Linux Command Line Cheat Sheet

by Dave Child (DaveChild) via cheatography.com/1/cs/49/

Bash Commands

uname -a	Show system and kernel
head -n1 /etc/issue	Show distribution
mount	Show mounted filesystems
date	Show system date
uptime	Show uptime
whoami	Show your username
man *command*	Show manual for *command*

Bash Shortcuts

CTRL-c	Stop current command
CTRL-z	Sleep program
CTRL-a	Go to start of line
CTRL-e	Go to end of line
CTRL-u	Cut from start of line
CTRL-k	Cut to end of line
CTRL-r	Search history
!!	Repeat last command
!*abc*	Run last command starting with *abc*
!*abc*:p	Print last command starting with *abc*
!$	Last argument of previous command
ALT-.	Last argument of previous command
!*	All arguments of previous command
^*abc*^*123*	Run previous command, replacing *abc* with *123*

Bash Variables

env	Show environment variables
echo *$NAME*	Output value of *$NAME* variable

Bash Variables (cont)

export *NAME=value*	Set *$NAME* to *value*
$PATH	Executable search path
$HOME	Home directory
$SHELL	Current shell

IO Redirection

cmd < *file*

> Input of *cmd* from *file*

cmd1 <(*cmd2*)

> Output of *cmd2* as file input to *cmd1*

cmd > *file*

> Standard output (stdout) of *cmd* to *file*

cmd > /dev/null

> Discard stdout of *cmd*

cmd >> *file*

> Append stdout to *file*

cmd 2> *file*

> Error output (stderr) of *cmd* to *file*

cmd 1>&2

> stdout to same place as stderr

cmd 2>&1

> stderr to same place as stdout

cmd &> *file*

> Every output of *cmd* to *file*

cmd refers to a command.

Pipes

cmd1 | *cmd2*

> stdout of *cmd1* to *cmd2*

cmd1 |& *cmd2*

> stderr of *cmd1* to *cmd2*

Command Lists

cmd1 ; *cmd2*

> Run *cmd1* then *cmd2*

cmd1 && *cmd2*

> Run *cmd2* if *cmd1* is successful

cmd1 || *cmd2*

> Run *cmd2* if *cmd1* is not successful

cmd &

> Run *cmd* in a subshell

Directory Operations

pwd	Show current directory
mkdir *dir*	Make directory *dir*
cd *dir*	Change directory to *dir*
cd ..	Go up a directory
ls	List files

ls Options

-a	Show all (including hidden)
-R	Recursive list
-r	Reverse order
-t	Sort by last modified
-S	Sort by file size
-l	Long listing format
-1	One file per line
-m	Comma-separated output
-Q	Quoted output

Search Files

grep *pattern files*	Search for *pattern* in *files*
grep -i	Case insensitive search
grep -r	Recursive search
grep -v	Inverted search
grep -o	Show matched part of file only
find */dir/* -name *name**	Find files starting with *name* in *dir*

By **Dave Child** (DaveChild)
cheatography.com/davechild/
aloneonahill.com

Published 28th October, 2011.
Last updated 29th February, 2020.

Cheatography

Linux Command Line Cheat Sheet

by Dave Child (DaveChild) via cheatography.com/1/cs/49/

Search Files (cont)

find */dir/* -user *name*	Find files owned by *name* in *dir*
find */dir/* -mmin *num*	Find files modifed less than *num* minutes ago in *dir*
whereis *command*	Find binary / source / manual for *command*
locate *file*	Find *file* (quick search of system index)

File Operations

touch *file1*

> Create *file1*

cat *file1 file2*

> Concatenate files and output

less *file1*

> View and paginate *file1*

file *file1*

> Get type of *file1*

cp *file1 file2*

> Copy *file1* to *file2*

mv *file1 file2*

> Move *file1* to *file2*

rm *file1*

> Delete *file1*

head *file1*

> Show first 10 lines of *file1*

tail *file1*

> Show last 10 lines of *file1*

tail -F *file1*

> Output last lines of *file1* as it changes

Watch a Command

watch -n 5 'ntpq -p'

> Issue the 'ntpq -p' command every 5 seconds and display output

Process Management

ps	Show snapshot of processes
top	Show real time processes
kill *pid*	Kill process with id *pid*
pkill *name*	Kill process with name *name*
killall *name*	Kill all processes with names beginning *name*

