Visual Vocabulary

Deviation

Emphasise variations (+/-) from a fixed reference point. Typically the reference point is zero but it can also be a target or a long-term average. Can also be used to show sentiment (positive/neutral/negative).

Example FT uses
Trade surplus/deficit, climate change

Diverging bar

A simple standard bar chart that can handle both negative and positive magnitude values.

Diverging stacked bar

Perfect for presenting survey results which involve sentiment (eg disagree/neutral/agree).

Spine

Splits a single value into two contrasting components (eg male/female).

Surplus/deficit filled line

The shaded area of these charts allows a balance to be shown – either against a baseline or between two series.

Correlation

Show the relationship between two or more variables. Be mindful that, unless you tell them otherwise, many readers will assume the relationships you show them to be causal (i.e. one causes the other).

Example FT uses
Inflation and unemployment, income and life expectancy

Scatterplot

The standard way to show the relationship between two continuous variables, each of which has its own axis.

Column + line timeline

A good way of showing the relationship between an amount (columns) and a rate (line).

Connected scatterplot

Usually used to show how the relationship between 2 variables has changed over time.

Bubble

Like a scatterplot, but adds additional detail by sizing the circles according to a third variable.

XY heatmap

A good way of showing the patterns between 2 categories of data, less effective at showing fine differences in amounts.

Ranking

Use where an item's position in an ordered list is more important than its absolute or relative value. Don't be afraid to highlight the points of interest.

Example FT uses
Wealth, deprivation, league tables, constituency election results

Ordered bar

Standard bar charts display the ranks of values much more easily when sorted into order.

Ordered column

See above.

Ordered proportional symbol

Use when there are big variations between values and/or seeing fine differences between data is not so important.

Dot strip plot

Dots placed in order on a strip are a space-efficient method of laying out ranks across multiple categories.

Slope

Perfect for showing how ranks have changed over time or vary between categories.

Lollipop

Lollipops draw more attention to the data value than standard bar/column and can also show rank and value effectively.

Bump

Effective for showing changing rankings across multiple dates. For large datasets, consider grouping lines using colour.

Distribution

Show values in a dataset and how often they occur. The shape (or 'skew') of a distribution can be a memorable way of highlighting the lack of uniformity or equality in the data.

Example FT uses
Income distribution, population (age/sex) distribution, revealing inequality

Histogram

The standard way to show a statistical distribution - keep the gaps between columns small to highlight the 'shape' of the data.

Dot plot

A simple way of showing the change or range (min/max) of data across multiple categories.

Dot strip plot

Good for showing individual values in a distribution, can be a problem when too many dots have the same value.

Barcode plot

Like dot strip plots, good for displaying all the data in a table, they work best when highlighting individual values.

Boxplot

Summarise multiple distributions by showing the median (centre) and range of the data

Violin plot

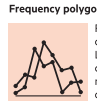
Similar to a box plot but more effective with complex distributions (data that cannot be summarised with simple average).

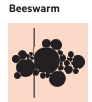
Population pyramid

A standard way for showing the age and sex breakdown of a population distribution; effectively, back to back histograms.

Cumulative curve

A good way of showing how unequal a distribution is: y axis is always cumulative frequency, x axis is always a measure.

Frequency polygons

For displaying multiple distributions of data. Like a regular line chart, best limited to a maximum of 3 or 4 datasets.

Beeswarm

Use to emphasise individual points in a distribution. Points can be sized to an additional variable. Best with medium-sized datasets

Change over Time

Give emphasis to changing trends. These can be short (intra-day) movements or extended series traversing decades or centuries: Choosing the correct time period is important to provide suitable context for the reader.

Example FT uses
Share price movements, economic time series, sectoral changes in a market

Line

The standard way to show a changing time series. If data are irregular, consider markers to represent data points.

Column

Columns work well for showing change over time - but usually best with only one series of data at a time.

Column + line timeline

A good way of showing the relationship over time between an amount (columns) and a rate (line).

Slope

Good for showing changing data as long as the data can be simplified into 2 or 3 points without missing a key part of story.

Area chart

Use with care – these are good at showing changes to total, but seeing change in components can be very difficult.

Candlestick

Usually focused on day-to-day activity, these charts show opening/closing and high/low points of each day.

Fan chart (projections)

Use to show the uncertainty in future projections - usually this grows the further forward to projection.

Connected scatterplot

A good way of showing changing data for two variables whenever there is a relatively clear pattern of progression.

Calendar heatmap

A great way of showing temporal patterns (daily, weekly, monthly) – at the expense of showing precision in quantity.

Priestley timeline

Great when date and duration are key elements of the story in the data.

Circle timeline

Good for showing discrete values of varying size across multiple categories (eg earthquakes by continent).

Vertical timeline

Presents time on the Y axis. Good for displaying detailed time series that work especially well when scrolling on mobile.

Seismogram

Another alternative to the circle timeline for showing series where there are big variations in the data.

Streamgraph

A type of area chart; use when seeing changes in proportions over time is more important than individual values

Magnitude

Show size comparisons. These can be relative (just being able to see larger/bigger) or absolute (need to see fine differences). Usually these show a 'counted' number (for example, barrels, dollars or people) rather than a calculated rate or per cent.

Example FT uses
Commodity production, market capitalisation, volumes in general

Column

The standard way to compare the size of things. Must always start at 0 on the axis.

