3 = fig.add_subplot(212)
>>> fig3, axes = plt.subplots(nrows=2,ncols=2)
>>> fig4, axes2 = plt.subplots(ncols=3)
```

> Save Plot

```
>>> plt.savefig('foo.png') #Save figures
>>> plt.savefig('foo.png', transparent=True) #Save transparent figures
```

> Show Plot

```
>>> plt.show()
```

> Plotting Routines

1D Data

```
>>> fig, ax = plt.subplots()
>>> lines = ax.plot(x,y) #Draw points with lines or markers connecting them
>>> ax.scatter(x,y) #Draw unconnected points, scaled or colored
>>> axes[0,0].bar([1,2,3],[3,4,5]) #Plot vertical rectangles (constant width)
>>> axes[1,0].barh([0.5,1,2.5],[0,1,2]) #Plot horional rectangles (constant height)
>>> axes[1,1].axhline(0.45) #Draw a horizontal line across axes
>>> axes[0,1].axvline(0.65) #Draw a vertical line across axes
>>> ax.fill(x,y,color='blue') #Draw filled polygons
>>> ax.fill_between(x,y,color='yellow') #Fill between y-values and 0
```

2D Data

```
>>> fig, ax = plt.subplots()
>>> im = ax.imshow(img, #Colormapped or RGB arrays
                  cmap='gist_earth',
                  interpolation='nearest',
                  vmin=-2,
                  vmax=2)
>>> axes2[0].pcolor(data2) #Pseudocolor plot of 2D array
>>> axes2[0].pcolormesh(data) #Pseudocolor plot of 2D array
>>> CS = plt.contour(Y,X,U) #Plot contours
>>> axes2[2].contourf(data1) #Plot filled contours
>>> axes2[2]= ax.clabel(CS) #Label a contour plot
```

Vector Fields

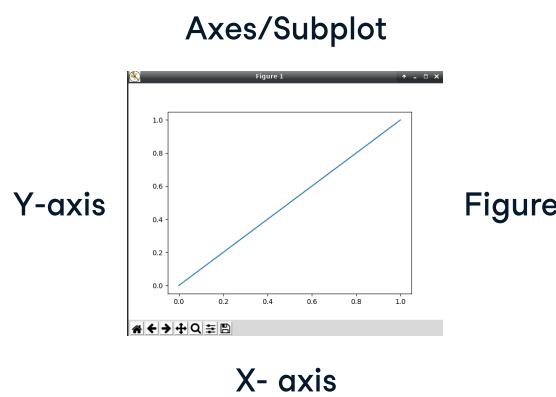
```
>>> axes[0,1].arrow(0,0,0.5,0.5) #Add an arrow to the axes
>>> axes[1,1].quiver(y,z) #Plot a 2D field of arrows
>>> axes[0,1].streamplot(X,Y,U,V) #Plot a 2D field of arrows
```

Data Distributions

```
>>> ax1.hist(y) #Plot a histogram
>>> ax3.boxplot(y) #Make a box and whisker plot
>>> ax3.violinplot(z) #Make a violin plot
```

> Plot Anatomy & Workflow

Plot Anatomy



Workflow

The basic steps to creating plots with matplotlib are:

1 Prepare Data 2 Create Plot 3 Plot 4 Customized Plot 5 Save Plot 6 Show Plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4] #Step 1
>>> y = [10,20,25,30]
>>> fig = plt.figure() #Step 2
>>> ax = fig.add_subplot(111) #Step 3
>>> ax.plot(x, y, color='lightblue', linewidth=3) #Step 3, 4
>>> ax.scatter([2,4,6],
              [5,15,25],
              color='darkgreen',
              marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png') #Step 5
>>> plt.show() #Step 6
```

> Close and Clear

```
>>> plt.cla() #Clear an axis
>>> plt.clf() #Clear the entire figure
>>> plt.close() #Close a window
```

> Plotting Cutomize Plot

Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
                  cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,y,'--',x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

Text & Annotations

```
>>> ax.text(1,
          -2.1,
          'Example Graph',
          style='italic')
>>> ax.annotate("Sine",
              xy=(8, 0),
              xycoords='data',
              xytext=(10.5, 0),
              textcoords='data',
              arrowprops=dict(arrowstyle="→",
                              connectionstyle="arc3"),)
```

Mathtext

```
>>> plt.title(r'$\sigma_i=15$', fontsize=20)
```

Limits, Legends and Layouts

Limits & Autoscaling

```
>>> ax.margins(x=0.0,y=0.1) #Add padding to a plot
>>> ax.axis('equal') #Set the aspect ratio of the plot to 1
>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5]) #Set Limits for x-and y-axis
>>> ax.set_xlim(0,10.5) #Set limits for x-axis
```

Legends

```
>>> ax.set(title='An Example Axes', #Set a title and x-and y-axis labels
          ylabel='Y-Axis',
          xlabel='X-Axis')
>>> ax.legend(loc='best') #No overlapping plot elements
```

Ticks

```
>>> ax.xaxis.set(ticks=range(1,5), #Manually set x-ticks
               tickLabels=[3,100,-12,"foo"])
>>> ax.tick_params(axis='y', #Make y-ticks longer and go in and out
                 direction='inout',
                 length=10)
```

Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5, #Adjust the spacing between subplots
                       hspace=0.3,
                       left=0.125,
                       right=0.9,
                       top=0.9,
                       bottom=0.1)
>>> fig.tight_layout() #Fit subplot(s) in to the figure area
```

Axis Spines