Nano Shortcuts

Files	
Ctrl-R	Read file
Ctrl-O	Save file
Ctrl-X	Close file
Cut and Paste	
ALT-A	Start marking text
CTRL-K	Cut marked text or line
CTRL-U	Paste text
Navigate File	
ALT-/	End of file
CTRL-A	Beginning of line
CTRL-E	End of line
CTRL-C	Show line number
CTRL-_	Go to line number
Search File	
CTRL-W	Find
ALT-W	Find next
CTRL-\	Search and replace

More nano info at:
http://www.nano-editor.org/docs.php

Screen Shortcuts

screen

> Start a screen session.

screen -r

> Resume a screen session.

Screen Shortcuts (cont)

screen -list

> Show your current screen sessions.

CTRL-A

> Activate commands for screen.

CTRL-A c

> Create a new instance of terminal.

CTRL-A n

> Go to the next instance of terminal.

CTRL-A p

> Go to the previous instance of terminal.

CTRL-A "

> Show current instances of terminals.

CTRL-A A

> Rename the current instance.

More screen info at:
http://www.gnu.org/software/screen/

File Permissions

chmod 775 *file*

> Change mode of *file* to 775

chmod -R 600 *folder*

> Recursively chmod *folder* to 600

chown *user.group file*

> Change *file* owner to *user* and group to *group*

File Permission Numbers

First digit is owner permission, second is group and third is everyone.

Calculate permission digits by adding numbers below.

4	read (r)
2	write (w)
1	execute (x)

By **Dave Child** (DaveChild)
cheatography.com/davechild/
aloneonahill.com

Published 28th October, 2011.
Last updated 29th February, 2020.

datacamp

julia

Julia Basics Cheat Sheet

Learn Julia online at www.DataCamp.com

> Accessing help

```
# Access help mode with an empty ?
?

# Get help on a function with ?functionname
?first

# Search for help on a topic with ?topic
?function
```

> Comments

```
# This is a single-line comment
```

```
#= This is a
multi-line comment =#
```

> Information about objects

```
# Get the type of an object with typeof() – Example returns Int64
typeof(20)
```

> Using packages

Packages are libraries of pre-written code (by other programmers) that we can add to our Julia installation, which help us solve specific problems. Here's how to install and work with packages in Julia.

```
# Enter package mode with ] to install and work with packages
]

# Install a new package with add
add CSV

# Exit package mode with DELETE
<DEL>

# Load a package with using
using CSV

# Load a package with import without an alias
import CSV

# Load a package with import with an alias
import DataFrames as df
```

> The working directory

The working directory is a file path that Julia will use as the starting point for relative file paths. That is, it's the default location for importing and exporting files. An example of a working directory looks like "/Users/myname/workspace/myproject"

```
# Get current working director with pwd()
pwd()
"/home/programming_languages/julia"

# Set the current directory with cd()
cd("/home/programming_languages/julia/cheatsheets")
```

> Operators

Arithmetic operators

```
# Add two numbers with +
37 + 102

# Subtract two numbers with -
102 - 37

# Multiply two numbers with *
4 * 6

# Divide a number by another with /
21/7

# Integer divide a number with ÷
22 ÷ 7 # This returns 3

# Inverse divide a number with \
5 \ 0 # This is equivalent to 0/5

# Raise to the power using ^
3 ^ 3

# Get the remainder after division with %
22 % 7
```

Assignment operators

```
# Assign a value to an object with =
a = 5

# Add two objects; store in left-hand object with +=
a += 3 # This is the same as a = a + 3

# Subtract an object from another; store in left-hand object with -=
a -= 3 # This is the same as a = a - 3
```

Numeric comparison operators

```
# Test for equality with ==
3 == 3 # This returns true

# Test for not-equality with ≠
3 ≠ 3 # This returns false

# Test greater than with >
3 > 1

# Test greater or equal than with ≥
3 ≥ 3

# Test less than with <
3 < 4

# Test less or equal than with ≤
3 ≤ 4
```