Bar

See above. Good when the data are not time series and labels have long category names.

Paired column

As per standard column but allows for multiple series. Can become tricky to read with more than 2 series.

Paired bar

See above.

Marimekko

A good way of showing the size and proportion of data at the same time – as long as the data are not too complicated.

Proportional symbol

Use when there are big variations between values and/or seeing fine differences between data is not so important.

Isotype (pictogram)

Excellent solution in some instances – use only with whole numbers (do not slice off an arm to represent a decimal).

Lollipop

Lollipop charts draw more attention to the data value than standard bar/column – but not have to start at zero (but preferable).

Radar

A space-efficient way of showing value of multiple variables – but make sure they are organised in a way that makes sense to reader.

Parallel coordinates

An alternative to radar charts – again, the arrangement of the variables is important. Usually benefits from highlighting values.

Bullet

Good for showing a measurement against the context of a target or performance range.

Grouped symbol

An alternative to bar/column charts when being able to count data or highlight individual elements is useful.

Part-to-whole

Show how a single entity can be broken down into its component elements. If the reader's interest is solely in the size of the components, consider a magnitude-type chart instead.

Example FT uses
Fiscal budgets, company structures, national election results

Stacked column/bar

A simple way of showing part-to-whole relationships but can be difficult to read with more than a few components.

Marimekko

A good way of showing the size and proportion of data at the same time – as long as the data are not too complicated.

Pie

A common way of showing part-to-whole data – but be aware that it's difficult to accurately compare the size of the segments.

Donut

Similar to a pie chart – but the centre can be a good way of making space to include more information about the data (eg total).

Treemap

Use for hierarchical part-to-whole relationships; can be difficult to read when there are many small segments.

Voronoi

A way of turning points into areas – any point within each area is closer to the central point than any other centroid.

Arc

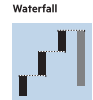
A hemicycle, often used for visualising parliamentary composition by number of seats.

Gridplot

Good for showing % information, they work best when used on whole numbers and work well in small multiple layout form.

Venn

Generally only used for schematic representation.

Waterfall

Can be useful for showing part-to-whole relationships where some of the components are negative.

Spatial

Aside from locator maps only used when precise locations or geographical patterns in data are more important to the reader than anything else.

Example FT uses
Population density, natural resource locations, natural disaster risk/impact, catchment areas, variation in election results

Basic choropleth (rate/ratio)

The standard approach for putting data on a map – should always be rates rather than totals and use a sensible base geography.

Proportional symbol (count/magnitude)

Use for totals rather than rates – be wary that small differences in data will be hard to see.

Flow map

For showing unambiguous movement across a map.

Contour map

For showing areas of equal value on a map. Can use deviation colour schemes for showing +/- values

Equalised cartogram

Converting each unit on a map to a regular and equally-sized shape – good for representing voting regions with equal value.

Scaled cartogram (value)

Stretching and shrinking a map so that each area is sized according to a particular value.

Dot density

Used to show the location of individual events/locations – make sure to annotate any patterns the reader should see.

Heat map

Grid-based data values mapped with an intensity colour scale. As choropleth map – but not snapped to an admin/political unit.

Flow

Show the reader volumes or intensity of movement between two or more states or conditions. These might be logical sequences or geographical locations.

Example FT uses
Movement of funds, trade, migrants, lawsuits, information; relationship graphs.

Sankey

Shows changes in flows from one condition to at least one other; good for tracing the eventual outcome of a complex process.

Waterfall

Designed to show the sequencing of data through a flow process, typically budgets. Can include +/- components.

Chord

A complex but powerful diagram which can illustrate 2-way flows (and net winner) in a matrix.

Network

Used for showing the strength and inter-connectedness of relationships of varying types.

Read the FT for free

Add fresh examples and insights to your work, make more informed career choices, and feel more confident managing money with the Financial Times.

Teachers can register your school and students can create a login by scanning here.

ft.com/schoolsarefree

FT graphic: Alan Smith; Chris Campbell; Ian Bott; Liz Faunce; Graham Parrish; Billy Ehrenberg-Shannon; Paul McCallum; Martin Stabe
Inspired by the Graphic Continuum by Jon Schwabish and Severino Ribecca

FT FINANCIAL TIMES

© Financial Times 2016-2019. This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Royal Geographical Society with IBG
Advancing geography and geographical learning

matplotlib Cheat sheet

Version 3.5.0

Quick start

API

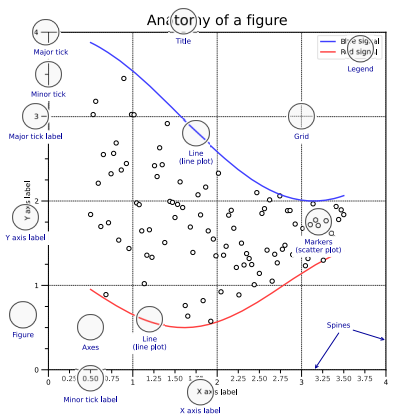
```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt


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

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

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

Anatomy of a figure



Subplots layout

API

```
subplot[s](rows,cols,…)
fig, axs = 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
- https://gitter.im/matplotlib/matplotlib
- twitter.com/matplotlib
- Matplotlib users mailing list

Basic plots

API

```
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,…)
```
Z, cmap, interpolation, extent, origin