```
>>> ax1.spines['top'].set_visible(False) #Make the top axis line for a plot invisible
>>> ax1.spines['bottom'].set_position(('outward',10)) #Move the bottom axis line outward
```

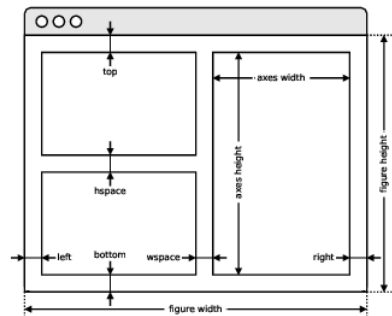


Learn Data Skills Online at www.DataCamp.com

Axes adjustments

API

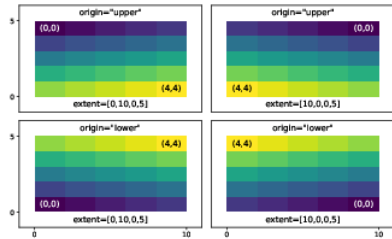
`plt.subplots_adjust(...)`



Extent & origin

API

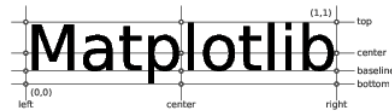
`ax.imshow(..., origin=...)`



Text alignments

API

`ax.text(..., ha=..., va=..., ...)`



Text parameters

API

`ax.text(..., family=..., size=..., weight=...)`
`ax.text(..., fontproperties=...)`

The quick brown fox
The quick brown fox
The quick brown fox
The quick brown fox
The quick brown fox
The quick brown fox
The quick brown fox
The quick brown fox

xx-large (1.73)
x-large (1.44)
large (1.20)
medium (1.00)
small (0.83)
x-small (0.69)
xx-small (0.58)

The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog

black (900)
bold (700)
semibold (600)
normal (400)
ultralight (100)

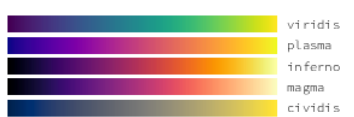
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog

monospace
serif
sans
cursive

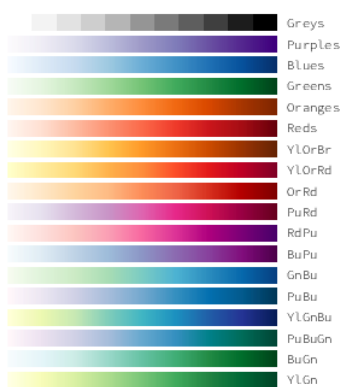
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog

italic
normal
small-caps
normal

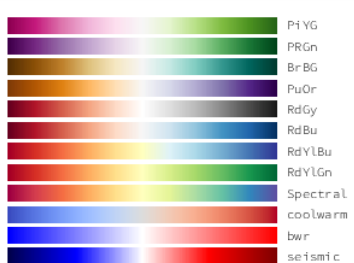
Uniform colormaps



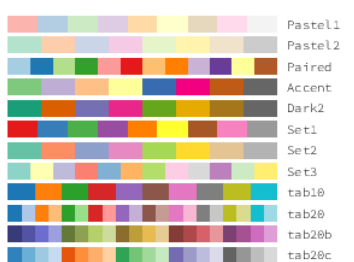
Sequential colormaps



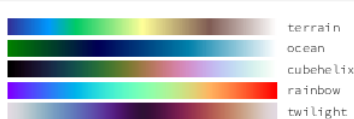
Diverging colormaps



Qualitative colormaps



Miscellaneous colormaps



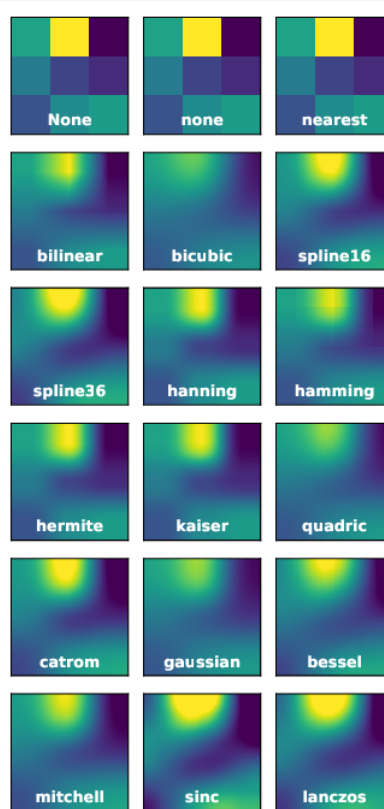
Color names

API

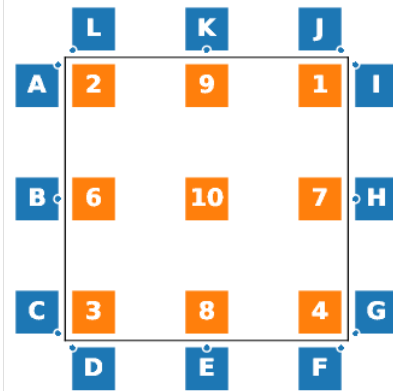


Image interpolation

API



Legend placement



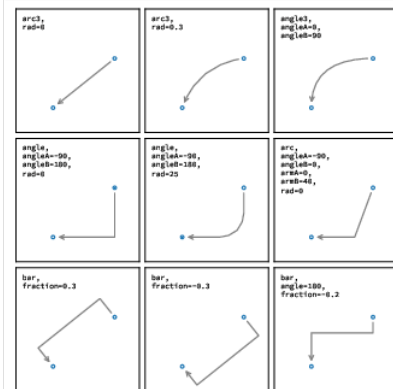
`ax.legend(loc="string", bbox_to_anchor=(x,y))`

2: upper left
6: center left
3: lower left
9: upper center
10: center
8: lower center
1: upper right
7: center right
4: lower right

A: upper right / (-0.1, 0.9)
C: lower right / (-0.1, 0.1)
E: upper center / (0.5, -0.1)
G: lower left / (1.1, 0.1)
I: upper left / (1.1, 0.9)
K: lower center / (0.5, 1.1)
B: center right / (-0.1, 0.5)
D: upper left / (0.1, -0.1)
F: upper right / (0.9, -0.1)
H: center left / (1.1, 0.5)
J: lower right / (0.9, 1.1)
L: lower left / (0.1, 1.1)

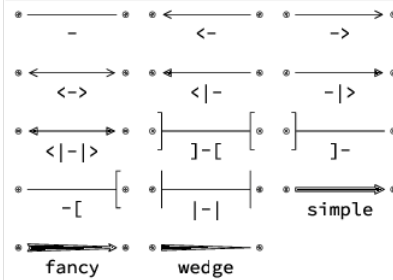
Annotation connection styles

API



Annotation arrow styles

API



How do I ...

... resize a figure?
→ `fig.set_size_inches(w,h)`

... save a figure?
→ `fig.savefig("figure.pdf")`

... save a transparent figure?
→ `fig.savefig("figure.pdf", transparent=True)`

... clear a figure?
→ `ax.clear()`

... close all figures?
→ `plt.close("all")`

... remove ticks?
→ `ax.set_xticks([])`

... remove tick labels?
→ `ax.set_xlabel('')`

... rotate tick labels?
→ `ax.set_xlabel(' ', rotation=90)`

... hide top spine?
→ `ax.spines['top'].set_visible(False)`

... hide legend border?
→ `ax.legend(frameon=False)`

... show error as shaded region?
→ `ax.fill_between(X, Y+error, Y-error)`

... draw a rectangle?
→ `ax.add_patch(plt.Rectangle((0, 0), 1, 1))`

... draw a vertical line?
→ `ax.axvline(x=0.5)`

... draw outside frame?
→ `ax.plot(..., clip_on=False)`

... use transparency?
→ `ax.plot(..., alpha=0.25)`

... convert an RGB image into a gray image?
→ `gray = 0.2989*R+0.5870*G+0.1140*B`

... set figure background color?
→ `fig.patch.set_facecolor("grey")`

... get a reversed colormap?
→ `plt.get_cmap("viridis_r")`

... get a discrete colormap?
→ `plt.get_cmap("viridis", 10)`

... show a figure for one second?
→ `fig.show(block=False), time.sleep(1)`

Performance tips

`scatter(X, Y)` slow
`plot(X, Y, marker="o", ls="")` fast
`for i in range(n): plot(X[i])` slow
`plot(sum([x+[None] for x in X], []))` fast
`cla(), imshow(u), canvas.draw()` slow
`im.set_data(u), canvas.draw()` fast

Beyond Matplotlib

Seaborn: Statistical Data Visualization
Cartopy: Geospatial Data Processing
yt: Volumetric data Visualization
mpld3: Bringing Matplotlib to the browser
Datashader: Large data processing pipeline
plotnine: A Grammar of Graphics for Python

Matplotlib Cheatsheets
Copyright (c) 2021 Matplotlib Development Team
Released under a CC-BY 4.0 International License

NUMFOCUS
OPEN CODE = BETTER SCIENCE

Python For Data Science Cheat Sheet

Pandas Basics

Learn Python for Data Science Interactively at [www.DataCamp.com](https://www.datacamp.com)



Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis tools** for the Python programming language.



Use the following import convention:

```
>>> import pandas as pd
```

Pandas Data Structures

Series

A **one-dimensional** labeled array capable of holding any data type

| | |
|---|----|
| a | 3 |
| b | -5 |
| c | 7 |
| d | 4 |

Index

```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame

| | Country | Capital | Population |
|---|---------|-----------|------------|
| 0 | Belgium | Brussels | 11190846 |
| 1 | India | New Delhi | 1303171035 |
| 2 | Brazil | Brasília | 207847528 |

A **two-dimensional** labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
           'Capital': ['Brussels', 'New Delhi', 'Brasília'],
           'Population': [11190846, 1303171035, 207847528]}
```

```
>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> df.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')

Read multiple sheets from the same file
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

```
>>> s['b']
-5

>>> df[1:]
   Country  Capital  Population
1   India  New Delhi  1303171035
2  Brazil  Brasília  207847528
```

Get one element

Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[[0], [0]]
'Belgium'

>>> df.iat([0], [0])
'Belgium'
```

Select single value by row & column

By Label

```
>>> df.loc[[0], ['Country']]
'Belgium'

>>> df.at([0], ['Country'])
'Belgium'
```

Select single value by row & column labels

By Label/Position

```
>>> df.ix[2]
Country      Brazil
Capital    Brasília
Population  207847528
```

Select single row of subset of rows

```
>>> df.ix[:, 'Capital']
0      Brussels
1    New Delhi
2     Brasilia
```

Select a single column of subset of columns

```
>>> df.ix[1, 'Capital']
'New Delhi'
```

Select rows and columns

Boolean Indexing

```
>>> s[~(s > 1)]
>>> s[(s < -1) | (s > 2)]
>>> df[df['Population'] > 1200000000]
```

Series **s** where value is not >1
s where value is <-1 or >2
Use filter to adjust DataFrame

Setting

```
>>> s['a'] = 6
```

Set index **a** of Series **s** to 6

Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
```