Other operators

```
# Determine if a value is in an array with x in arr
x = [11, 13, 19]
13 in x # This returns true

# Pipe values to a function with value ▷ fn
x ▷ (y → length(y) + sum(y)) # This returns 43
```

Logical operators

```
# Logical not with ~
~(2 == 2) # Returns false

# Elementwise and with &
(1 ≠ 1) & (1 < 1) # Returns false

# Elementwise or with |
(1 ≥ 1) | (1 < 1) # Returns true

# Elementwise xor (exclusive or) with ⊻
(1 ≠ 1)⊻(1 < 1) # Returns false
```

> Vectors

Vectors are one-dimensional arrays in Julia. They allow a collection of items such as floats, integers, strings or a mix that allows duplicate values.

Creating vectors

```
# Create vectors with square brackets, [x1, x2, x3]
x = [1, 2, 3]

# Create vectors, specifying element types using Vector{type}()
Vector{Float64}([1, 2, 3])

# Create sequence of numbers from a to b with a:b
37:100

# Create sequence of numbers from a to b in steps with a:step:b
1:2:101

# Create vector that repeats m times and each element repeats n times
repeat(vector, inner=n, outer=m)
```

Vector functions

```
# Sorting vectors with sort(x)
x = [9, 1, 4]
sort(x)

# Reversing vectors with reverse(x)
reverse(x)

# Reversing in-place with reverse!(x)
reverse!(x)

# Get vector's unique elements with unique()
unique(x)
```

Selecting vector elements

```
# Selecting the 6th element of a vector with x[6]
x = [9, 1, 4, 6, 7, 11, 5]
x[6]

# Selecting the first element of a vector with x[begin]
x[begin] # This is the same as x[1]

# Selecting the last element of a vector with x[end]
x[end] # This is the same as x[7]

# Slicing elements two to six from a vector with x[2:6]
x[2:6]

# Selecting the 2nd and 6th element of a vector with x[[2, 6]]
x[[2,6]]

# Selecting elements equal to 5 with x[x .== 5]
x[x .== 5]

# Selecting elements less than 5 with x[x .< 5]
x[x .< 5]

# Selecting elements in the vector 2, 5, 8 with x[in([2, 5, 8]).(x)]
x[in([2, 5, 8]).(x)]
```

datacamp

julia

Julia Basics Cheat Sheet

Learn Julia online at www.DataCamp.com

> Math functions

```
# Example vector
x = [9, 1, 4, 6, 7, 11, 5]

# Get the logarithm of a number with log()
log(2)

# Get the element-wise logarithm of a vector with log.()
log.(x)

# Get the exponential of a number with exp()
exp(2)

# Get the element-wise exponential of a vector with exp.()
exp.(x)

# Get the maximum of a vector with maximum()
maximum(x)

# Get the minimum of a vector with minimum()
minimum(x)

# Get the sum of a vector with sum()
sum(x)
```

The following code requires installing and loading the `Statistics` and `StatsBase` packages. This can be done with the command below

```
] # Enter package mode
add Statistics # Add the Statistics package
add StatsBase # Add the StatsBase package
using Statistics # Load the package with using
using StatsBase # Load the package with using

# Get the mean of a vector with mean()
mean(x)

# Get the median of a vector with median()
median(x)

# Get quantiles of a vector with quantile(x, p)
quantile(x, [0.25, 0.75])

# Round values of a vector with round.(x, digits = n)
round.(x, 2)

# Get the ranking of vector elements with StatsBase.ordinalrank()
ordinalrank(x)

# Get the variance of a vector with var()
var(x)

# Get the standard deviation of a vector with std()
std(x)

# Get the correlation between two vectors with cor(x, y)
y = [1, 4, 2, 10, 23, 16, 5]
cor(x, y)
```

> Getting started with characters and strings

Characters and strings are text data types in Julia. Characters refer to text data with exactly one character, and are created with single quotes, `''`. Strings are sequences of characters, and are created with double or triple-double quotes, `" "` or `""" """`.

```
# Create a character variable with single quotes
char = 'a'

# Create a string variable with double quotes
string = "Hello World!"

# Create a string variable with triple double quotes
string = """Hello
            World!"""

# Extract a single character from a string
string = "Hello World!"

string[1] # This extracts the first character
string[begin] # This extracts the first character
string[end] # This extracts the last character

# Extract a string from a string
string[1:3] # Extract first three characters as a string
string[begin:4] # Extract first four characters as a string
string[end-2: end] # Extract last three characters as a string
```