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

```
pcolormesh([X],[Y],Z,…)
```
X, Y, **Z**, vmin, vmax, cmap

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

```
pie(X,…)
```
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

API

```
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, gridsize, bins

Scales

API

```
ax.set_[xy]scale(scale,…)
```
linear any values
log values > 0
symlog any values
logit 0 < values < 1

Projections

API

```
subplot(…,projection=p)
```
p='polar'
p='3d'

p=ccrs.Orthographic()
import cartopy.crs as ccrs

Lines

API

linestyle or ls

"-" ":" "--" "-." (0,(0.01,2))

capstyle or dash_capstyle

"butt" "round" "projecting"

Markers

API

markevery

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

Colors

API

'Cn'
'x'
'name'
(R,G,B[,A])
'#RRGGBB[AA]'
'x.y'

Colormaps

API

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

Uniform: viridis, magma, plasma

Sequential: Greys, YlOrBr, Wistia

Diverging: Spectral, coolwarm, RdGy

Qualitative: tab10, tab20

Cyclic: twilight

Tick locators

API

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

ticker.NullLocator()
ticker.MultipleLocator(0.5)
ticker.FixedLocator([0, 1, 5])
ticker.LinearLocator(numticks=3)
ticker.IndexLocator(base=0.5, offset=0.25)
ticker.AutoLocator()
ticker.MaxNLocator(n=4)
ticker.LogLocator(base=10, numticks=15)
```

Tick formatters

API

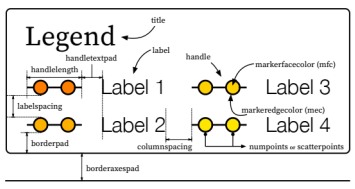
```
from matplotlib import ticker
ax.[xy]axis.set_[minor|major]_formatter(formatter)

ticker.NullFormatter()
ticker.FixedFormatter(['zero', 'one', 'two', …])
ticker.FuncFormatter(lambda x, pos: "[%.2f]" % x)
ticker.FormatStrFormatter('>%d<')
ticker.ScalarFormatter()
ticker.StrMethodFormatter('{x}')
ticker.PercentFormatter(xmax=5)
```

Ornaments

API

```
ax.legend(…)
```
handles, labels, loc, title, frameon




```
ax.colorbar(…)
```
mappable, ax, cax, orientation

```
ax.annotate(…)
```
text, **xy**, **xytext**, xycoords, textcoords, arrowprops

Event handling

API

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

Animation

API

```
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

API

```
plt.style.use(style)
```

default, classic, grayscale, ggplot, seaborn, fast, bmh, Solarize_Light2, seaborn-notebook

Quick reminder

```
ax.grid()
ax.set_[xy]lim(vmin, vmax)
ax.set_[xy]label(label)
ax.set_[xy]ticks(ticks, [labels])
ax.set_[xy]ticklabels(labels)
ax.set_title(title)
ax.tick_params(width=10, …)
ax.set_axis_[on|off]()