Drop values from rows (axis=0)
Drop values from columns (axis=1)

Sort & Rank

```
>>> df.sort_index()
>>> df.sort_values(by='Country')
>>> df.rank()
```

Sort by labels along an axis
Sort by the values along an axis
Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```

(rows, columns)
Describe index
Describe DataFrame columns
Info on DataFrame
Number of non-NA values

Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min()/df.max()
>>> df.idxmin()/df.idxmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```

Sum of values
Cumulative sum of values
Minimum/maximum values
Minimum/Maximum index value
Summary statistics
Mean of values
Median of values

Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
>>> df.applymap(f)
```

Apply function
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
a      10.0
b      NaN
c       5.0
d       7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
a      10.0
b     -5.0
c       5.0
d       7.0

>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

`read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()`

```
>>> df.to_sql('myDf', engine)
```

DataCamp

Learn Python for Data Science Interactively





Statistical Data Visualization With Seaborn

The Python visualization library **Seaborn** is based on **matplotlib** and provides a high-level interface for drawing attractive statistical graphics.

Make use of the following aliases to import the libraries:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
```

The basic steps to creating plots with Seaborn are:

- 1. Prepare some data
- 2. Control figure aesthetics
- 3. Plot with Seaborn
- 4. Further customize your plot

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> tips = sns.load_dataset("tips")
>>> sns.set_style("whitegrid")
>>> g = sns.lmplot(x="tip", y="total_bill", data=tips, aspect=2)
>>> g = (g.set_axis_labels("Tip", "Total bill (USD)")).set(xlim=(0,10),ylim=(0,100))
>>> plt.title("title")
>>> plt.show(g)
```

1 Data

Also see Lists, NumPy & Pandas

```
>>> import pandas as pd
>>> import numpy as np
>>> uniform_data = np.random.rand(10, 12)
>>> data = pd.DataFrame({'x':np.arange(1,101), 'y':np.random.normal(0,4,100)})
```

Seaborn also offers built-in data sets:

```
>>> titanic = sns.load_dataset("titanic")
>>> iris = sns.load_dataset("iris")
```

2 Figure Aesthetics

```
>>> f, ax = plt.subplots(figsize=(5,6))
```

Create a figure and one subplot

Seaborn styles

```
>>> sns.set()
>>> sns.set_style("whitegrid")
>>> sns.set_style("ticks", {'xtick.major.size':8, 'ytick.major.size':8})
>>> sns.axes_style("whitegrid")
```

(Re)set the seaborn default Set the matplotlib parameters Set the matplotlib parameters

Return a dict of params or use with to temporarily set the style

3 Plotting With Seaborn

Axis Grids

```
>>> g = sns.FacetGrid(titanic, col="survived", row="sex")
>>> g = g.map(plt.hist, "age")
>>> sns.factorplot(x="pclass", y="survived", hue="sex", data=titanic)
>>> sns.lmplot(x="sepal_width", y="sepal_length", hue="species", data=iris)
```

Subplot grid for plotting conditional relationships

Draw a categorical plot onto a Facetgrid

Plot data and regression model fits across a FacetGrid

Categorical Plots

```
Scatterplot
>>> sns.stripplot(x="species", y="petal_length", data=iris)
>>> sns.swarmplot(x="species", y="petal_length", data=iris)

Bar Chart
>>> sns.barplot(x="sex", y="survived", hue="class", data=titanic)

Count Plot
>>> sns.countplot(x="deck", data=titanic, palette="Greens_d")

Point Plot
>>> sns.pointplot(x="class", y="survived", hue="sex", data=titanic, palette={"male": "g", "female": "m"}, markers=["^", "o"], linestyle=["-", "--"])

Boxplot
>>> sns.boxplot(x="alive", y="age", hue="adult_male", data=titanic)
>>> sns.boxplot(data=iris, orient="h")

Violinplot
>>> sns.violinplot(x="age", y="sex", hue="survived", data=titanic)
```

Scatterplot with one categorical variable

Categorical scatterplot with non-overlapping points

Show point estimates and confidence intervals with scatterplot glyphs

Show count of observations

Show point estimates and confidence intervals as rectangular bars

Boxplot

Boxplot with wide-form data

Violin plot

Context Functions

```
>>> sns.set_context("talk")
>>> sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth":2.5})
```

Set context to "talk" Set context to "notebook", scale font elements and override param mapping

Color Palette

```
>>> sns.set_palette("husl", 3)
>>> sns.color_palette("husl")
>>> flatui = ["#9b59b6", "#3498db", "#95a5a6", "#e74c3c", "#34495e", "#2ecc71"]
>>> sns.set_palette(flatui)
```

Define the color palette Use with with to temporarily set palette Set your own color palette

```
>>> h = sns.PairGrid(iris)
>>> h = h.map(plt.scatter)
>>> sns.pairplot(iris)
>>> i = sns.JointGrid(x="x", y="y", data=data)
>>> i = i.plot(sns.regplot, sns.distplot)
>>> sns.jointplot("sepal_length", "sepal_width", data=iris, kind='kde')
```

Subplot grid for plotting pairwise relationships Plot pairwise bivariate distributions Grid for bivariate plot with marginal univariate plots

Plot bivariate distribution

Regression Plots

```
>>> sns.regplot(x="sepal_width", y="sepal_length", data=iris, ax=ax)
```

Plot data and a linear regression model fit

Distribution Plots

```
>>> plot = sns.distplot(data.y, kde=False, color="b")
```

Plot univariate distribution

Matrix Plots

```
>>> sns.heatmap(uniform_data, vmin=0, vmax=1)
```

Heatmap

4 Further Customizations

Also see Matplotlib

Axisgrid Objects

```
>>> g.despine(left=True)
>>> g.set_ylabels("Survived")
>>> g.set_xticklabels(rotation=45)
>>> g.set_axis_labels("Survived", "Sex")
>>> h.set(xlim=(0,5), ylim=(0,5), xticks=[0,2.5,5], yticks=[0,2.5,5])
```

Remove left spine Set the labels of the y-axis Set the tick labels for x Set the axis labels

Set the limit and ticks of the x-and y-axis

Plot

```
>>> plt.title("A Title")
>>> plt.ylabel("Survived")
>>> plt.xlabel("Sex")
>>> plt.ylim(0,100)
>>> plt.xlim(0,10)
>>> plt.setp(ax, yticks=[0,5])
>>> plt.tight_layout()
```

Add plot title Adjust the label of the y-axis Adjust the label of the x-axis Adjust the limits of the y-axis Adjust the limits of the x-axis Adjust a plot property Adjust subplot params

5 Show or Save Plot

Also see Matplotlib

```
>>> plt.show()
>>> plt.savefig("foo.png")
>>> plt.savefig("foo.png", transparent=True)
```

Show the plot Save the plot as a figure Save transparent figure

Close & Clear

Also see Matplotlib

```
>>> plt.cla()
>>> plt.clf()
>>> plt.close()
```

Clear an axis Clear an entire figure Close a window





Python For Data Science Seaborn Cheat Sheet

Learn Seaborn online at [www.DataCamp.com](https://www.datacamp.com)

Statistical Data Visualization With Seaborn

The Python visualization library **Seaborn** is based on **matplotlib** and provides a high-level interface for drawing attractive statistical graphics.

Make use of the following aliases to import the libraries:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
```

The basic steps to creating plots with Seaborn are:

1. Prepare some data
2. Control figure aesthetics
3. Plot with Seaborn
4. Further customize your plot
5. Show your plot

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> tips = sns.load_dataset("tips") #Step 1
>>> sns.set_style("whitegrid") #Step 2
>>> g = sns.lmplot(x="tip", #Step 3
                  y="total_bill",
                  data=tips,
                  aspect=2)
>>> g = (g.set_axis_labels("Tip", "Total bill(USD)").
set(xlim=(0,10),ylim=(0,100)))
>>> plt.title("title") #Step 4
>>> plt.show(g) #Step 5
```

1 Data

Also see [Lists](#), [NumPy](#) & [Pandas](#)

```
>>> import pandas as pd
>>> import numpy as np
>>> uniform_data = np.random.rand(10, 12)
>>> data = pd.DataFrame({'x':np.arange(1,101),
                        'y':np.random.normal(0,4,100)})
```

Seaborn also offers built-in data sets:

```
>>> titanic = sns.load_dataset("titanic")
>>> iris = sns.load_dataset("iris")
```

2 Figure Aesthetics

Also see [Matplotlib](#)

```
>>> f, ax = plt.subplots(figsize=(5,6)) #Create a figure and one subplot
```

Seaborn styles

```
>>> sns.set() #(Re)set the seaborn default
>>> sns.set_style("whitegrid") #Set the matplotlib parameters
>>> sns.set_style("ticks", #Set the matplotlib parameters
                {"xtick.major.size":8,
                 "ytick.major.size":8})
#Return a dict of params or use with with to temporarily set the style
>>> sns.axes_style("whitegrid")
```

3 Plotting With Seaborn

Axis Grids

```
>>> g = sns.FacetGrid(titanic, #Subplot grid for plotting conditional relationships
                    col="survived",
                    row="sex")
>>> g = g.map(plt.hist,"age")
>>> sns.factorplot(x="pclass", #Draw a categorical plot onto a Facetgrid
                 y="survived",
                 hue="sex",
                 data=titanic)
>>> sns.lmplot(x="sepal_width", #Plot data and regression model fits across a FacetGrid
              y="sepal_length",
              hue="species",
              data=iris)
>>> h = sns.PairGrid(iris) #Subplot grid for plotting pairwise relationships
>>> h = h.map(plt.scatter)
>>> sns.pairplot(iris) #Plot pairwise bivariate distributions
>>> i = sns.JointGrid(x="x", #Grid for bivariate plot with marginal univariate plots
                   y="y",
                   data=data)
>>> i = i.plot(sns.regplot,
              sns.distplot)
>>> sns.jointplot("sepal_length", #Plot bivariate distribution
                "sepal_width",
                data=iris,
                kind='kde')
```

4 Further Customizations

Also see [Matplotlib](#)

Axisgrid Objects

```
>>> g.despine(left=True) #Remove left spine
>>> g.set_ylabels("Survived") #Set the labels of the y-axis
>>> g.set_xticklabels(rotation=45) #Set the tick labels for x
>>> g.set_axis_labels("Survived", #Set the axis labels
                    "Sex")
>>> h.set(xlim=(0,5), #Set the limit and ticks of the x-and y-axis
        ylim=(0,5),
        xticks=[0,2.5,5],
        yticks=[0,2.5,5])
```

Plot

```
>>> plt.title("A Title") #Add plot title
>>> plt.ylabel("Survived") #Adjust the label of the y-axis
>>> plt.xlabel("Sex") #Adjust the label of the x-axis
>>> plt.ylim(0,100) #Adjust the limits of the y-axis
>>> plt.xlim(0,10) #Adjust the limits of the x-axis
>>> plt.setp(ax, yticks=[0,5]) #Adjust a plot property
>>> plt.tight_layout() #Adjust subplot params
```

Regression Plots

```
>>> sns.regplot(x="sepal_width", #Plot data and a linear regression model fit
               y="sepal_length",
               data=iris,
               ax=ax)
```

Distribution Plots

```
>>> plot = sns.distplot(data.y, #Plot univariate distribution
                       kde=False,
                       color="b")
```

Matrix Plots

```
>>> sns.heatmap(uniform_data, vmin=0, vmax=1) #Heatmap
```

Categorical Plots

Scatterplot

```
>>> sns.stripplot(x="species", #Scatterplot with one categorical variable
                 y="petal_length",
                 data=iris)
>>> sns.swarmplot(x="species", #Categorical scatterplot with non-overlapping points
                 y="petal_length",
                 data=iris)
```

Bar Chart

```
>>> sns.barplot(x="sex", #Show point estimates & confidence intervals with scatterplot glyphs
               y="survived",
               hue="class",
               data=titanic)
```

Count Plot

```
>>> sns.countplot(x="deck", #Show count of observations
                 data=titanic,
                 palette="Greens_d")
```

Point Plot

```
>>> sns.pointplot(x="class", #Show point estimates & confidence intervals as rectangular bars
                 y="survived",
                 hue="sex",
                 data=titanic,
                 palette={"male": "g",
                          "female": "m"},
                 markers=["^", "o"],
                 linestyle=["-", "--"])
```

Boxplot

```
>>> sns.boxplot(x="alive", #Boxplot
               y="age",
               hue="adult_male",
               data=titanic)
>>> sns.boxplot(data=iris, orient="h") #Boxplot with wide-form data
```

Violinplot

```
>>> sns.violinplot(x="age", #Violin plot
                  y="sex",
                  hue="survived",
                  data=titanic)
```

5 Show or Save Plot

Also see [Matplotlib](#)

```
>>> plt.show() #Show the plot
>>> plt.savefig("foo.png") #Save the plot as a figure
>>> plt.savefig("foo.png", #Save transparent figure
               transparent=True)
```

> Close & Clear

Also see [Matplotlib](#)

```
>>> plt.cla() #Clear an axis
>>> plt.clf() #Clear an entire figure
>>> plt.close() #Close a window
```



Learn Data Skills Online at [www.DataCamp.com](https://www.datacamp.com)

Advanced R

Cheat Sheet

Created by: Arianne Colton and Sean Chen

Environment Basics

Environment – **Data structure** (with two components below) that powers lexical scoping

```
Create environment: env1<-new.env()
```

1. **Named list** (“Bag of names”) – each name points to an object stored elsewhere in memory.

If an object has no names pointing to it, it gets automatically deleted by the garbage collector.

- Access with: `ls('env1')`
- 2. **Parent environment** – used to implement lexical scoping. If a name is not found in an environment, then R will look in its parent (and so on).
- Access with: `parent.env('env1')`

Four special environments

1. **Empty environment** – ultimate ancestor of all environments
 - Parent: none
 - Access with: `emptyenv()`
2. **Base environment** - environment of the base package
 - Parent: empty environment
 - Access with: `baseenv()`
3. **Global environment** – the interactive workspace that you normally work in
 - Parent: environment of last attached package
 - Access with: `globalenv()`
4. **Current environment** – environment that R is currently working in (may be any of the above and others)
 - Parent: empty environment
 - Access with: `environment()`

Environments

Search Path

Search path – mechanism to look up objects, particularly functions.

- Access with : `search()` – lists all parents of the global environment (see Figure 1)
- Access any environment on the search path:
`as.environment('package:base')`

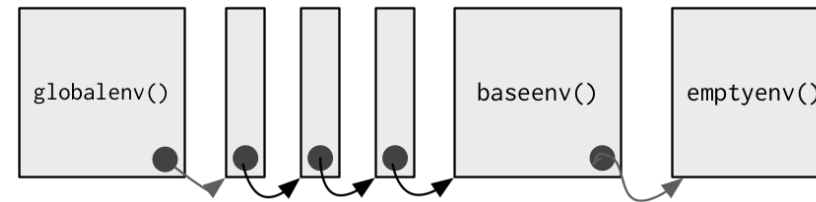


Figure 1 – The Search Path

- Mechanism : always start the search from global environment, then inside the latest attached package environment.
- New package loading with `library()/require()` : new package is attached right after global environment. (See Figure 2)
- Name conflict in two different package : functions with the same name, latest package function will get called.