Combining and splitting strings

```
# Combine strings with *
"Listen" * " to " * "DataFramed!" # This returns "Listen to DataFramed!"

# Repeat strings with ^
"Echo! " ^ 3 # Returns "Echo! Echo! Echo! "

# Interpolate strings with "$value"
language = "Julia"
"I'm learning $language" # Returns "I'm learning Julia"

# Split strings on a delimiter with split()
split("lions and tigers and bears", " and ") # Returns 3-element vector
```

Finding and mutating strings

```
# Detect the presence of a pattern in a string with occursin()
occursin("Julia", "Julia for data science is cool") # This returns true

# Find the position of the first match in a string with findfirst()
findfirst("Julia", "Julia for data science is cool") # This returns 1:5

# Convert a string to upper case with uppercase()
uppercase("Julia") # Returns "JULIA"

# Convert a string to lower case with lowercase()
lowercase("Julia") # Returns "julia"

# Convert a string to title case case with titlecase()
titlecase("Julia programming") # Returns "Julia Programming"

# Replace matches of a pattern with a new string with replace()
replace("Learn Python on DataCamp.", "Python" => "Julia")
```

> Getting started with DataFrames

```
# Install the DataFrames and CSV packages
]
add DataFrames
add CSV
using DataFrames
using CSV

# Create a DataFrame with DataFrame()
df = DataFrame(
    numeric_column = 1:4, # Vector of integers
    string_column= ['M', 'F', 'F', 'M'], # Vector of characters
    a_number = 0, # Fill whole column with one integer
    a_string = "data frames" # Fill whole column with one string
)

# Select a row from a data frame with [ and column number
df[3, :] # Return the third row and all columns

# Select a column from a DataFrame using . and column name
df.string_column

# Select a column from a DataFrame using [ and column number
df[:, 2] # Return the second column and all rows

# Select an element from a DataFrame using [ and row and column numbers
df[1, 2] # Return the first row of the second column
```

> Manipulating data frames

```
# Concatenate two data frames horizontally with hcat()
df1 = DataFrame(column_A = 1:3, column_B = 1:3)
df2 = DataFrame(column_C = 4:6, column_D = 4:6)

df3 = hcat(df1, df2) # Returns 4-column DataFrame with columns A, B, C, D

# Filter for rows of a df3 with filter() where column_A > 2
df_filter = filter(row => row.column_A > 2, df3)

# Select columns of a data frame with select()
select(df3, 2) # Return the second column

# Drop columns of a data frame with select(Not())
select(df3, Not(2)) # Return everything except second column

# Rename columns of a data frame with rename(old => new)
rename(df3, ["column_A" => "first_column"])

# Get rows of a df3 with distinct values in column_A with unique(df, :col)
unique(df3, :column_A)

# Order the rows of a data frame with sort()
sort(df3, :numeric_column)

# Get data frame summary statistics with describe()
describe(df3)
```

Basics

1 Data are supplied to the `plot` function as arguments (`x`, or `x,y`, or `x,y,z`). Keyword arguments specify attributes.

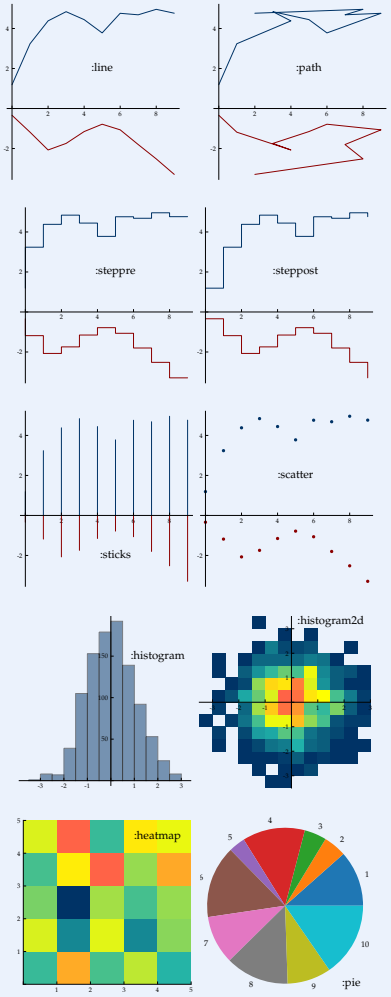
2 Arguments are interpreted flexibly: `x` and `y` can be vectors, or `x` can be a vector and `y` a function to be applied to x, or `x` can be omitted and inferred as `eachindex(y)`.