fig.suptitle(title)
fig.tight_layout()
plt.gcf(), plt.gca()
mpl.rc('axes', linewidth=1, …)
[fig|ax].patch.set_alpha(0)
text=r'$\frac{-e^{i\pi}}{2^n}$'
```

Keyboard shortcuts

API

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

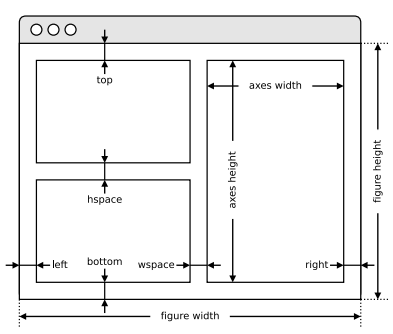
READ

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 mislead the reader
8. Avoid "chartjunk"
9. Message trumps beauty
10. Get the right tool

Axes adjustments

API

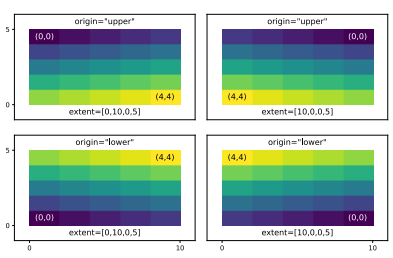
plt.**subplots_adjust**(…)



Extent & origin

API

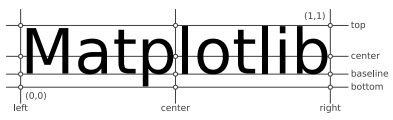
ax.**imshow**(extent=…, origin=…)



Text alignments

API

ax.**text**(…, ha=… , va=…, …)



Text parameters

API

ax.**text**(…, family=…, size=…, weight=…)
ax.**text**(…, fontproperties=…)

The quick brown fox — xx-large (1.73)
The quick brown fox — x-large (1.44)
The quick brown fox — large (1.20)
The quick brown fox — medium (1.00)
The quick brown fox — small (0.83)
The quick brown fox — x-small (0.69)
The quick brown fox — xx-small (0.58)

The quick brown fox jumps over the lazy dog — black (900)
The quick brown fox jumps over the lazy dog — bold (700)
The quick brown fox jumps over the lazy dog — semibold (600)
The quick brown fox jumps over the lazy dog — normal (400)
The quick brown fox jumps over the lazy dog — ultralight (100)

The quick brown fox jumps over the lazy dog — monospace
The quick brown fox jumps over the lazy dog — serif
The quick brown fox jumps over the lazy dog — sans
The quick brown fox jumps over the lazy dog — cursive

The quick brown fox jumps over the lazy dog — italic
The quick brown fox jumps over the lazy dog — normal

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG — small-caps
The quick brown fox jumps over the lazy dog — normal

Uniform colormaps

viridis, plasma, inferno, magma, cividis

Sequential colormaps

Greys, Purples, Blues, Greens, Oranges, Reds, YlOrBr, YlOrRd, OrRd, PuRd, RdPu, BuPu, GnBu, PuBu, YlGnBu, PuBuGn, BuGn, YlGn

Diverging colormaps

PiYG, PRGn, BrBG, PuOr, RdGy, RdBu, RdYlBu, RdYlGn, Spectral, coolwarm, bwr, seismic

Qualitative colormaps

Pastel1, Pastel2, Paired, Accent, Dark2, Set1, Set2, Set3, tab10, tab20, tab20b, tab20c

Miscellaneous colormaps

terrain, ocean, cubehelix, rainbow, twilight

Color names

API

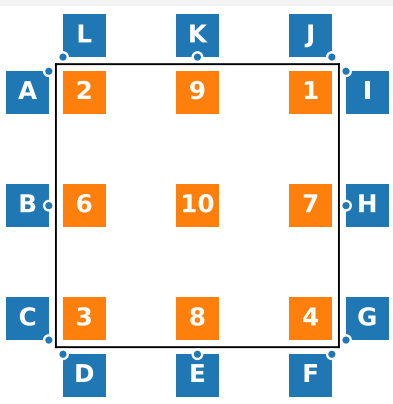
black, k, dimgray, dimgrey, gray, grey, darkgray, darkgrey, silver, lightgray, lightgrey, gainsboro, whitesmoke, w, white, snow, rosybrown, lightcoral, indianred, brown, firebrick, maroon, darkred, r, red, mistyrose, salmon, tomato, darksalmon, coral, orangered, lightsalmon, sienna, seashell, chocolate, saddlebrown, sandybrown, peachpuff, peru, linen, bisque, darkorange, burlywood, antiquewhite, tan, navajowhite, blanchedalmond, papayawhip, moccasin, orange, wheat, oldlace, floralwhite, darkgoldenrod, goldenrod, cornsilk, gold, lemonchiffon, khaki, palegoldenrod, darkkhaki, ivory, beige, lightyellow, lightgoldenrodyellow, olive, y, yellow, olivedrab, yellowgreen, darkolivegreen, greenyellow, chartreuse, lawngreen, honeydew, darkseagreen, palegreen, lightgreen, forestgreen, limegreen, darkgreen, g, green, lime, seagreen, mediumseagreen, springgreen, mintcream, mediumspringgreen, mediumaquamarine, aquamarine, turquoise, lightseagreen, mediumturquoise, azure, lightcyan, paleturquoise, darkslategray, darkslategrey, teal, darkcyan, c, aqua, cyan, darkturquoise, cadetblue, powderblue, lightblue, deepskyblue, skyblue, lightskyblue, steelblue, aliceblue, dodgerblue, lightslategray, lightslategrey, slategray, slategrey, lightsteelblue, cornflowerblue, royalblue, ghostwhite, lavender, midnightblue, navy, darkblue, mediumblue, b, blue, slateblue, darkslateblue, mediumslateblue, mediumpurple, rebeccapurple, blueviolet, indigo, darkorchid, darkviolet, mediumorchid, thistle, plum, violet, purple, darkmagenta, m, fuchsia, magenta, orchid, mediumvioletred, deeppink, hotpink, lavenderblush, palevioletred, crimson, pink, lightpink

Image interpolation

API

None, none, nearest, bilinear, bicubic, spline16, spline36, hanning, hamming, hermite, kaiser, quadric, catrom, gaussian, bessel, mitchell, sinc, lanczos

Legend placement



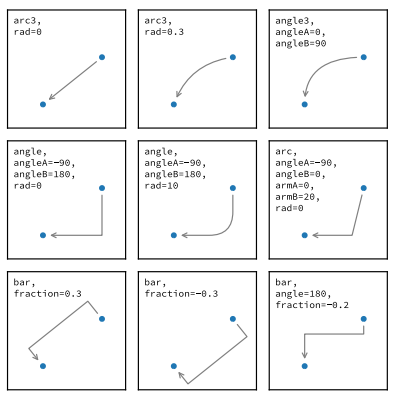
ax.**legend**(loc="string", bbox_to_anchor=(x,y))

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

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

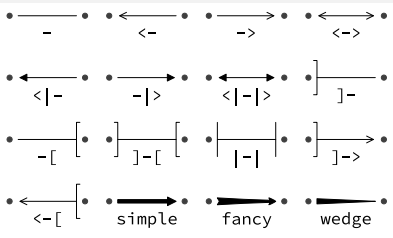
Annotation connection styles

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/an axes?
→ fig.clear() → ax.clear()
… close all figures?
→ plt.close("all")
… remove ticks?
→ ax.set_[xy]ticks([])
… remove tick labels ?
→ ax.set_[xy]ticklabels([])
… rotate tick labels ?
→ ax.tick_params(axis="x", 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?
→ fig.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(…), canvas.draw()`	slow
`im.set_data(…), 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

Numpy Cheat Sheet

Quick Reference For Essential Functions

NumPy Basics

1. Installing And Importing NumPy

You can install NumPy from the command line using the command below:

```
pip install numpy
```

Once it's installed, import it into your code.