```
search() :
```

```
'.GlobalEnv' ... 'Autoloads' 'package:base'
```

```
library(reshape2); search()
```

```
'.GlobalEnv' 'package:reshape2' ... 'Autoloads' 'package:base'
```

NOTE: Autoloads : special environment used for saving memory by only loading package objects (like big datasets) when needed

Figure 2 – Package Attachment

Binding Names to Values

Assignment – act of binding (or rebinding) a name to a value in an environment.

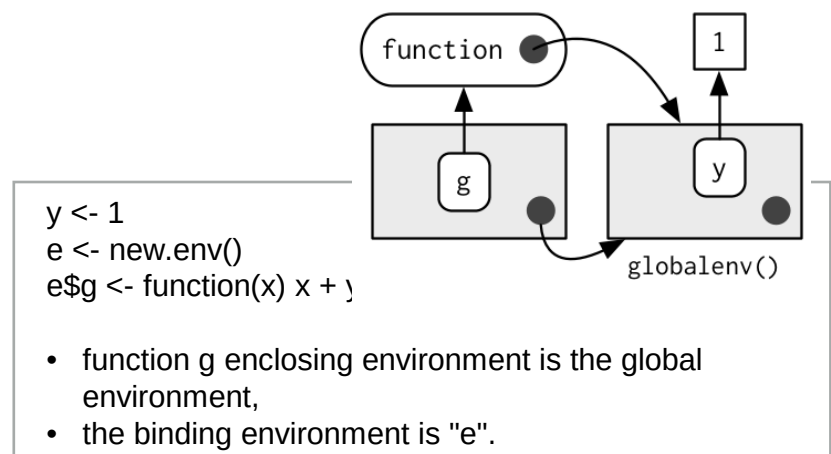
1. `<-` (Regular assignment arrow) – always creates a variable in the current environment
2. `<<-` (Deep assignment arrow) - modifies an existing variable found by walking up the parent environments

Warning: If `<<-` doesn't find an existing variable, it will create one in the global environment.

Function Environments

1. **Enclosing environment** - an environment where the function is created. It determines how function finds value.
 - Enclosing environment never changes, even if the function is moved to a different environment.
 - Access with: `environment('func1')`
2. **Binding environment** - all environments that the function has a binding to. It determines how we find the function.
 - Access with: `pryr::where('func1')`

Example (for enclosing and binding environment):



3. **Execution environment** - new created environments to host a function call execution.
 - Two parents :
 - I. Enclosing environment of the function
 - II. Calling environment of the function
 - Execution environment is thrown away once the function has completed.
4. **Calling environment** - environments where the function was called.
 - Access with: `parent.frame('func1')`
 - Dynamic scoping :
 - About : look up variables in the calling environment rather than in the enclosing environment
 - Usage : most useful for developing functions that aid interactive data analysis

Data Structures

| | Homogeneous | Heterogeneous |
|----|---------------|---------------|
| 1d | Atomic vector | List |
| 2d | Matrix | Data frame |
| nd | Array | |

Note: R has no 0-dimensional or scalar types. Individual numbers or strings, are actually vectors of length one, NOT scalars.

Human readable description of any R data structure :

```
str(variable)
```

Every **Object** has a mode and a class

1. **Mode:** represents how an object is stored in memory

- 'type' of the object from R's point of view
- Access with: **typeof()**

2. **Class:** represents the object's abstract type

- 'type' of the object from R's object-oriented programming point of view
- Access with: **class()**

| | typeof() | class() |
|------------------------------|-----------|------------|
| strings or vector of strings | character | character |
| numbers or vector of numbers | numeric | numeric |
| list | list | list |
| data.frame | list | data.frame |

Factors

1. Factors are built on top of integer vectors using two attributes :

```
class(x) -> 'factor'
```

```
levels(x) # defines the set of allowed values
```

2. Useful when you know the possible values a variable may take, even if you don't see all values in a given dataset.

Warning on Factor Usage:

1. Factors look and often behave like character vectors, they are actually integers. Be careful when treating them like strings.
2. Most data loading functions automatically convert character vectors to factors. (Use argument `stringAsFactors = FALSE` to suppress this behavior)

Object Oriented (OO) Field Guide

Object Oriented Systems

R has three object oriented systems :

1. **S3** is a very casual system. It has no formal definition of classes. It implements generic function OO.

- **Generic-function OO** - a special type of function called a generic function decides which method to call.

| | |
|-----------|--------------------------|
| Example: | drawRect(canvas, 'blue') |
| Language: | R |

- **Message-passing OO** - messages (methods) are sent to objects and the object determines which function to call.

| | |
|-----------|-------------------------|
| Example: | canvas.drawRect('blue') |
| Language: | Java, C++, and C# |

2. **S4** works similarly to S3, but is more formal. Two major differences to S3 :

- **Formal class definitions** - describe the representation and inheritance for each class, and has special helper functions for defining generics and methods.
- **Multiple dispatch** - generic functions can pick methods based on the class of any number of arguments, not just one.

3. **Reference classes** are very different from S3 and S4:

- **Implements message-passing OO** - methods belong to classes, not functions.
- **Notation** - \$ is used to separate objects and methods, so method calls look like **canvas\$drawRect('blue')**.

S3

1. **About S3 :**

- R's first and simplest OO system
- Only OO system used in the base and stats package
- Methods belong to functions, not to objects or classes.

2. **Notation :**

- **generic.class()**

| | |
|-------------|--------------------------------------|
| mean.Date() | Date method for the generic - mean() |
|-------------|--------------------------------------|

3. **Useful 'Generic' Operations**

- Get all methods that belong to the 'mean' generic:
 - **Methods('mean')**
- List all generics that have a method for the 'Date' class :
 - **methods(class = 'Date')**

4. **S3 objects** are usually built on top of lists, or atomic vectors with attributes.

- Factor and data frame are S3 class
- Useful operations:

| | |
|--|--|
| Check if object is an S3 object | is.object(x) & !isS4(x) or pryr::otype() |
| Check if object inherits from a specific class | inherits(x, 'classname') |
| Determine class of any object | class(x) |

Base Type (C Structure)

R base types - the internal C-level types that underlie the above OO systems.

- **Includes** : atomic vectors, list, functions, environments, etc.
- **Useful operation** : Determine if an object is a base type (Not S3, S4 or RC) **is.object(x)** returns FALSE

- **Internal representation** : C structure (or struct) that includes :

- Contents of the object
- Memory Management Information
- Type
 - Access with: **typeof()**

Functions

Function Basics

Functions – objects in their own right
All R functions have three parts:

| | |
|---------------|---|
| body() | code inside the function |
| formals() | list of arguments which controls how you can call the function |
| environment() | “map” of the location of the function’s variables (see “Enclosing Environment”) |

Every operation is a function call

- +, for, if, [, \$, { ...
- $x + y$ is the same as ``+`(x, y)`

Note: the backtick (```), lets you refer to functions or variables that have otherwise reserved or illegal names.

Lexical Scoping

What is Lexical Scoping?

- Looks up value of a symbol. (see “Enclosing Environment”)
- **findGlobals()** - lists all the external dependencies of a function

```
f <- function() x + 1
codetools::findGlobals(f)
> '+' 'x'
```

```
environment(f) <- emptyenv()
f()
# error in f(): could not find function “+”
```

- R relies on lexical scoping to find everything, even the + operator.

Function Arguments

Arguments – passed by reference and copied on modify

1. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.

2. Check if an argument was supplied : **missing()**

```
i <- function(a, b) {
  missing(a) -> # return true or false
}
```

3. Lazy evaluation – since x is not used **stop("This is an error!")** never get evaluated.

```
f <- function(x) {
  10
}
f(stop("This is an error!")) -> 10
```

4. Force evaluation

```
f <- function(x) {
  force(x)
  10
}
```

5. Default arguments evaluation

```
f <- function(x = ls()) {
  a <- 1
  x
}
```

| | |
|-----------------------------|--------------------------------------|
| <code>f()</code> -> 'a' 'x' | ls() evaluated inside f |
| <code>f(ls())</code> | ls() evaluated in global environment |

Return Values

- **Last expression evaluated or explicit return().**
Only use explicit return() when returning early.
- **Return ONLY single object.**
Workaround is to return a list containing any number of objects.
- **Invisible return object value** - not printed out by default when you call the function.

```
f1 <- function() invisible(1)
```

Primitive Functions

What are Primitive Functions?

1. Call C code directly with **.Primitive()** and contain no R code

```
print(sum) :
> function (..., na.rm = FALSE) .Primitive('sum')
```

2. **formals()**, **body()**, and **environment()** are all NULL
3. Only found in base package
4. More efficient since they operate at a low level

Infix Functions

What are Infix Functions?

1. Function name comes in between its arguments, like + or –
2. All user-created infix functions must start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
'new' %+% 'string'
```

3. Useful way of providing a default value in case the output of another function is NULL:

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

Replacement Functions

What are Replacement Functions?