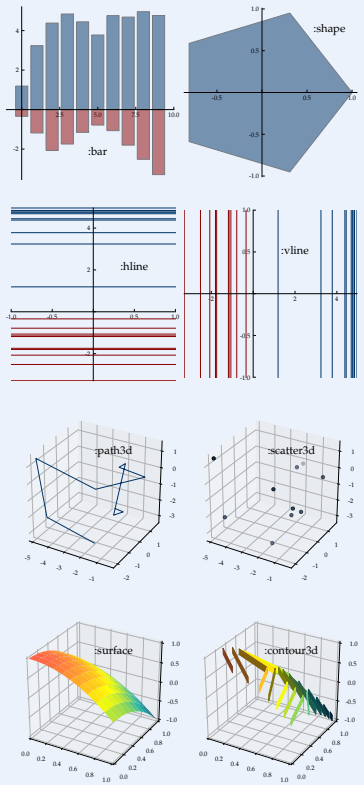
3 `plot(args...;kwargs...)` creates a new plot object, and `plot!(p,args...;kwargs...)` modifies the plot `p`. If omitted, `p` defaults to the plot `current()`.

4 A **series** is a set of data to be plotted together. The possible `seriestype`s are

```
[:line, :path, :steppre, :steppost, :sticks,
:scatter, :heatmap, :hexbin, :barbins, :barhist,
:histogram, :scatterbins, :scatterhist, :stepbins,
:stephist, :bins2d, :histogram2d, :histogram3d,
:density, :bar, :hline, :vline, :contour, :pie,
:shape, :image, :path3d, :scatter3d, :surface,
:wireframe, :contour3d, :volume]
```

The `seriestype` is specified as a keyword argument with key `seriestype` or `st`.





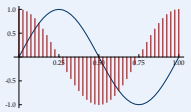
5 Most series types have function aliases, like `scatter(x,y)` for `plot(x,y,seriestype=:scatter)` and same for `scatter!`. Use the aliases for series docstrings (`?scatter`).

6 If a data argument or attribute is a 2D array, its **columns** are interpreted as separate series.

Combining plots

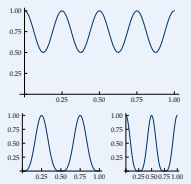
1 Series may be combined on the same axes using `plot!`.

```
x = 0:0.025:1
plot(x, x->sin(2π*x))
plot!(x, x->cos(2π*x),
    seriestype = :sticks)
```



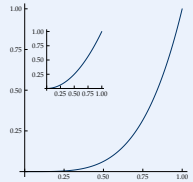
2 Series may be combined on separate axes using `@layout`.

```
l = @layout [a{0.6h};
             b{0.6w} c]
f(x) = sin(2π*x)^4 +
             cos(2π*x)^4
p1 = plot(x,f)
p2 = plot(x,x->sin(2π*x)^4)
p3 = plot(x,x->cos(2π*x)^4)
plot(p1, p2, p3, layout=l)
```



3 Inset plots: supply (parent plot index, bounding box) to `inset`. `bbox` arguments are `x, y, width, height`, each as a proportion of the corresponding parent plot dimension. Also, specify the `subplot` index for the new plot.

```
BB = bbox(0.15,0.15,
          0.35,0.35)
plot(x, x->x^4)
plot!(x, x->x^2,
      inset = (1, BB),
      subplot = 2)
```



Plot styling

1 **Plot attributes** (Default values followed by other possible values are shown in parentheses.)

(i) **Plots**

- `background_color/bg` (`RGB(1,1,1)`, `:Firebrick`).
- `size` (`(600, 400)`, `(300, 300)`)
- `dpi` (`100`, `50`, `200`)
- `fontfamily` (`sans-serif`, `serif`)

(ii) **Subplots**

- `title` (`nothing`, `"My favorite plot"`)
- `legend/leg` (`:none`, `:best`, `:right`, `:left`, `:top`, `:bottom`, `:inside`, `:legend`, `:topright`, `:topleft`, `:bottomleft`, `:bottomright`)
- `framestyle/frame` (`:box`, `:semi`, `:axes`, `:origin`, `:zerolines`, `:grid`, `:none`)
- `aspect_ratio/ratio` (`:none`, `:equal`, `2.0`)
- `camera/cam` (`(30,30)`, `(45,45)`)
- `color_palette/palette` (`:auto`, `[:blue,:red,:green]`)

(iii) **Axes**

- `grid` (`true/false`)
- `gridlinewidth` (`0.5`, `0.25`, `1.0`)
- `gridstyle` (`:solid`, `:auto`, `:dash`, `:dot`)
- `link` (`:none`, `:x`, `:y`, `:both`, `:all`)
- `xlims`, `ylims`, `zlims`, (`:auto`, `(-10,5)`)
- `xticks`, `yticks`, `zticks` (`:auto`, `-4:2:4`)
- `xscale`, `yscale`, `yscale` (`:none`, `:ln`, `:log2`, `:log10`)
- `xguide/xlabel`, `yguide/ylabel` (`nothing`, `"time (s)"`)

2 **Series attributes**

(i) **Points**

- `markercolor/mc` (`:auto`, `:blue`, `RGB(0.2,0.4,0.2)`)
- `markeralpha/ma` (`1.0`, `0.5`, `0.2`)
- `markersize/ms` (`4`, `2`, `8`)
- `markershape/shape` (`:none`, `:auto`, `:circle`, `:rect`, `:star5`, `:diamond`, `:hexagon`, `:cross`, `:xcross`, `:utriangle`, `:dtriangle`, `:rtriangle`, `:ltriangle`, `:pentagon`, `:heptagon`, `:octagon`, `:star4`, `:star6`, `:star7`, `:star8`, `:vline`, `:hline`, `:+`, `:x`)
- `markerstrokecolor/msc` (`:auto`, `:blue`, `RGB(0,0,0)`)
- `markerstrokealpha/msa` (`1.0`, `0.5`, `0.2`)
- `markerstrokewidth/msw` (`0.5`, `1`)

(ii) **Lines**

- `linecolor/lc` (`:auto`, `:blue`, `RGB(0.2,0.4,0.2)`)
- `linealpha/la` (`1.0`, `0.5`, `0.2`)
- `linestyle/ls` (`:solid`, `:auto`, `:dash`, `:dot`, `:dashdot`, `:dashdotdot`)
- `linewidth/lw`

(iii) **Surfaces**

- `fillrange` (`nothing`, `0`, `sin.(x)`)
- `fillcolor/fc` (`:auto`, `:blue`, `RGB(0.2,0.4,0.2)`)
- `fillalpha/fa` (`1.0`, `0.5`, `0.2`)

Annotations and images

1 Add text with the `annotations/ann` attribute. Value should be a vector of tuples of the form `(x,y,txt)`, where `txt` is either a string or an object created with `text`.

```
ann = [(-π/2,-0.85,"min."),
       (-0.25,0.25,
       text("inflection point",
            pointsize=12, halign=:right,
            valign=:center, rotation=45))]
plot(sin, ann=ann)
# add arrowhead to line plot:
plot!([(-0.5,0.2),(-0.02,0.02)],arrow=1.0)
```

2 Add an image to a plot:

```
using Images
img = load("example.png")
x = range(-2, 2, length=size(img,1))
y = range(0, 1, length=size(img,2))
plot(x,y,img) # plots the image in [-2,2] × [0,1]
plot!(sin) # draw curve over image
```

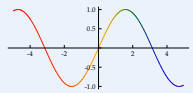
Color gradients

1 There are five collections of color gradients. `:Plots`, `:cmocean`, `:misc`, `:colorcet`, `:colorbrewer`. Choose one with `clibrary`.

2 Select your color gradient with `markercolor/linecolor/fillcolor`

3 Supply `z`-values for coloring with `marker_z/line_z/fill_z`

```
clibrary(:misc)
x = 0:0.01:1
plot(x, sin.(x),
     linecolor = :rainbow,
     line_z = cos.(x))
```



Miscellaneous

1 Data points can be grouped into separate series using the `group` attribute.

```
x,y = randn(100), randn(100)
class = rand(1:3, 100)
plot(x,y, group = class,
          color = [:blue :green :red])
```

2 `DataFrame` support:

```
using StatsPlots, DataFrames
D = DataFrame(a = randn(10),
              b = randn(10),
              c = rand(10))
@df D scatter(:a, :b, marker_z = :c)
```

3 **Recipes** provide support for custom types throughout Plots.

```
@recipe function f(A::Array{<:Complex})
    xguide := "Re(x)" # set attribute
    yguide --> "Im(x)" # set tentatively
    real.(A), imag.(A) # transformed data
end
```

4 `plotattr` provides information about plot attributes.

```
plotattr() # get help with plotattr
plotattr(:Series) # list Series attributes
plotattr("fill_z") # documentation for fill_z
```

5 Write figures to disk:

```
p = plot(x -> sin(x))
savefig(p, "myfig.pdf")
savefig("myfig.pdf") # uses p = current()
```

Formats for PyPlot backend are `eps`, `ps`, `pdf`, `png`, `svg`.

Plots.jl cheatsheet · Samuel S. Watson

Basics

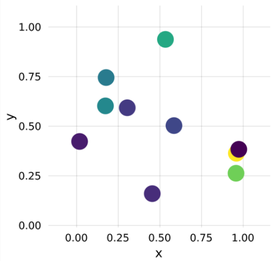
Set the Scene

The Scene object holds everything in a plot
Initializing: scene = Scene()

Basic plotting

You can put your mouse in the plot window and scroll to zoom. Right click and drag lets you pan around the scene, and left click and drag lets you do selection zoom (in 2D plots), or orbit around the scene (in 3D plots).

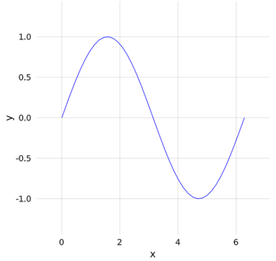
It is worth noting initally that if you run a Makie.jl example and nothing shows up, you likely need to do display(scene) to render the example on screen.



Scatter plot

```
using Makie

x = rand(10)
y = rand(10)
colors = rand(10)
scene = scatter
(x, y, color = colors)
```



Line plot

```
using Makie

x = range(0, stop = 2pi,
length = 40)
f(x) = sin.(x)
y = f(x)
scene = lines
(x, y, color = :blue)
```

Animation

Makie.jl saves to: .mkv, .mp4, .webm, .gif

All you need to do is wrap your changes in the record function.

```
using Makie
scene = lines(rand(10); linewidth=10)
record(scene, "line_changing_colour.mp4", 1:255;
framerate = 60) do i
  scene.plots[2][:color] = RGBf0(i/255, (255 - i)/255, 0)
  # animate scene
end
```

Using time

time = Node(0.0)
lift is using to set up a pipeline to access its value.
Whenever the Node time is updated (e.g. when you push! to it), the plot will also be updated.

```
push!(time, Base.time())
```

Appending data to a plot

If you're planning to append to a plot, like a lines or scatter plot, you will want to pass an Observable Array of Points to the plotting function, instead of passing x, y as separate Arrays. This will mean that you won't run into dimension mismatch issues.

Animating in a loop

for loop:

```
for i = 1:length(r)
   s[:markersize] = r[i]
   sleep(1/24)
end
```

You don't need to use AbstractPlotting.force_update!() in a loop

If you want to animate a plot while interacting, use async_latest

Interaction

Node interaction pipeline

A Node is a Julia structure that allows its value to be updated interactively.

```
x = Node(0.0)
```

The value of the x can be changed simply using push!

to_value to get the value of a Node

Nodes depending on other Nodes

You can create a node depending on another node using lift:

```
f(a) = a^2
y = lift(a -> f(a), x)
```

Updating the value of x will also update the value of y!