```
import numpy as np
```

Keep in mind that you can use any other name apart from **np**. However, **np** is the standard NumPy import convention used by most developers and data scientists.

2. Array Creation

Creating arrays in NumPy is simple and straightforward. You can create arrays from lists or tuples using the numpy.array() function:

```
import numpy as np
a = np.array([1, 2, 3]) # Creates a 1D array
b = np.array([(1, 2, 3), (4, 5, 6)]) # Creates a 2D array
```

You can also generate arrays of specific shapes and values using various functions:

- **np.zeros():** Creates an array filled with zeros
- **np.ones():** Creates an array filled with ones
- **np.identity():** Creates an identity matrix array
- **np.empty():** Creates an array without initializing its elements to any particular value
- **np.arange():** Creates an array with regularly spaced values between a start and end value
- **np.linspace():** Creates an array with a specified number of evenly spaced values between a start and end value

Note: You cannot generate an empty array in NumPy. Each NumPy array has a fixed, immutable size and each element in the array must be filled in when the array is created.

The **np.empty()** function creates the required array shape and fills it with random values. The default method creates an array of random floats.

You can create a different array datatype using the **dtype** parameter.

3. Array Attributes

NumPy arrays have several attributes that provide useful information about the array. Let's look at some of them:

- **ndarray.shape:** Returns the dimensions of the array as a tuple (rows, columns)
- **ndarray.ndim:** Returns the number of dimensions in the array
- **ndarray.size:** Returns the total number of elements in the array
- **ndarray.dtype:** Returns the data type of the array elements

To access these attributes, use the dot notation, like so:

```
a = np.array([(1, 2, 3), (4, 5, 6)])
#Print out the array shape
print(a.shape) # Output: (2, 3)
```

4. Data Types

NumPy provides several data types to store data in arrays, such as integer, string, float, boolean and complex. By default, NumPy tries to deduce the data type based on the input elements.

However, you can also explicitly specify the data type using the dtype keyword. For example:

```
import numpy as np
a = np.array([1, 2, 3], dtype=float)
# Creates an array of floats
```

Common NumPy data types include:

- np.int32: 32-bit integer
- np.int64: 64-bit integer
- np.float32: 32-bit floating-point number
- np.float64: 64-bit floating-point number
- np.complex: Complex number, represented by two 64-bit floating-point numbers

You can also convert arrays from one data type to another. In this example, here's how we can convert the Integer array a into a Boolean array **arr** using the **np.array()** method.

Understanding these basic concepts of NumPy will allow you to effectively work with arrays and perform a variety of mathematical NumPy operations. For example, you can check out our video on How To Transform and Code Addresses In Python.

Youtube link

In it, we used Python Pandas and NumPy data types to geocode home addresses.

Array Manipulation

1. Reshaping

Reshaping an array in NumPy is a common task you'll perform. You might need to change the shape of your array to match the requirements of a function or an algorithm.

To reshape an array, use the reshape() function:

```
arr = np.array([1, 2, 3, 4, 5, 6])
new_arr = arr.reshape(2, 3)
```

This will convert your one-dimensional array into a two-dimensional array with 2 rows and 3 columns.

Note: Make sure the new shape you provide has the same size (number of array elements) as the original array.

2. Copying

You can copy the elements in one NumPy array to another using the copy() method. You should note that using the assignment operator '=' creates a shallow copy.

```
#Creating a shallow copy of a NumPy array
a = np.array([9, 6, 12, 16, 20])
b = a
b[0] = 19
print(a) #Output:[19, 6, 12, 16, 20]
print(b) #Output:[19, 6, 12, 16, 20]
```

The new array only references the old array in the system's memory. They contain the same elements and they are not independent of each other.

By using the deep copy, you create a new NumPy array that contains the same data as the old one while being independent of it.

```
#Creating a deep copy of a NumPy array
a = np.array([9, 6, 12, 16, 20])
b = np.copy(a)
b[0] = 19
print(a) #Output:[9, 6, 12, 16, 20]
print(b) #Output:[19, 6, 12, 16, 20]
```

3. Concatenation

Occasionally, you may need to merge two arrays into a single one. In NumPy, you can use the concatenate() function to join arrays along an existing axis:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = np.concatenate((arr1, arr2))
```

This combines **arr1** and **arr2** into a single array. Keep in mind that the arrays being concatenated should have the same shape, except along the specified axis.

4. Splitting

Splitting is the opposite of concatenation. You can divide an array into smaller sub-arrays using the split() function:

```
arr = np.array([1, 2, 3, 4, 5, 6])
result = np.split(arr, 3)
```

This splits the array into 3 equal-sized sub-arrays. Ensure that the number of splits you specify can evenly divide the size of the array along the given axis.

5. Adding/Removing Elements

Adding or removing elements in a NumPy array can be achieved using the append() and delete() functions. You can use the former to append values to the end of the array while the latter deletes the element at a specified index.