1. Act like they modify their arguments in place, and have the special name `xxx <-`
2. Actually create a modified copy. Can use **pryr::address()** to find the memory address of the underlying object

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
```

Subsetting

Subsetting returns a copy of the original data, NOT copy-on modified

Simplifying vs. Preserving Subsetting

1. Simplifying subsetting

- Returns the **simplest** possible data structure that can represent the output

2. Preserving subsetting

- Keeps the structure of the output the **same** as the input.
- When you use `drop = FALSE`, it's preserving

| | Simplifying* | Preserving |
|------------|--|--|
| Vector | <code>x[[1]]</code> | <code>x[1]</code> |
| List | <code>x[[1]]</code> | <code>x[1]</code> |
| Factor | <code>x[1:4, drop = T]</code> | <code>x[1:4]</code> |
| Array | <code>x[1,]</code> or <code>x[, 1]</code> | <code>x[1, , drop = F]</code> or <code>x[, 1, drop = F]</code> |
| Data frame | <code>x[, 1]</code> or <code>x[[1]]</code> | <code>x[, 1, drop = F]</code> or <code>x[1]</code> |

Simplifying behavior varies slightly between different data types:

1. Atomic Vector

- `x[[1]]` is the same as `x[1]`

2. List

- `[ ]` always returns a list
- Use `[[ ]]` to get list contents, this returns a single value piece out of a list

3. Factor

- Drops any unused levels but it remains a factor class

4. Matrix or Array

- If any of the dimensions has length 1, that dimension is dropped

5. Data Frame

- If output is a single column, it returns a vector instead of a data frame

Data Frame Subsetting

Data Frame – possesses the **characteristics of both lists and matrices**. If you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices

1. Subset with a single vector : Behave like lists

```
df1[c('col1', 'col2')]
```

2. Subset with two vectors : Behave like matrices

```
df1[, c('col1', 'col2')]
```

The results are the same in the above examples, however, results are different if subsetting with only one column. (see below)

1. Behave like matrices

```
str(df1[, 'col1']) -> int [1:3]
```

- Result: the result is a vector

2. Behave like lists

```
str(df1['col1']) -> 'data.frame'
```

- Result: the result remains a data frame of 1 column

\$ Subsetting Operator

1. About Subsetting Operator

- Useful shorthand for `[[` combined with character subsetting

```
x$y is equivalent to x[['y', exact = FALSE]]
```

2. Difference vs. `[[`

- `$` does partial matching, `[[` does not

```
x <- list(abc = 1)
x$a -> 1      # since "exact = FALSE"
x[['a']] ->   # would be an error
```

3. Common mistake with `$`

- Using it when you have the name of a column stored in a variable

```
var <- 'cyl'
x$var
# doesn't work, translated to x[['var']]
# Instead use x[[var]]
```

Examples

1. Lookup tables (character subsetting)

```
x <- c('m', 'f', 'u', 'f', 'f', 'm', 'm')
lookup <- c(m = 'Male', f = 'Female', u = NA)
lookup[x]
> m f u f f m m
> 'Male' 'Female' NA 'Female' 'Female' 'Male' 'Male'
unname(lookup[x])
> 'Male' 'Female' NA 'Female' 'Female' 'Male' 'Male'
```

2. Matching and merging by hand (integer subsetting)

Lookup table which has multiple columns of information:

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(
  grade = 3:1,
  desc = c('Excellent', 'Good', 'Poor'),
  fail = c(F, F, T)
)
```

First Method

```
id <- match(grades, info$grade)
info[id, ]
```

Second Method

```
rownames(info) <- info$grade
info[as.character(grades), ]
```

3. Expanding aggregated counts (integer subsetting)

- Problem:** a data frame where identical rows have been collapsed into one and a count column has been added
- Solution:** `rep()` and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index: `rep(x, y)` `rep` replicates the values in `x`, `y` times.

```
df1$countCol is c(3, 5, 1)
rep(1:nrow(df1), df1$countCol)
> 1 1 1 2 2 2 2 2 3
```

4. Removing columns from data frames (character subsetting)

There are two ways to remove columns from a data frame:

| | |
|--|-------------------------------------|
| Set individual columns to NULL | <code>df1\$col3 <- NULL</code> |
| Subset to return only columns you want | <code>df1[c('col1', 'col2')]</code> |

5. Selecting rows based on a condition (logical subsetting)

- This is the most commonly used technique for extracting rows out of a data frame.

```
df1[df1$col1 == 5 & df1$col2 == 4, ]
```


Subsetting continued

Boolean Algebra vs. Sets (Logical and Integer Subsetting)

1. **Using integer subsetting** is more effective when:
 - You want to find the first (or last) TRUE.
 - You have very few TRUEs and very many FALSEs; a set representation may be faster and require less storage.
2. **which()** - conversion from boolean representation to integer representation

```
which(c(T, F, T F)) -> 1 3
```

- Integer representation length : is always $\leq$ boolean representation length
- Common mistakes :
 - I. Use **x[which(y)]** instead of **x[y]**
 - II. **x[-which(y)]** is not equivalent to **x[!y]**

Recommendation:

Avoid switching from logical to integer subsetting unless you want, for example, the first or last TRUE value

Subsetting with Assignment

1. All subsetting operators can be combined with assignment to modify selected values of the input vector.

```
df1$col1[df1$col1 < 8] <- 0
```

2. Subsetting with nothing in conjunction with assignment :
 - Why : Preserve original object class and structure

```
df1[] <- lapply(df1, as.integer)
```

Debugging, Condition Handling and Defensive Programming

Debugging Methods

1. **traceback()** or **RStudio's error inspector**
 - Lists the sequence of calls that lead to the error
2. **browser()** or **RStudio's breakpoints tool**
 - Opens an interactive debug session at an arbitrary location in the code
3. **options(error = browser)** or **RStudio's "Rerun with Debug" tool**
 - Opens an interactive debug session where the error occurred
 - Error Options:
options(error = recover)
 - Difference vs. 'browser': can enter environment of any of the calls in the stack
options(error = dump_and_quit)
 - Equivalent to 'recover' for non-interactive mode
 - Creates **last.dump.rda** in the current working directory

In batch R process :

```
dump_and_quit <- function() {  
  # Save debugging info to file  
  last.dump.rda  
  dump.frames(to.file = TRUE)  
  # Quit R with error status  
  q(status = 1)  
}  
options(error = dump_and_quit)
```

In a later interactive session :

```
load("last.dump.rda")  
debugger()
```

Condition Handling of Expected Errors

1. **Communicating potential problems to users:**
 - I. **stop()**
 - Action : raise fatal error and force all execution to terminate
 - Example usage : when there is no way for a function to continue
 - II. **warning()**
 - Action : generate warnings to display potential problems
 - Example usage : when some of elements of a vectorized input are invalid
 - III. **message()**
 - Action : generate messages to give informative output
 - Example usage : when you would like to print the steps of a program execution
2. **Handling conditions programmatically:**
 - I. **try()**
 - Action : gives you the ability to continue execution even when an error occurs
 - II. **tryCatch()**
 - Action : lets you specify handler functions that control what happens when a condition is signaled

```
result = tryCatch(code,  
  error = function(c) "error",  
  warning = function(c) "warning",  
  message = function(c) "message"  
)
```

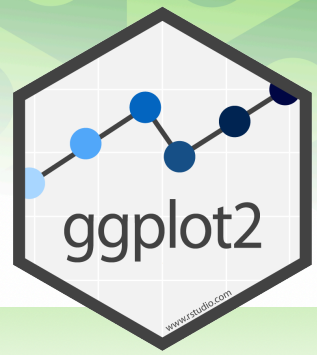
Use `conditionMessage(c)` or `c$message` to extract the message associated with the original error.

Defensive Programming

Basic principle : "fail fast", to raise an error as soon as something goes wrong

1. **stopifnot()** or use 'assertthat' package - check inputs are correct
2. **Avoid subset(), transform() and with()** - these are non-standard evaluation, when they fail, often fail with uninformative error messages.
3. **Avoid [and apply()** - functions that can return different types of output.
 - Recommendation : Whenever subsetting a data frame in a function, you should always use **drop = FALSE**

Data visualization with ggplot2 : : CHEAT SHEET

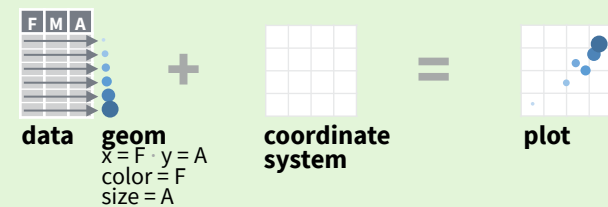


Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and **geoms**—visual marks that represent data points.



To display values, map variables in the data to visual properties of the geom (**aesthetics**) like **size**, **color**, and **x** and **y** locations.



Complete the template below to build a graph.

```
ggplot (data = <DATA>) +  
  <GEOM_FUNCTION> (mapping = aes (<MAPPINGS>),  
    stat = <STAT>, position = <POSITION>) +  
  <COORDINATE_FUNCTION> +  
  <FACET_FUNCTION> +  
  <SCALE_FUNCTION> +  
  <THEME_FUNCTION>
```

required

Not required, sensible defaults supplied

ggplot(data = mpg, aes(x = cty, y = hwy)) Begins a plot that you finish by adding layers to. Add one geom function per layer.

last_plot() Returns the last plot.

ggsave("plot.png", width = 5, height = 5) Saves last plot as 5' x 5' file named "plot.png" in working directory. Matches file type to file extension.

Aes Common aesthetic values.

color and **fill** - string ("red", "#RRGGBB")

linetype - integer or string (0 = "blank", 1 = "solid", 2 = "dashed", 3 = "dotted", 4 = "dotdash", 5 = "longdash", 6 = "twodash")

lineend - string ("round", "butt", or "square")

linejoin - string ("round", "mitre", or "bevel")

size - integer (line width in mm)

shape - integer/shape name or a single character ("a")

0 1 2 3 4 5 6 7 8 9 10 11 12
□ ○ △ + × ◇ ▼ ☆ * ◇ ⊗ ⊕ ⊗ ⊕
13 14 15 16 17 18 19 20 21 22 23 24 25
⊠ ⊡ ⊢ ⊣ ⊤ ⊥ ⊦ ⊧ ⊨ ⊩ ⊪ ⊫ ⊬ ⊭ ⊮ ⊯



Geoms

Use a geom function to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.

GRAPHICAL PRIMITIVES

```
a <- ggplot(economics, aes(date, unemploy))  
b <- ggplot(seals, aes(x = long, y = lat))
```

a + geom_blank() and **a + expand_limits()**
Ensure limits include values across all plots.

b + geom_curve(aes(yend = lat + 1, xend = long + 1, curvature = 1) - x, xend, y, yend, alpha, angle, color, curvature, linetype, size)

a + geom_path(lineend = "butt", linejoin = "round", linemitre = 1) - x, y, alpha, color, group, linetype, size

a + geom_polygon(aes(alpha = 50)) - x, y, alpha, color, fill, group, subgroup, linetype, size

b + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1)) - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size

a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900)) - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS

common aesthetics: x, y, alpha, color, linetype, size

b + geom_abline(aes(intercept = 0, slope = 1))
b + geom_hline(aes(yintercept = lat))
b + geom_vline(aes(xintercept = long))

b + geom_segment(aes(yend = lat + 1, xend = long + 1))
b + geom_spoke(aes(angle = 1:1155, radius = 1))

ONE VARIABLE continuous

```
c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)
```

c + geom_area(stat = "bin")
x, y, alpha, color, fill, linetype, size

c + geom_density(kernel = "gaussian")
x, y, alpha, color, fill, group, linetype, size, weight

c + geom_dotplot()
x, y, alpha, color, fill

c + geom_freqpoly()
x, y, alpha, color, group, linetype, size

c + geom_histogram(binwidth = 5)
x, y, alpha, color, fill, linetype, size, weight

c2 + geom_qq(aes(sample = hwy))
x, y, alpha, color, fill, linetype, size, weight

discrete

```
d <- ggplot(mpg, aes(fl))
```

d + geom_bar()
x, alpha, color, fill, linetype, size, weight

TWO VARIABLES

both continuous

```
e <- ggplot(mpg, aes(cty, hwy))
```

e + geom_label(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

e + geom_point()
x, y, alpha, color, fill, shape, size, stroke

e + geom_quantile()
x, y, alpha, color, group, linetype, size, weight

e + geom_rug(sides = "bl")
x, y, alpha, color, linetype, size

e + geom_smooth(method = lm)
x, y, alpha, color, fill, group, linetype, size, weight

e + geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

one discrete, one continuous

```
f <- ggplot(mpg, aes(class, hwy))
```

f + geom_col()
x, y, alpha, color, fill, group, linetype, size

f + geom_boxplot()
x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight

f + geom_dotplot(binaxis = "y", stackdir = "center")
x, y, alpha, color, fill, group

f + geom_violin(scale = "area")
x, y, alpha, color, fill, group, linetype, size, weight

both discrete

```
g <- ggplot(diamonds, aes(cut, color))
```

g + geom_count()
x, y, alpha, color, fill, shape, size, stroke

e + geom_jitter(height = 2, width = 2)
x, y, alpha, color, fill, shape, size

THREE VARIABLES

```
seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))
```

l + geom_contour(aes(z = z))
x, y, z, alpha, color, group, linetype, size, weight

l + geom_contour_filled(aes(fill = z))
x, y, alpha, color, fill, group, linetype, size, subgroup

continuous bivariate distribution

```
h <- ggplot(diamonds, aes(carat, price))
```

h + geom_bin2d(binwidth = c(0.25, 500))
x, y, alpha, color, fill, linetype, size, weight

h + geom_density_2d()
x, y, alpha, color, group, linetype, size

h + geom_hex()
x, y, alpha, color, fill, size

continuous function

```
i <- ggplot(economics, aes(date, unemploy))
```

i + geom_area()
x, y, alpha, color, fill, linetype, size

i + geom_line()
x, y, alpha, color, group, linetype, size

i + geom_step(direction = "hv")
x, y, alpha, color, group, linetype, size

visualizing error

```
df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)  
j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))
```

j + geom_crossbar(fatten = 2) - x, y, ymax, ymin, alpha, color, fill, group, linetype, size

j + geom_errorbar() - x, ymax, ymin, alpha, color, group, linetype, size, width
Also **geom_errorbarh**()

j + geom_linerange()
x, ymin, ymax, alpha, color, group, linetype, size

j + geom_pointrange() - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

maps

```
data <- data.frame(murder = USArrests$Murder,  
  state = tolower(rownames(USArrests)))
```

```
map <- map_data("state")  
k <- ggplot(data, aes(fill = murder))
```

k + geom_map(aes(map_id = state), map = map) + **expand_limits**(x = map\$long, y = map\$lat)
map_id, alpha, color, fill, linetype, size

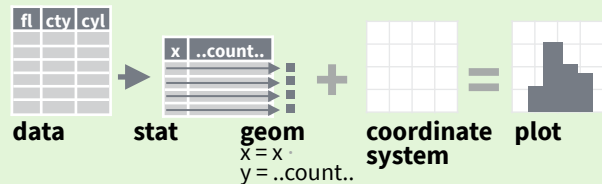
l + geom_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE)
x, y, alpha, fill

l + geom_tile(aes(fill = z))
x, y, alpha, color, fill, linetype, size, width

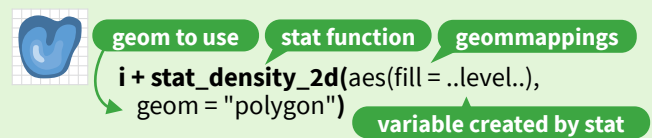
Stats

An alternative way to build a layer.

A stat builds new variables to plot (e.g., count, prop).