Event triggering

Often it is the case that you want an event to be triggered each time a Node has its value updated. This is done using the on-do block from Observables.

```
on(x) do val
   println("x just got the value $val")
end
push!(x, 5.0);
```

Functions

text

text(string)

Plots a text.

meshscatter

meshscatter(positions)
meshscatter(x, y)
meshscatter(x, y, z)

Plots a mesh for each element (x, y, z), (x, y), or positions. markersize is a scaling applied to the primitive passed as marker.

scatter

scatter(positions)
scatter(x, y)
scatter(x, y, z)

Plots a marker for each element in (x, y, z), (x, y), or positions.

mesh

mesh(x, y, z)
mesh(mesh_object)
mesh(x, y, z, faces)
mesh(xyz, faces)

Plots a 3D mesh.

lines

mesh(x, y, z)
mesh(mesh_object)
mesh(x, y, z, faces)
mesh(xyz, faces)

Creates a connected line plot for each element in (x, y, z), (x, y) or positions.

volume

volume(volume_data)

Plots a volume.

image

image(x, y, image)
image(image)

Plots an image on range x, y

contour

contour(x, y, z)

Creates a contour plot of the plane spanning x::Vector, y::Vector, z::Matrix

surface

surface(x, y, z)

Plots a surface, where (x, y) define a grid whose heights are the entries in z.

Check Makie.jl docs for more informations

Paragraphs and Line Breaks

- Para = one or more consecutive lines
- Empty lines = end of paragraph
- > 2 spaces at EoL = HTML line break

Headers

- # This is an H1
- ## This is an H2
- ###### This is an H6

Alternative: ==s for H1, --s for H2

Emphasis

- *This is an *
  and _so is this_
- **This is a **
  and __so is this__
- Use same closing marker as opening
- Emphasis possible in middle of word
- * or _ for a literal * or _

Blockquotes

- Prefix > for each blockquote line...
  ... or even hard-wrapped paragraph
- Additional > for nested blockquotes
- Blockquotes can contain Markdown

Images

- ![Alt text](URL "Title")
- ![Alt text][id]
  [id]: URL "Title"
- See syntax for Links

Links

Inline	★ This is [an example](http://example.com/) link ★ This is [example](http://example.com/ "Title") with a title
Reference	★ This is a [reference link][id] with an id [id]: http://example.com/ "Title" ★ This is a [reference link][] without an id [reference link]: http://example.com/ (Title)

- Link id can be letters, numbers, spaces, and punctuation
- Link id is case-insensitive
- In ref links, link definition can be indented up to 3 spaces
- In ref links, use parentheses, single or double quotes for title
- In ref links, link URL can be surrounded by < >
- In ref links, title can be on separate line with indentation
- In ref links, link definitions can appear anywhere in document

Lists

Unordered	Asterisks, pluses & hyphens interchangeably	+ Red * Green - Blue
Ordered	Numbers, not necessarily sequential. Always start with 1.	1. Red 1. Green 1. Blue

- List marker must be followed by 1/more spaces or a tab
- List marker can be indented upto 3 spaces
- Hanging indent supported but not required
- Blank lines between list items implies paragraph
- Paragraph start must be indented by 4 spaces or a tab
- To avoid unintended list numbering (e.g 1986. What a great season) backslash the period (1986\. What a great season).

Markdown

➔ http://daringfireball.net/projects/markdown/
Cheat sheet from: **Ahren Code** ➔ http://ahren.org/code/

Code Block

- Indent code by 4 spaces or 1 tab
- One level indentation will be removed
- & and <, > are automatically HTML'ised
- Markdown not usable in code blocks

Code Spans

- Use backticks around code: `` `printf()` ``
- Use multiple backticks for literal backtick: ``` ``ls `date+%h` | wc -l`` ```
  - Use space if literal backtick is at start of code span: ``` `` ` `` ```

Miscellaneous

Horizontal Rule / Line

3 or more asterisks, hyphens or underscores

List with embedded Blockquote

Indent > delimiters:

```
* A list item with a blockquote:
  > This is a blockquote
```

List with embedded Code block

Indent code block 8 spaces or two tabs:

```
* A list item with a code block:
      ... some code ....
```

Automatic Links

- Use <, > for auto-linking URLs
- Use <, > for auto-linking email addrs
- Email text is auto obscured (somewhat)

For all else use HTML markup

- blank lines around block level elements
- no indentation for tags
- no Markdown inside HTML blocks