Here's an example:

```
arr = np.array([1, 2, 3])
arr = np.append(arr, [4, 5, 6]) # Appends values to the end of the array
arr = np.delete(arr, 0) # Removes the array element on index 0
```

Keep in mind that NumPy arrays have a fixed size. When using append() or delete(), a new array is created, and the original one is not modified.

6. Indexing

You can perform indexing operations on NumPy arrays the same way you'd do them on Python lists or tuples. Let's look at how you can access or change array elements in a given array.

```
arr = np.array([1, 2, 3])
#Returns the array element on index 1
element_2 = b[1]
#Change the array element on index 0
arr[0]= 89
```

7. Slicing

You can also slice NumPy arrays to extract or view a section of the data the same way you'd do Python lists or sets. Let's take a look at an example below:

```
arr1 = np.array([1, 2, 3, 4, 5, 6, 7])
arr2 = np.array([(1, 2, 3, 6, 0), (4, 5, 6, 11, 13)])
# To return the first 3 elements of arr1
print(arr1[0:3]) #Output: [1, 2, 3]
# To return the second row in arr2
b = arr2[1, : ].copy() #Output: [4, 5, 6, 11, 13]
```

Note: Slicing creates a shallow copy that still references the main array. So, any change you make to the sliced data will be applied to the main array and vice versa.

To avoid this, you can use the copy() method to create a deep, independent copy.

Numpy Cheat Sheet
Quick Reference For Essential Functions

Elementary Functions

1. Arithmetic Operations

NumPy offers various math operations on arrays that make them simple and efficient to work with. array mathematics vector math

Some of the operations are:

- **Addition:** numpy.add(x1, x2)
- **Subtraction:** numpy.subtract(x1, x2)
- **Multiplication:** numpy.multiply(x1, x2)
- **Division:** numpy.divide(x1, x2)
- **Modulus:** numpy.mod(x1, x2)
- **Power:** numpy.power(x1, x2)
- **Square root:** numpy.sqrt(x)

Note: When using these operations, the two arrays must be the same shape. If not, you'll run into errors.

There is an exception for certain arrays thanks to a NumPy feature called broadcasting. We'll cover that in a later section.

You can perform these operations element-wise on the arrays, which makes them highly efficient for large-scale data manipulation.

2. Trigonometry

Trigonometric functions play a significant role in various mathematical and scientific computations. NumPy provides a wide range of trigonometric functions.

Some of the essential functions are:

- **Sine:** numpy.sin(x)
- **Cosine:** numpy.cos(x)
- **Tangent:** numpy.tan(x)
- **Arcsine:** numpy.arcsin(x)
- **Arccosine:** numpy.arccos(x)
- **Arctangent:** numpy.arctan(x)

These functions work seamlessly with arrays, making it easier for you to perform vectorized computations on large datasets.

3. Exponents And Logarithms

Exponents and logarithms are crucial for various numerical operations. NumPy provides an extensive collection of functions for dealing with exponents and logarithms.

Some of the primary functions are:

- **Exponential:** numpy.exp(x)
- **Logarithm(base e):** numpy.log(x)
- **Logarithm(base 10):** numpy.log10(x)
- **Logarithm(base 2):** numpy.log2(x)

Utilizing these functions, you can quickly perform complex mathematical operations on each element in the array. This makes your data analysis tasks more accessible and efficient.

Array Analysis

1. Aggregate Functions

NumPy provides several aggregate functions that allow you to perform operations on arrays, such as summing all their elements, finding the minimum or maximum value, and more:

- **sum:** np.sum(your_array) - Calculate the sum of all the elements in the array.
- **min:** np.min(your_array) - Find the minimum array element.
- **max:** np.max(your_array) - Find the maximum array element.
- **mean:** np.mean(your_array) - Calculate the mean of the values in the array.
- **median:** np.median(your_array) - Find the median of the values in the array.

2. Statistical Functions

NumPy also has a variety of statistical functions to help you analyze data:

- **std:** np.std(your_array) - Calculate the standard deviation of the values in the array.
- **var:** np.var(your_array) - Calculate the variance of the values in the array.
- **corrcoef:** np.corrcoef(your_array) - Calculate the correlation coefficient of the array.

3. Searching

Searching in NumPy arrays can be done using various methods:

- **argmin:** np.argmin(your_array) - Find the index of the minimum array element.
- **argmax:** np.argmax(your_array) - Find the index of the maximum array element.
- **where:** np.where(condition) - Return the indices of elements in the array that satisfy the given condition.

4. Sorting

You can sort the elements in your array using the following functions:

- **sort:** np.sort(your_array) - Sort the elements in the array in ascending order.
- **argsort:** np.argsort(your_array) - Returns the indices that would sort the array.

With these functions and techniques, you can conveniently analyze and manipulate your NumPy arrays to uncover valuable insights and support your data analysis efforts.

Advanced Functions

1. Broadcasting

Broadcasting is a powerful NumPy feature that allows you to perform operations on arrays with different shapes and sizes. It works by automatically expanding the dimensions of the smaller array to match the larger array, making it easier to perform element-wise operations.

Here's an example:

```
import numpy as np
A = np.array([1, 2, 3])
B = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
C = A + B
```

Keep these rules in mind when working with broadcasting:

- The dimensions of the arrays must be compatible (either the same size or one of them is 1).
- Broadcasting is applied from the trailing dimensions and works towards the leading dimensions.