Visualize a stat by changing the default stat of a geom function, **geom_bar(stat="count")** or by using a stat function, **stat_count(geom="bar")**, which calls a default geom to make a layer (equivalent to a geom function). Use **..name..** syntax to map stat variables to aesthetics.



```
c + stat_bin(binwidth = 1, boundary = 10)
x, y | ..count.., ..ncount.., ..density.., ..ndensity..

c + stat_count(width = 1) x, y | ..count.., ..prop..

c + stat_density(adjust = 1, kernel = "gaussian")
x, y | ..count.., ..density.., ..scaled..

e + stat_bin_2d(bins = 30, drop = T)
x, y, fill | ..count.., ..density..

e + stat_bin_hex(bins = 30) x, y, fill | ..count.., ..density..

e + stat_density_2d(contour = TRUE, n = 100)
x, y, color, size | ..level..

e + stat_ellipse(level = 0.95, segments = 51, type = "t")

l + stat_contour(aes(z = z)) x, y, z, order | ..level..

l + stat_summary_hex(aes(z = z), bins = 30, fun = max)
x, y, z, fill | ..value..

l + stat_summary_2d(aes(z = z), bins = 30, fun = mean)
x, y, z, fill | ..value..

f + stat_boxplot(coef = 1.5)
x, y | ..lower.., ..middle.., ..upper.., ..width.., ..ymin.., ..ymax..

f + stat_ydensity(kernel = "gaussian", scale = "area") x, y
| ..density.., ..scaled.., ..count.., ..n.., ..violinwidth.., ..width..

e + stat_ecdf(n = 40) x, y | ..x.., ..y..

e + stat_quantile(quantiles = c(0.1, 0.9),
formula = y ~ log(x), method = "rq") x, y | ..quantile..

e + stat_smooth(method = "lm", formula = y ~ x, se = T,
level = 0.95) x, y | ..se.., ..x.., ..y.., ..ymin.., ..ymax..

ggplot() + xlim(-5, 5) + stat_function(fun = dnorm,
n = 20, geom = "point") x | ..x.., ..y..

ggplot() + stat_qq(aes(sample = 1:100))
x, y, sample | ..sample.., ..theoretical..

e + stat_sum() x, y, size | ..n.., ..prop..

e + stat_summary(fun.data = "mean_cl_boot")

h + stat_summary_bin(fun = "mean", geom = "bar")

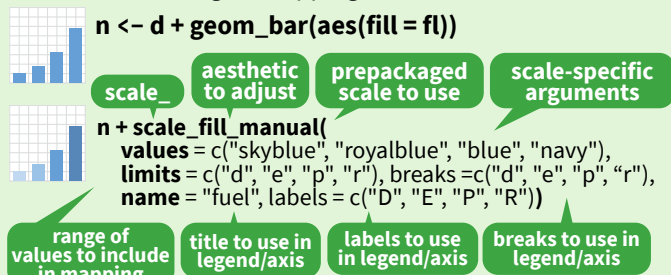
e + stat_identity()

e + stat_unique()
```

Scales

Override defaults with **scales** package.

Scales map data values to the visual values of an aesthetic. To change a mapping, add a new scale.



GENERAL PURPOSE SCALES

Use with most aesthetics

scale_*_continuous() - Map cont' values to visual ones.
scale_*_discrete() - Map discrete values to visual ones.
scale_*_binned() - Map continuous values to discrete bins.
scale_*_identity() - Use data values as visual ones.
scale_*_manual(values = c()) - Map discrete values to manually chosen visual ones.
scale_*_date(date_labels = "%m/%d"),
date_breaks = "2 weeks" - Treat data values as dates.
scale_*_datetime() - Treat data values as date times.
Same as **scale_*_date()**. See `?strptime` for label formats.

X & Y LOCATION SCALES

Use with x or y aesthetics (x shown here)

scale_x_log10() - Plot x on log10 scale.
scale_x_reverse() - Reverse the direction of the x axis.
scale_x_sqrt() - Plot x on square root scale.

COLOR AND FILL SCALES (DISCRETE)

n + scale_fill_brewer(palette = "Blues")
For palette choices:
`RColorBrewer::display.brewer.all()`
n + scale_fill_grey(start = 0.2,
end = 0.8, na.value = "red")

COLOR AND FILL SCALES (CONTINUOUS)

o <- c + geom_dotplot(aes(fill = ..x..))
o + scale_fill_distiller(palette = "Blues")
o + scale_fill_gradient(low="red", high="yellow")
o + scale_fill_gradient2(low = "red", high = "blue",
mid = "white", midpoint = 25)
o + scale_fill_gradientn(colors = topo.colors(6))
Also: `rainbow()`, `heat.colors()`, `terrain.colors()`,
`cm.colors()`, `RColorBrewer::brewer.pal()`

SHAPE AND SIZE SCALES

p <- e + geom_point(aes(shape = fl, size = cyl))
p + scale_shape() + scale_size()
p + scale_shape_manual(values = c(3:7))
p + scale_radius(range = c(1,6))
p + scale_size_area(max_size = 6)

Coordinate Systems

```
r <- d + geom_bar()

r + coord_cartesian(xlim = c(0, 5)) - xlim, ylim
The default cartesian coordinate system.

r + coord_fixed(ratio = 1/2)
ratio, xlim, ylim - Cartesian coordinates with
fixed aspect ratio between x and y units.

ggplot(mpg, aes(y = fl)) + geom_bar()
Flip cartesian coordinates by switching
x and y aesthetic mappings.

r + coord_polar(theta = "x", direction=1)
theta, start, direction - Polar coordinates.

r + coord_trans(y = "sqrt") - x, y, xlim, ylim
Transformed cartesian coordinates. Set xtrans
and ytrans to the name of a window function.

pi + coord_quickmap()
pi + coord_map(projection = "ortho", orientation
= c(41, -74, 0)) - projection, xlim, ylim
Map projections from the mapproj package
(mercator (default), azequalarea, lagrange, etc.).
```

Position Adjustments

Position adjustments determine how to arrange geoms that would otherwise occupy the same space.

```
s <- ggplot(mpg, aes(fl, fill = drv))

s + geom_bar(position = "dodge")
Arrange elements side by side.

s + geom_bar(position = "fill")
Stack elements on top of one
another, normalize height.

e + geom_point(position = "jitter")
Add random noise to X and Y position of
each element to avoid overplotting.

e + geom_label(position = "nudge")
Nudge labels away from points.

s + geom_bar(position = "stack")
Stack elements on top of one another.
```

Each position adjustment can be recast as a function with manual **width** and **height** arguments:

```
s + geom_bar(position = position_dodge(width = 1))
```

Themes

```
r + theme_bw()
White background
with grid lines.

r + theme_classic()
r + theme_light()

r + theme_gray()
Grey background
(default theme).

r + theme_dark()
Dark for contrast.

r + theme_linedraw()
Minimal theme.

r + theme_minimal()
Empty theme.
```

r + theme() Customize aspects of the theme such as axis, legend, panel, and facet properties.
r + ggtitle("Title") + theme(plot.title.position = "plot")
r + theme(panel.background = element_rect(fill = "blue"))

Faceting

Facets divide a plot into subplots based on the values of one or more discrete variables.

```
t <- ggplot(mpg, aes(cty, hwy)) + geom_point()
```

t + facet_grid(cols = vars(fl))
Facet into columns based on fl.
t + facet_grid(rows = vars(year))
Facet into rows based on year.
t + facet_grid(rows = vars(year), cols = vars(fl))
Facet into both rows and columns.
t + facet_wrap(vars(fl))
Wrap facets into a rectangular layout.

Set **scales** to let axis limits vary across facets.

t + facet_grid(rows = vars(drv), cols = vars(fl), scales = "free")
x and y axis limits adjust to individual facets:
"free_x" - x axis limits adjust
"free_y" - y axis limits adjust

Set **labeller** to adjust facet label:

```
t + facet_grid(cols = vars(fl), labeller = label_both)
fl: c fl: d fl: e fl: p fl: r

t + facet_grid(rows = vars(fl),
labeller = label_bquote(alpha ^ .(fl)))
alpha^c alpha^d alpha^e alpha^p alpha^r
```

Labels and Legends

Use **labs()** to label the elements of your plot.

t + labs(x = "New x axis label", y = "New y axis label", title = "Add a title above the plot", subtitle = "Add a subtitle below title", caption = "Add a caption below plot", alt = "Add alt text to the plot", <AES> = "New <AES> legend title")

t + annotate(geom = "text", x = 8, y = 9, label = "A")
Places a geom with manually selected aesthetics.

p + guides(x = guide_axis(n.dodge = 2)) Avoid crowded or overlapping labels with `guide_axis(n.dodge or angle)`.

n + guides(fill = "none") Set legend type for each aesthetic: colorbar, legend, or none (no legend).

n + theme(legend.position = "bottom")
Place legend at "bottom", "top", "left", or "right".

n + scale_fill_discrete(name = "Title", labels = c("A", "B", "C", "D", "E"))
Set legend title and labels with a scale function.

Zooming

Without clipping (preferred):
t + coord_cartesian(xlim = c(0, 100), ylim = c(10, 20))
With clipping (removes unseen data points):
t + xlim(0, 100) + ylim(10, 20)
t + scale_x_continuous(limits = c(0, 100)) + scale_y_continuous(limits = c(0, 100))