2. Linear Algebra

NumPy provides several linear algebra functions that can be useful when working with multidimensional arrays. Some of these functions include:

- **np.dot(A, B):** Computes the dot product of two arrays.
- **np.linalg.inv(A):** Computes the inverse of a square matrix.
- **np.linalg.eig(A):** Computes the eigenvalues and eigenvectors of a square matrix.
- **np.linalg.solve(A, B):** Solves a linear system of equations, where A is the coefficient matrix and B is the constant matrix.

Remember to always check if your matrices are compatible before performing these operations.

Input And Output

1. Saving And Loading Arrays

To save an array, you can use NumPy's **np.save()** function. This function takes the filename and the array as its two main arguments.

```
import numpy as np
arr = np.array([1, 2, 3])
np.save('my_array.npy', arr)
```

To load the saved array, use the **np.load()** function, providing the filename as the argument.

```
loaded_array = np.load('my_array.npy')
print(loaded_array)
# Output: array([1, 2, 3])
```

You can also save and load multiple arrays using the **np.save()** and **np.load()** functions.

2. Reading And Writing To Text Files

NumPy provides functions to read and write text files with arrays, such as **np.loadtxt()** and **np.savetxt()**. You can use these functions to save and load data from file formats like a txt or CSV file.

To read a text file into an array, use the **np.loadtxt()** function. It takes the filename as its main argument and also supports optional arguments for specifying delimiter, dtype, and more.

```
arr_from_txt = np.loadtxt('data.txt', delimiter=',')
print(arr_from_txt)
```

To read the data from a CSV file, you can also use the **np.loadtxt()** function. However, make sure the delimiter is always set to the comma, ",".

To write an array to a text file, use the **np.savetxt()** function. This function takes the filename and the array as its two main arguments, followed by optional arguments, such as delimiter and header.

```
arr_to_txt = np.array([[1, 2, 3], [4, 5, 6]])
np.savetxt('output_data.txt', arr_to_txt, delimiter=',')
```

These input and output functions allow you to efficiently work with arrays and text files in your data processing and manipulation tasks using NumPy.

Data Wrangling

with pandas Cheat Sheet
http://pandas.pydata.org

Pandas API Reference Pandas User Guide

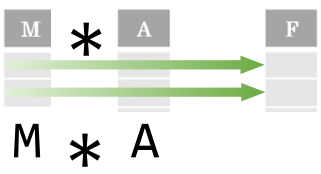
Tidy Data – A foundation for wrangling in pandas

In a tidy data set:

Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements pandas's vectorized operations. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.



Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4, 5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
       index = [1, 2, 3])
```
Specify values for each column.

```
df = pd.DataFrame(
     [[4, 7, 10],
      [5, 8, 11],
      [6, 9, 12]],
     index=[1, 2, 3],
     columns=['a', 'b', 'c'])
```
Specify values for each row.

		a	b	c
N	v			
D	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
        [('d', 1), ('d', 2),
         ('e', 2)], names=['n', 'v']))
```
Create DataFrame with a MultiIndex

Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

```
df = (pd.melt(df)
       .rename(columns={
               'variable':'var',
               'value':'val'})
       .query('val >= 200')
     )
```

Reshaping Data – Change layout, sorting, reindexing, renaming

pd.melt(df)
Gather columns into rows.

df.pivot(columns='var', values='val')
Spread rows into columns.

pd.concat([df1,df2])
Append rows of DataFrames

pd.concat([df1,df2], axis=1)
Append columns of DataFrames

df.sort_values('mpg')
Order rows by values of a column (low to high).

df.sort_values('mpg', ascending=False)
Order rows by values of a column (high to low).

df.rename(columns = {'y':'year'})
Rename the columns of a DataFrame

df.sort_index()
Sort the index of a DataFrame

df.reset_index()
Reset index of DataFrame to row numbers, moving index to columns.

df.drop(columns=['Length', 'Height'])
Drop columns from DataFrame

Subset Observations - rows

df[df.Length > 7]
Extract rows that meet logical criteria.

df.drop_duplicates()
Remove duplicate rows (only considers columns).

df.sample(frac=0.5)
Randomly select fraction of rows.

df.sample(n=10) Randomly select n rows.

df.nlargest(n, 'value')
Select and order top n entries.

df.nsmallest(n, 'value')
Select and order bottom n entries.

df.head(n)
Select first n rows.

df.tail(n)
Select last n rows.

Subset Variables - columns

df[['width', 'length', 'species']]
Select multiple columns with specific names.

df['width'] *or* **df.width**
Select single column with specific name.

df.filter(regex='*regex*')
Select columns whose name matches regular expression *regex*.

Using query

query() allows Boolean expressions for filtering rows.

```
df.query('Length > 7')
df.query('Length > 7 and Width < 8')
df.query('Name.str.startswith("abc")',
         engine="python")
```

Subsets - rows and columns

Use **df.loc[]** and **df.iloc[]** to select only rows, only columns or both.
Use **df.at[]** and **df.iat[]** to access a single value by row and column.
First index selects rows, second index columns.

df.iloc[10:20]
Select rows 10-20.

df.iloc[:, [1, 2, 5]]
Select columns in positions 1, 2 and 5 (first column is 0).

df.loc[:, 'x2':'x4']
Select all columns between x2 and x4 (inclusive).

df.loc[df['a'] > 10, ['a', 'c']]
Select rows meeting logical condition, and only the specific columns .

df.iat[1, 2] Access single value by index

df.at[4, 'A'] Access single value by label

Logic in Python (and pandas)			
<	Less than	!=	Not equal to
>	Greater than	df.column.isin(*values*)	Group membership
==	Equals	pd.isnull(*obj*)	Is NaN
<=	Less than or equals	pd.notnull(*obj*)	Is not NaN
>=	Greater than or equals	&,|,~,^,df.any(),df.all()	Logical and, or, not, xor, any, all

regex (Regular Expressions) Examples	
'\.'	Matches strings containing a period '.'
'Length$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]$'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
'^(?!Species$).*'	Matches strings except the string 'Species'

Cheatsheet for pandas (http://pandas.pydata.org/ originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

Summarize Data

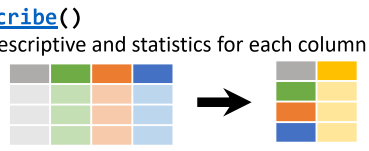
df['w'].value_counts()
Count number of rows with each unique value of variable

len(df)
# of rows in DataFrame.

df.shape
Tuple of # of rows, # of columns in DataFrame.

df['w'].nunique()
# of distinct values in a column.

df.describe()
Basic descriptive and statistics for each column (or GroupBy).



pandas provides a large set of summary functions that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

sum()
Sum values of each object.

count()
Count non-NA/null values of each object.

median()
Median value of each object.

quantile([0.25,0.75])
Quantiles of each object.

apply(*function*)
Apply function to each object.

min()
Minimum value in each object.

max()
Maximum value in each object.

mean()
Mean value of each object.

var()
Variance of each object.

std()
Standard deviation of each object.

Group Data

df.groupby(by="col")
Return a GroupBy object, grouped by values in column named "col".

df.groupby(level="ind")
Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group.
Additional GroupBy functions:

size()
Size of each group.

agg(*function*)
Aggregate group using function.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

shift(1)
Copy with values shifted by 1.

rank(method='dense')
Ranks with no gaps.

rank(method='min')
Ranks. Ties get min rank.

rank(pct=True)
Ranks rescaled to interval [0, 1].

rank(method='first')
Ranks. Ties go to first value.

shift(-1)
Copy with values lagged by 1.

cumsum()
Cumulative sum.

cummax()
Cumulative max.

cummin()
Cumulative min.

cumprod()
Cumulative product.

Windows

df.expanding()
Return an Expanding object allowing summary functions to be applied cumulatively.

df.rolling(n)
Return a Rolling object allowing summary functions to be applied to windows of length n.

Handling Missing Data

df.dropna()
Drop rows with any column having NA/null data.

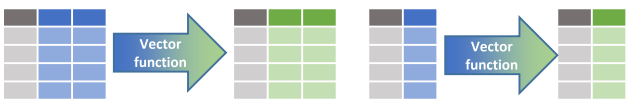
df.fillna(value)
Replace all NA/null data with value.

Make New Columns

df.assign(Area=lambda df: df.Length*df.Height)
Compute and append one or more new columns.

df['Volume'] = df.Length*df.Height*df.Depth
Add single column.

pd.qcut(df.col, n, labels=False)
Bin column into n buckets.



pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

max(axis=1)
Element-wise max.

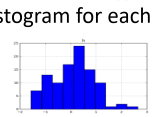
clip(lower=-10,upper=10)
Trim values at input thresholds

min(axis=1)
Element-wise min.

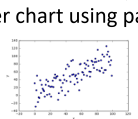
abs()
Absolute value.

Plotting

df.plot.hist()
Histogram for each column



df.plot.scatter(x='w',y='h')
Scatter chart using pairs of points



Combine Data Sets

adf

x1	x2
A	1
B	2
C	3

+

bdf

x1	x3
A	T
B	F
D	T

=

Standard Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NaN

pd.merge(adf, bdf, how='left', on='x1')
Join matching rows from bdf to adf.

x1	x2	x3
A	1.0	T
B	2.0	F
D	NaN	T

pd.merge(adf, bdf, how='right', on='x1')
Join matching rows from adf to bdf.

x1	x2	x3
A	1	T
B	2	F

pd.merge(adf, bdf, how='inner', on='x1')
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NaN
D	NaN	T

pd.merge(adf, bdf, how='outer', on='x1')
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

adf[adf.x1.isin(bdf.x1)]
All rows in adf that have a match in bdf.

x1	x2
C	3

adf[~adf.x1.isin(bdf.x1)]
All rows in adf that do not have a match in bdf.

ydf

x1	x2
A	1
B	2
C	3

+

zdf

x1	x2
B	2
C	3
D	4

=

Set-like Operations

x1	x2
B	2
C	3

pd.merge(ydf, zdf)
Rows that appear in both ydf and zdf (Intersection).

x1	x2
A	1
B	2
C	3
D	4

pd.merge(ydf, zdf, how='outer')
Rows that appear in either or both ydf and zdf (Union).

x1	x2
A	1

pd.merge(ydf, zdf, how='outer', indicator=True)
.query('_merge == "left_only"')
.drop(columns=['_merge'])
Rows that appear in ydf but not zdf (Setdiff).

Cheatsheet for pandas (http://pandas.pydata.org/) originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet