



Python For Data Science

python™ Basics Cheat Sheet

Learn Python Basics online at www.DataCamp.com

> Variables and Data Types

Variable Assignment

```
>>> x=5
>>> x
5
```

Calculations With Variables

```
>>> x+2 #Sum of two variables
7
>>> x-2 #Subtraction of two variables
3
>>> x*2 #Multiplication of two variables
10
>>> x**2 #Exponentiation of a variable
25
>>> x%2 #Remainder of a variable
1
>>> x/float(2) #Division of a variable
2.5
```

Types and Type Conversion

```
str()
'5', '3.45', 'True' #Variables to strings

int()
5, 3, 1 #Variables to integers

float()
5.0, 1.0 #Variables to floats

bool()
True, True, True #Variables to booleans
```

> Libraries

pandas	NumPy	matplotlib	learn
Data analysis	Scientific computing	2D plotting	Machine learning

Import Libraries

```
>>> import numpy
>>> import numpy as np
```

Selective import

```
>>> from math import pi
```

> Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2
'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
'thisStringIsAwesomeInnit'
>>> 'm' in my_string
True
```

String Indexing

Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```

String Methods

```
>>> my_string.upper() #String to uppercase
>>> my_string.lower() #String to lowercase
>>> my_string.count('w') #Count String elements
>>> my_string.replace('e', 'i') #Replace String elements
>>> my_string.strip() #Strip whitespaces
```

> NumPy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3],[4,5,6]])
```

Selecting Numpy Array Elements

Index starts at 0

```
Subset
>>> my_array[1] #Select item at index 1
2
```

```
Slice
>>> my_array[0:2] #Select items at index 0 and 1
array([1, 2])
```

```
Subset 2D Numpy arrays
>>> my_2darray[:,0] #my_2darray[rows, columns]
array([1, 4])
```

Numpy Array Operations

```
>>> my_array > 3
array([False, False, False, True], dtype=bool)
>>> my_array * 2
array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
array([6, 8, 10, 12])
```

Numpy Array Functions

```
>>> my_array.shape #Get the dimensions of the array
>>> np.append(other_array) #Append items to an array
>>> np.insert(my_array, 1, 5) #Insert items in an array
>>> np.delete(my_array,[1]) #Delete items in an array
>>> np.mean(my_array) #Mean of the array
>>> np.median(my_array) #Median of the array
>>> my_array.corrcoef() #Correlation coefficient
>>> np.std(my_array) #Standard deviation
```

> Lists

Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements

Index starts at 0

Subset

```
>>> my_list[1] #Select item at index 1
>>> my_list[-3] #Select 3rd last item
```

Slice

```
>>> my_list[1:3] #Select items at index 1 and 2
>>> my_list[1:] #Select items after index 0
>>> my_list[:3] #Select items before index 3
>>> my_list[:] #Copy my_list
```

Subset Lists of Lists

```
>>> my_list2[1][0] #my_list[list][itemOfList]
>>> my_list2[1][:2]
```

List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

List Methods

```
>>> my_list.index(a) #Get the index of an item
>>> my_list.count(a) #Count an item
>>> my_list.append('!)') #Append an item at a time
>>> my_list.remove('!)') #Remove an item
>>> del(my_list[0:1]) #Remove an item
>>> my_list.reverse() #Reverse the list
>>> my_list.extend('!)') #Append an item
>>> my_list.pop(-1) #Remove an item
>>> my_list.insert(0,'!') #Insert an item
>>> my_list.sort() #Sort the list
```

> Python IDEs (Integrated Development Environment)



DataCamp Workspace
is an in-browser Jupyter IDE



SPYDER
Free IDE that is included
with Anaconda



jupyter
Create and share
documents with live code

> Asking For Help

```
>>> help(str)
```

Learn Data Skills Online at
www.DataCamp.com



Working with Dates and Times in Python

Learn Python Basics online at www.DataCamp.com

> Key definitions

When working with dates and times, you will encounter technical terms and jargon such as the following:

- **Date:** Handles dates without time.
- **POSIXct:** Handles date & time in calendar time.
- **POSIXlt:** Handles date & time in local time.
- **Hms:** Parses periods with hour, minute, and second
- **Timestamp:** Represents a single pandas date & time
- **Interval:** Defines an open or closed range between dates and times
- **Time delta:** Computes time difference between different datetimes

> The ISO8601 datetime format

The **ISO 8601 datetime format** specifies datetimes from the largest to the smallest unit of time (**YYYY-MM-DD HH:MM:SS TZ**). Some of the advantages of ISO 8601 are:

- It avoids ambiguities between MM/DD/YYYY and DD/MM/YYYY formats.
- The 4-digit year representation mitigates overflow problems after the year 2099.
- Using numeric month values (08 not AUG) makes it language independent, so dates make sense throughout the world.
- Python is optimized for this format since it makes comparison and sorting easier.

> Packages used in this cheat sheet

Load the packages and dataset used in this cheatsheet.

```
import datetime as dt
import time as tm
import pytz
import pandas as pd
```

In this cheat sheet, we will be using 3 pandas series — `iso`, `us`, `non_us`, and 1 pandas DataFrame `parts`

iso
1969-07-20 20:17:40
1969-11-19 06:54:35
1971-02-05 09:18:11

us
07/20/1969 20:17:40
11/19/1969 06:54:35
02/05/1971 09:18:11

non_us
20/07/1969 20:17:40
19/11/1969 06:54:35
05/02/1971 09:18:11

parts		
year	month	day
1969	7	20
1969	11	19
1971	2	5

> Getting the current date and time

```
# Get the current date
dt.date.today()
```

```
# Get the current date and time
dt.datetime.now()
```

> Reading date, datetime, and time columns in a CSV file

```
# Specify datetime column
pd.read_csv("filename.csv", parse_dates = ["col1", "col2"])

# Specify datetime column
pd.read_csv("filename.csv", parse_dates = {"col1": ["year", "month", "day"]})
```

> Parsing dates, datetimes, and times

```
# Parse dates in ISO format
pd.to_datetime(iso)

# Parse dates in US format
pd.to_datetime(us, dayfirst=False)

# Parse dates in NON US format
pd.to_datetime(non_us, dayfirst=True)

# Parse dates, guessing a single format
pd.to_datetime(iso, infer_datetime_format=True)

# Parse dates in single, specified format
pd.to_datetime(iso, format="%Y-%m-%d %H:%M:%S")

# Parse dates in single, specified format
pd.to_datetime(us, format="%m/%d/%Y %H:%M:%S")

# Make dates from components
pd.to_datetime(parts)
```

> Extracting components

```
# Parse strings to datetimes
dttm = pd.to_datetime(iso)

# Get year from datetime pandas series
dttm.dt.year

# Get day of the year from datetime pandas series
dttm.dt.day_of_year

# Get month name from datetime pandas series
dttm.dt.month_name()

# Get day name from datetime pandas series
dttm.dt.day_name()

# Get datetime.datetime format from datetime pandas series
dttm.dt.to_pydatetime()
```

> Rounding dates

```
# Rounding dates to nearest time unit
dttm.dt.round('1min')

# Flooring dates to nearest time unit
dttm.dt.floor('1min')

# Ceiling dates to nearest time unit
dttm.dt.ceil('1min')
```

> Arithmetic

```
# Create two datetimes
now = dt.datetime.now()
then = pd.Timestamp('2021-09-15 10:03:30')
```

```
# Get time elapsed as timedelta object
now - then
```

```
# Get time elapsed in seconds
(now - then).total_seconds()
```

```
# Adding a day to a datetime
dt.datetime(2022,8,5,11,13,50) + dt.timedelta(days=1)
```

> Time Zones

```
# Get current time zone
tm.localtime().tm_zone
```

```
# Get a list of all time zones
pytz.all_timezones
```

```
# Parse strings to datetimes
dttm = pd.to_datetime(iso)
```

```
# Get datetime with timezone using location
dttm.dt.tz_localize('CET', ambiguous='infer')
```

```
# Get datetime with timezone using UTC offset
dttm.dt.tz_localize('+0100')
```

```
# Convert datetime from one timezone to another
dttm.dt.tz_localize('+0100').tz_convert('US/Central')
```

> Time Intervals

```
# Create interval datetimes
start_1 = pd.Timestamp('2021-10-21 03:02:10')
finish_1 = pd.Timestamp('2022-09-15 10:03:30')
start_2 = pd.Timestamp('2022-08-21 03:02:10')
finish_2 = pd.Timestamp('2022-12-15 10:03:30')
```

```
# Specify the interval between two datetimes
pd.Interval(start_1, finish_1, closed='right')
```

```
# Get the length of an interval
pd.Interval(start_1, finish_1, closed='right').length
```

```
# Determine if two intervals are intersecting
pd.Interval(start_1, finish_1, closed='right').overlaps(pd.Interval(start_2, finish_2, closed='right'))
```

> Time Deltas

```
# Define a timedelta in days
pd.Timedelta(7, "d")
```

```
# Convert timedelta to seconds
pd.Timedelta(7, "d").total_seconds()
```

Learn Data Skills Online at
www.DataCamp.com

Working with text data in Python

Learn Python online at www.DataCamp.com

> Example data used throughout this cheat sheet

Throughout this cheat sheet, we'll be using two pandas series named suits and rock_paper_scissors.

```
import pandas as pd

suits = pd.Series(["clubs", "Diamonds", "hearts", "Spades"])
rock_paper_scissors = pd.Series(["rock", "paper", "scissors"])
```

> String lengths and substrings

```
# Get the number of characters with .str.len()
suits.str.len() # Returns 5 8 6 6

# Get substrings by position with .str[]
suits.str[2:5] # Returns "ubs" "amo" "art" "ade"

# Get substrings by negative position with .str[]
suits.str[:-3] # "cl" "Diamo" "hea" "Spa"

# Remove whitespace from the start/end with .str.strip()
rock_paper_scissors.str.strip() # "rock" "paper" "scissors"

# Pad strings to a given length with .str.pad()
suits.str.pad(8, fillchar="_") # "__clubs" "Diamonds" "__hearts" "__Spades"
```

> Changing case

```
# Convert to lowercase with .str.lower()
suits.str.lower() # "clubs" "diamonds" "hearts" "spades"

# Convert to uppercase with .str.upper()
suits.str.upper() # "CLUBS" "DIAMONDS" "HEARTS" "SPADES"

# Convert to title case with .str.title()
pd.Series("hello, world!").str.title() # "Hello, World!"

# Convert to sentence case with .str.capitalize()
pd.Series("hello, world!").str.capitalize() # "Hello, world!"
```

> Formatting settings

```
# Generate an example DataFrame named df
df = pd.DataFrame({"x": [0.123, 4.567, 8.901]})
#   x
# 0 0.123
# 1 4.567
# 2 8.901
```

```
# Visualize and format table output
df.style.format(precision = 1)
```

-	x
0	0.1
1	4.5
2	8.9

The output of style.format is an HTML table

> Splitting strings

```
# Split strings into list of characters with .str.split(pat="")
suits.str.split(pat="")
```

```
# [, "c" "l" "u" "b" "s", ]
# [, "D" "i" "a" "m" "o" "n" "d" "s", ]
# [, "h" "e" "a" "r" "t" "s", ]
# [, "S" "p" "a" "d" "e" "s", ]
```

```
# Split strings by a separator with .str.split()
suits.str.split(pat = "a")
```

```
# ["clubs"]
# ["Di", "monds"]
# ["he", "rts"]
# ["Sp", "des"]
```

```
# Split strings and return DataFrame with .str.split(expand=True)
suits.str.split(pat = "a", expand=True)
```

```
#      0      1
# 0 clubs  None
# 1 Di     monds
# 2 he     rts
# 3 Sp     des
```

> Joining or concatenating strings

```
# Combine two strings with +
suits + "5" # "clubs5" "Diamonds5" "hearts5" "spades5"
```

```
# Collapse character vector to string with .str.cat()
suits.str.cat(sep=", ") # "clubs, Diamonds, hearts, Spades"
```

```
# Duplicate and concatenate strings with *
suits * 2 # "clubsclubs" "DiamondsDiamonds" "heartshearts" "spadesSpades"
```

> Detecting Matches

```
# Detect if a regex pattern is present in strings with .str.contains()
suits.str.contains("[ae]") # False True True True
```

```
# Count the number of matches with .str.count()
suits.str.count("[ae]") # 0 1 2 2
```

```
# Locate the position of substrings with str.find()
suits.str.find("e") # -1 -1 1 4
```

> Extracting matches

```
# Extract matches from strings with str.findall()
suits.str.findall(".[ae]") # [] ["ia"] ["he"["pa", "de"]]
```

```
# Extract capture groups with .str.extractall()
suits.str.extractall("[ae](.)")
```

#	0 1
# 1 0	a m
# 2 0	e a
# 3 0	a d
# 1	e s

```
# Get subset of strings that match with x[x.str.contains()]
suits[suits.str.contains("d")] # "Diamonds" "Spades"
```

> Replacing matches

```
# Replace a regex match with another string with .str.replace()
suits.str.replace("a", "4") # "clubs" "Di4monds" "he4rts" "Sp4des"
```

```
# Remove a suffix with .str.removesuffix()
suits.str.removesuffix # "club" "Diamond" "heart" "Spade"
```

```
# Replace a substring with .str.slice_replace()
rhymes = pd.Series(["vein", "gain", "deign"])
rhymes.str.slice_replace(0, 1, "r") # "rein" "rain" "reign"
```

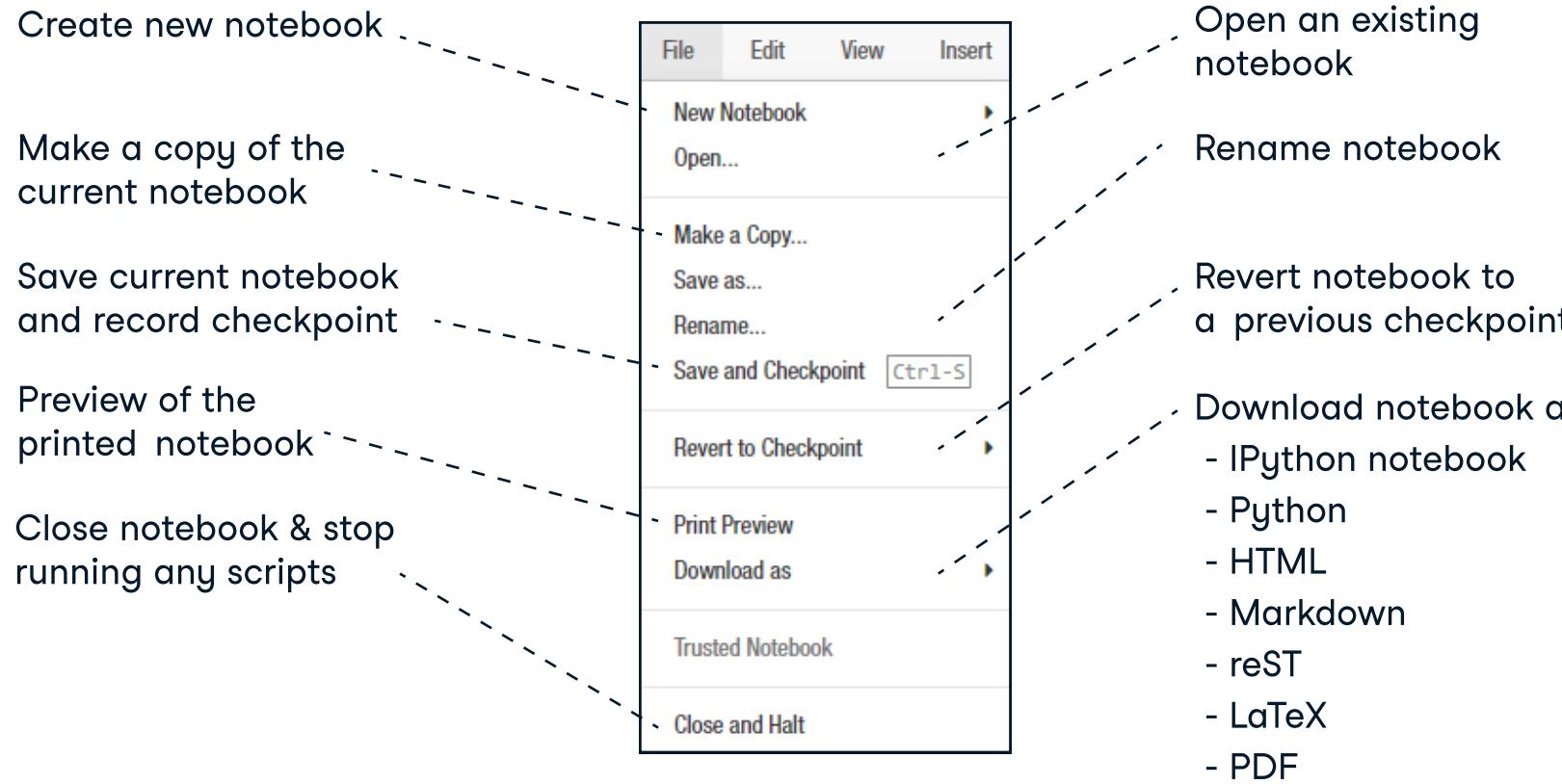
Learn Python Online at
www.DataCamp.com

Python For Data Science

Jupyter Cheat Sheet

Learn Jupyter online at www.DataCamp.com

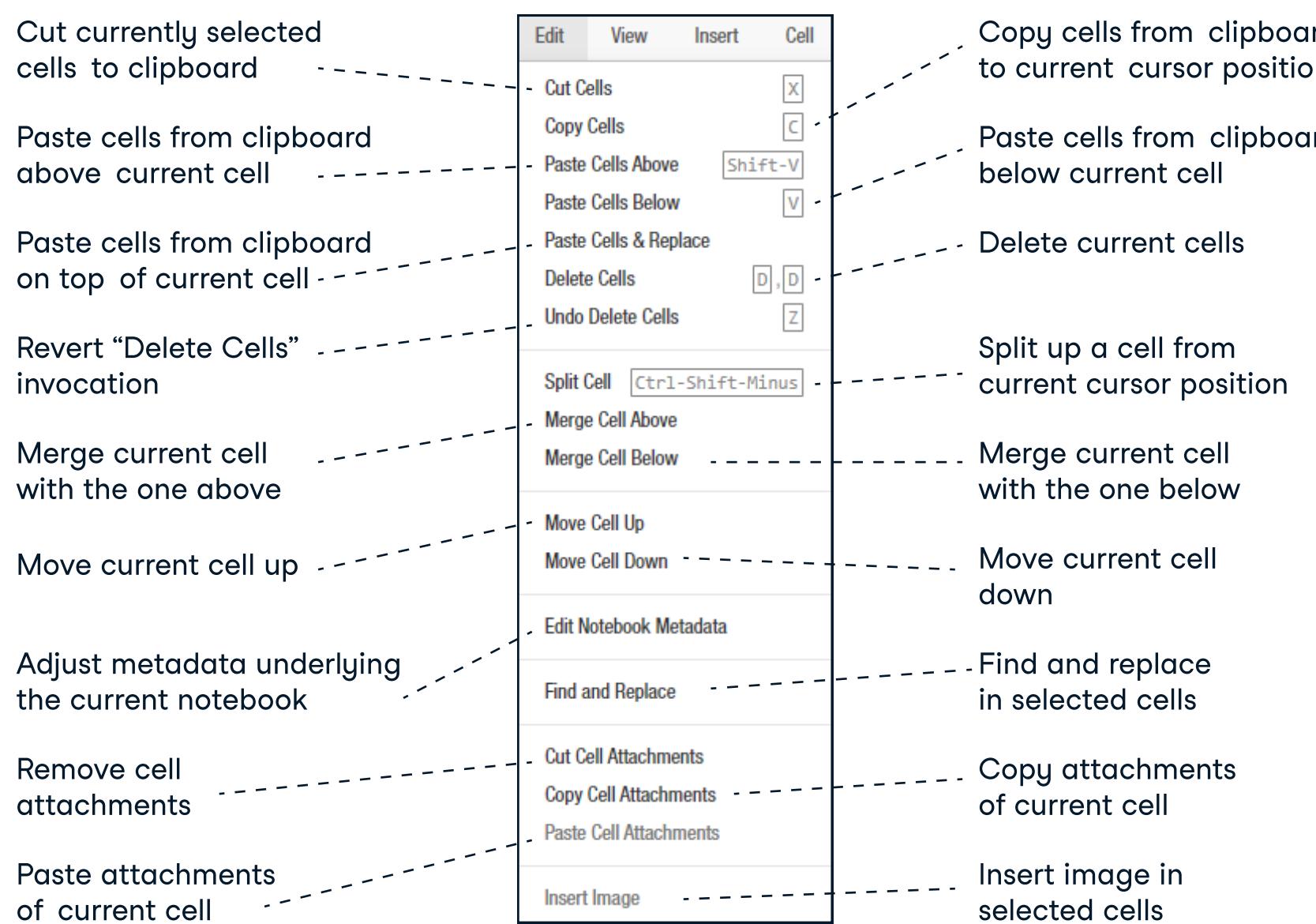
> Saving/Loading Notebooks



> Writing Code And Text

Code and text are encapsulated by 3 basic cell types: markdown cells, code cells, and raw NBConvert cells

Edit Cells



Insert Cells



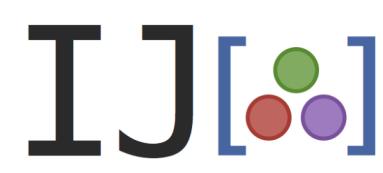
> Working with Different Programming Languages

Kernels provide computation and communication with front-end interfaces like the notebooks.
There are three main kernels:

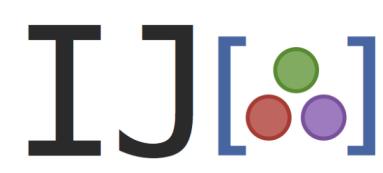
IP[y]:



IPython

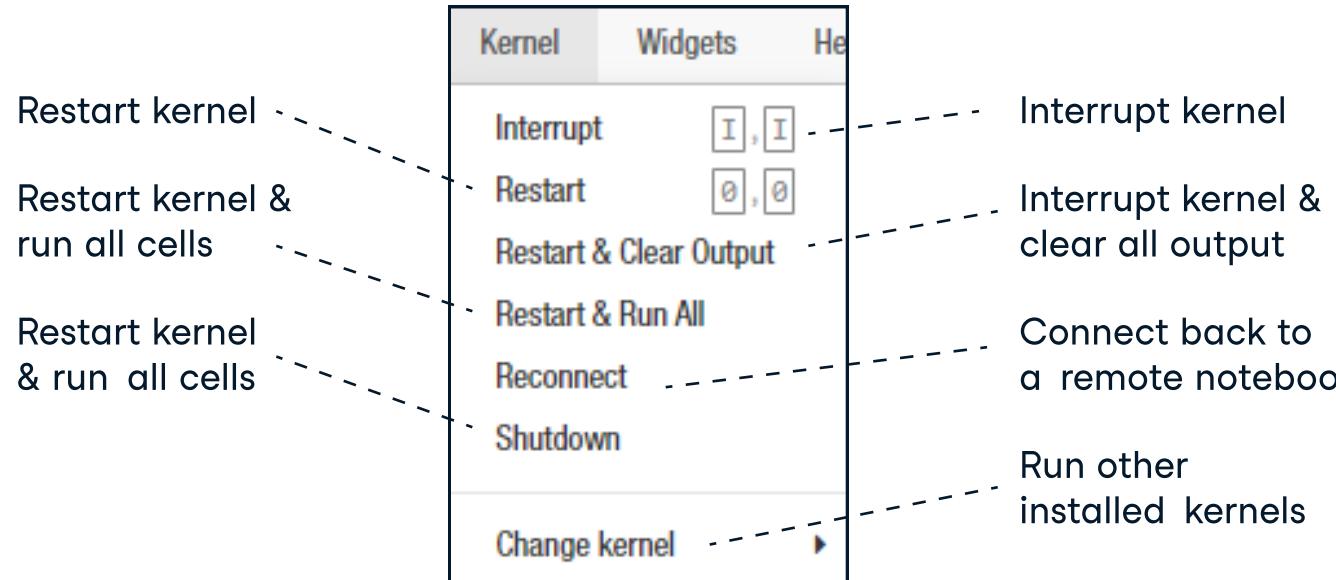


IRkernel

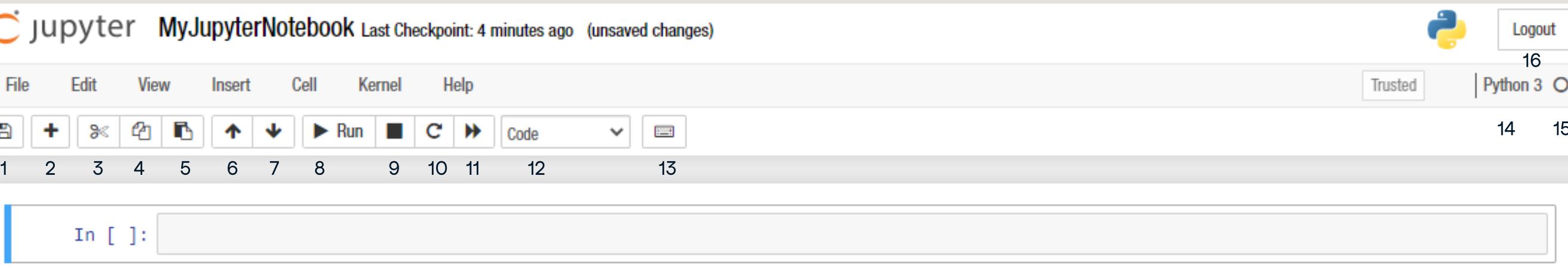


IJulia

Installing Jupyter Notebook will automatically install the IPython kernel.



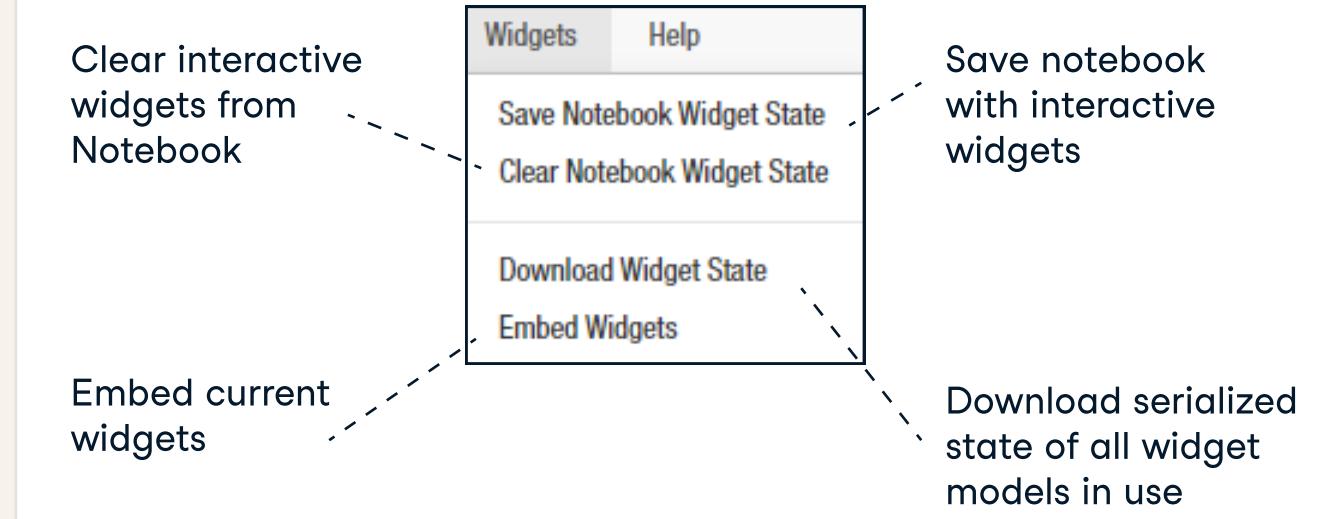
Command Mode:



> Widgets

Notebook widgets provide the ability to visualize and control changes in your data, often as a control like a slider, textbox, etc.

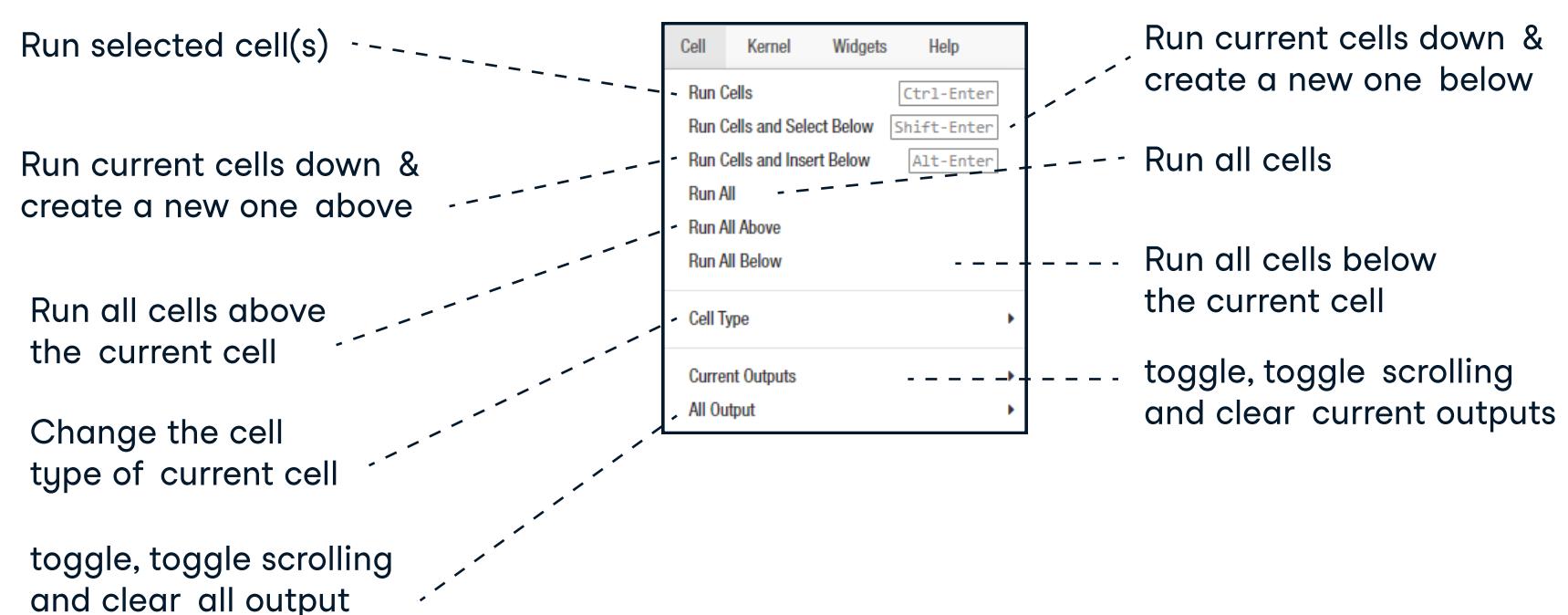
You can use them to build interactive GUIs for your notebooks or to synchronize stateful and stateless information between Python and JavaScript.



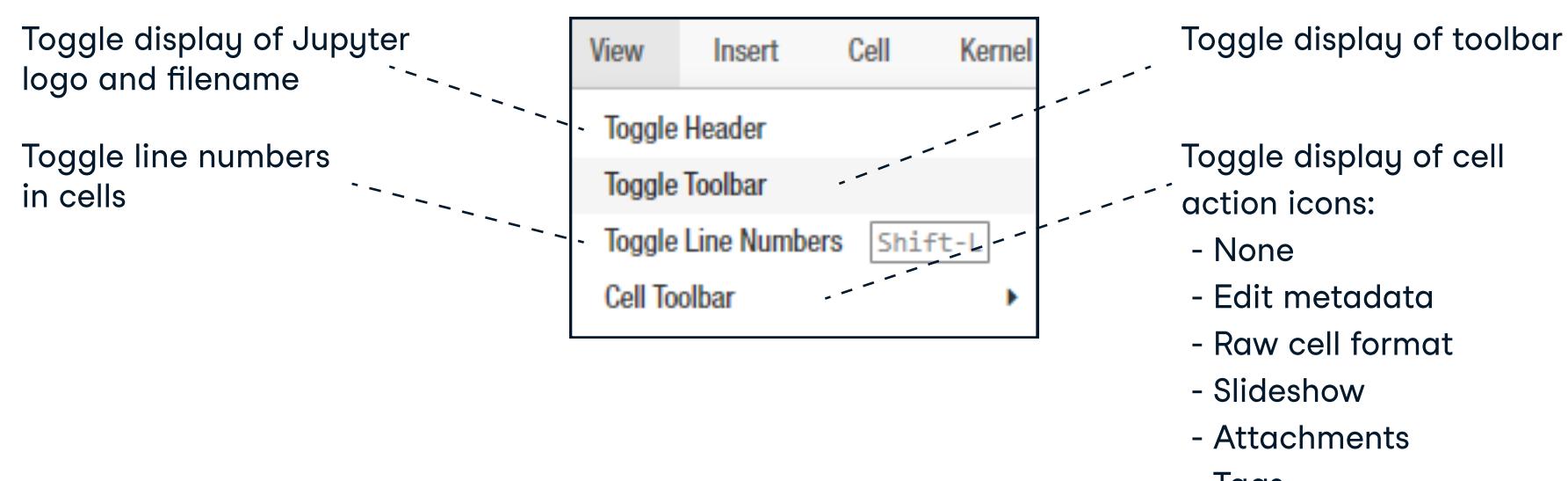
Edit Mode:

In []: |

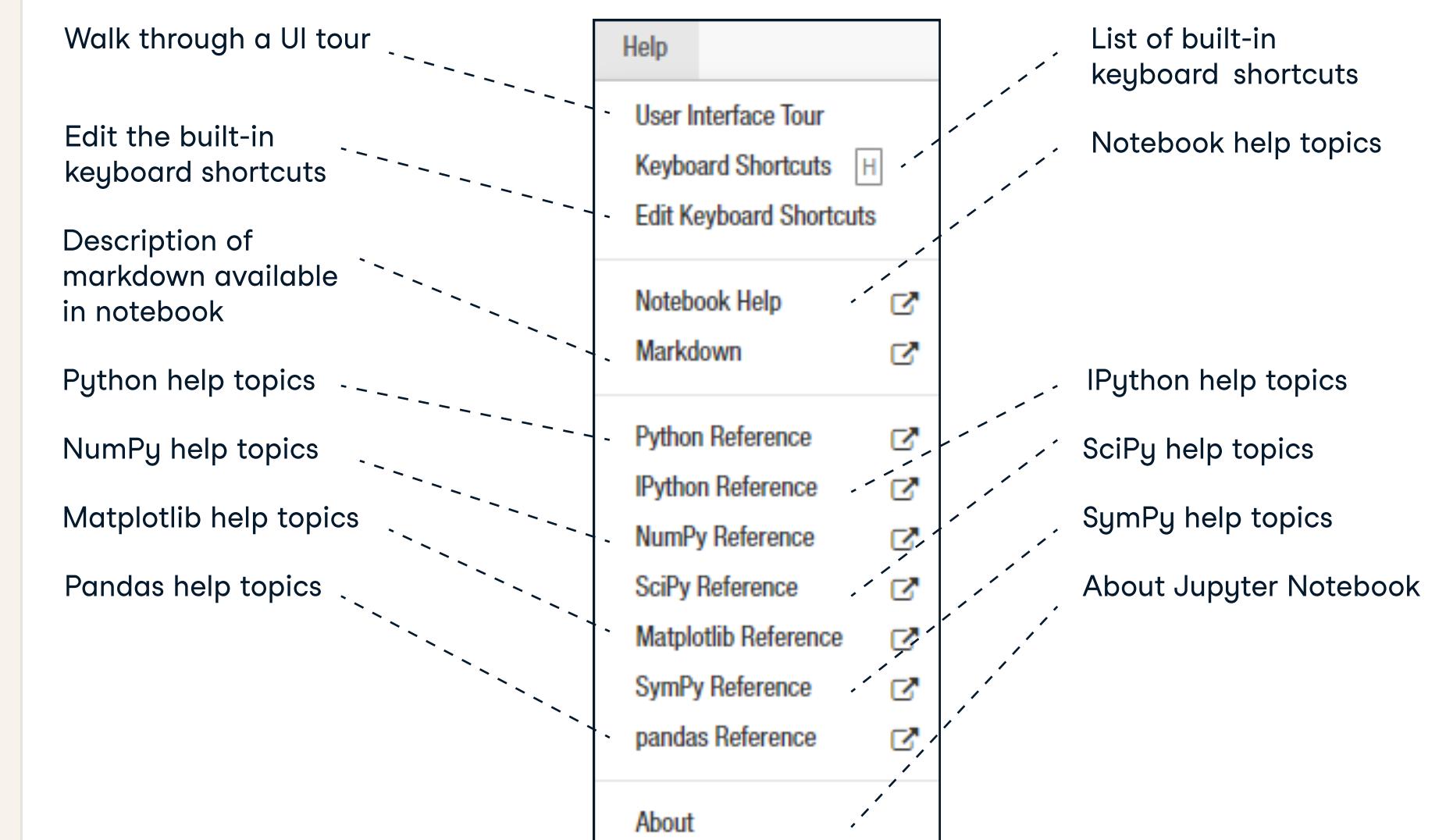
Executing Cells



View Cells



Asking For Help



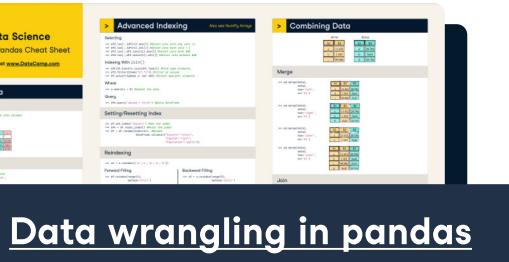
Python Basics

Python Cheat Sheet for Beginners

Learn Python online at www.DataCamp.com

> How to use this cheat sheet

Python is the most popular programming language in data science. It is easy to learn and comes with a wide array of powerful libraries for data analysis. This cheat sheet provides beginners and intermediate users a guide to starting using python. Use it to jump-start your journey with python. If you want more detailed Python cheat sheets, check out the following cheat sheets below:



> Accessing help and getting object types

```
1 + 1 # Everything after the hash symbol is ignored by Python
help(max) # Display the documentation for the max function
type('a') # Get the type of an object - this returns str
```

> Importing packages

Python packages are a collection of useful tools developed by the open-source community. They extend the capabilities of the python language. To install a new package (for example, pandas), you can go to your command prompt and type in pip install pandas. Once a package is installed, you can import it as follows.

```
import pandas # Import a package without an alias
import pandas as pd # Import a package with an alias
from pandas import DataFrame # Import an object from a package
```

> The working directory

The working directory is the default file path that python reads or saves files into. An example of the working directory is "C://file/path". The os library is needed to set and get the working directory.

```
import os # Import the operating system package
os.getcwd() # Get the current directory
os.chdir("new/working/directory") # Set the working directory to a new file path
```

> Operators

Arithmetic operators

```
102 + 37 # Add two numbers with +
102 - 37 # Subtract a number with -
4 * 6 # Multiply two numbers with *
22 / 7 # Divide a number by another with /
```

```
22 // 7 # Integer divide a number with //
3 ** 4 # Raise to the power with **
22 % 7 # Returns 1 # Get the remainder after division with %
```

Assignment operators

```
a = 5 # Assign a value to a
x[0] = 1 # Change the value of an item in a list
```

Numeric comparison operators

```
3 == 3 # Test for equality with ==
3 != 3 # Test for inequality with !=
3 > 1 # Test greater than with >
```

```
3 >= 3 # Test greater than or equal to with >=
3 < 4 # Test less than with <
3 <= 4 # Test less than or equal to with <=
```

Logical operators

```
~(2 == 2) # Logical NOT with ~
(1 != 1) & (1 < 1) # Logical AND with &
```

```
(1 >= 1) | (1 < 1) # Logical OR with |
(1 != 1) ^ (1 < 1) # Logical XOR with ^
```

> Getting started with lists

A list is an ordered and changeable sequence of elements. It can hold integers, characters, floats, strings, and even objects.

Creating lists

```
# Create lists with [], elements separated by commas
x = [1, 3, 2]
```

List functions and methods

```
x.sorted(x) # Return a sorted copy of the list e.g., [1,2,3]
x.sort() # Sorts the list in-place (replaces x)
reversed(x) # Reverse the order of elements in x e.g., [2,3,1]
x.reversed() # Reverse the list in-place
x.count(2) # Count the number of element 2 in the list
```

Selecting list elements

Python lists are zero-indexed (the first element has index 0). For ranges, the first element is included but the last is not.

```
# Define the list
x = ['a', 'b', 'c', 'd', 'e']           x[1:3] # Select 1st (inclusive) to 3rd (exclusive)
x[0] # Select the 0th element in the list x[2:] # Select the 2nd to the end
x[-1] # Select the last element in the list x[:3] # Select 0th to 3rd (exclusive)
```

Concatenating lists

```
# Define the x and y lists
x = [1, 3, 6]           x + y # Returns [1, 3, 6, 10, 15, 21]
y = [10, 15, 21]         3 * x # Returns [1, 3, 6, 1, 3, 6, 1, 3, 6]
```

> Getting started with dictionaries

A dictionary stores data values in key-value pairs. That is, unlike lists which are indexed by position, dictionaries are indexed by their keys, the names of which must be unique.

Creating dictionaries

```
# Create a dictionary with {}
{'a': 1, 'b': 4, 'c': 9}
```

Dictionary functions and methods

```
x = {'a': 1, 'b': 2, 'c': 3} # Define the x dictionary
x.keys() # Get the keys of a dictionary, returns dict_keys(['a', 'b', 'c'])
x.values() # Get the values of a dictionary, returns dict_values([1, 2, 3])
```

Selecting dictionary elements

```
x['a'] # 1 # Get a value from a dictionary by specifying the key
```

> NumPy arrays

NumPy is a python package for scientific computing. It provides multidimensional array objects and efficient operations on them. To import NumPy, you can run this Python code import numpy as np

Creating arrays

```
# Convert a python list to a NumPy array
np.array([1, 2, 3]) # Returns array([1, 2, 3])
# Return a sequence from start (inclusive) to end (exclusive)
np.arange(1,5) # Returns array([1, 2, 3, 4])
# Return a stepped sequence from start (inclusive) to end (exclusive)
np.arange(1,5,2) # Returns array([1, 3])
# Repeat values n times
np.repeat([1, 3, 6], 3) # Returns array([1, 1, 1, 3, 3, 3, 6, 6, 6])
# Repeat values n times
np.tile([1, 3, 6], 3) # Returns array([1, 3, 6, 1, 3, 6, 1, 3, 6])
```

> Math functions and methods

All functions take an array as the input.

```
np.log(x) # Calculate logarithm
np.exp(x) # Calculate exponential
np.max(x) # Get maximum value
np.min(x) # Get minimum value
np.sum(x) # Calculate sum
np.mean(x) # Calculate mean
np.quantile(x, q) # Calculate q-th quantile
np.round(x, n) # Round to n decimal places
np.var(x) # Calculate variance
np.std(x) # Calculate standard deviation
```

> Getting started with characters and strings

```
# Create a string with double or single quotes
"DataCamp"
```

```
# Embed a quote in string with the escape character \
"He said, \"DataCamp\""
```

```
# Create multi-line strings with triple quotes
"""
```

```
A Frame of Data
Tidy, Mine, Analyze It
Now You Have Meaning
Citation: https://mdsr-book.github.io/haikus.html
"""

str[0] # Get the character at a specific position
str[0:2] # Get a substring from starting to ending index (exclusive)
```

Combining and splitting strings

```
"Data" + "Framed" # Concatenate strings with +, this returns 'DataFramed'
3 * "data" # Repeat strings with *, this returns 'data data data'
"beekeepers".split("e") # Split a string on a delimiter, returns ['b', ' ', 'k', ' ', 'p', 'rs']
```

Mutate strings

```
str = "Jack and Jill" # Define str
str.upper() # Convert a string to uppercase, returns 'JACK AND JILL'
str.lower() # Convert a string to lowercase, returns 'jack and jill'
str.title() # Convert a string to title case, returns 'Jack And Jill'
str.replace("J", "P") # Replaces matches of a substring with another, returns 'Pack and Pill'
```

> Getting started with DataFrames

Pandas is a fast and powerful package for data analysis and manipulation in python. To import the package, you can use import pandas as pd. A pandas DataFrame is a structure that contains two-dimensional data stored as rows and columns. A pandas series is a structure that contains one-dimensional data.

Creating DataFrames

```
# Create a dataframe from a dictionary
pd.DataFrame({
  'a': [1, 2, 3],
  'b': np.array([4, 4, 6]),
  'c': ['x', 'x', 'y']
})
```

```
# Create a dataframe from a list of dictionaries
pd.DataFrame([
  {'a': 1, 'b': 4, 'c': 'x'},
  {'a': 1, 'b': 4, 'c': 'x'},
  {'a': 3, 'b': 6, 'c': 'y'}
])
```

Selecting DataFrame Elements

Select a row, column or element from a dataframe. Remember: all positions are counted from zero, not one.

```
# Select the 3rd row
df.iloc[3]
# Select one column by name
df['col1']
# Select multiple columns by names
df[['col1', 'col2']]
# Select 2nd column
df.iloc[:, 2]
# Select the element in the 3rd row, 2nd column
df.iloc[3, 2]
```

Manipulating DataFrames

```
# Concatenate DataFrames vertically
pd.concat([df, df])
# Concatenate DataFrames horizontally
pd.concat([df, df], axis="columns")
# Get rows matching a condition
df.query("logical_condition")
# Drop columns by name
df.drop(columns=['col_name'])
# Rename columns
df.rename(columns={"oldname": "newname"})
# Add a new column
df.assign(temp_f=9 / 5 * df['temp_c'] + 32)
# Calculate the mean of each column
df.mean()
# Get summary statistics by column
df.agg(aggregation_function)
# Get unique rows
df.drop_duplicates()
# Sort by values in a column
df.sort_values(by='col_name')
# Get rows with largest values in a column
df.nlargest(n, 'col_name')
```

Python For Data Science

Importing Data Cheat Sheet

Learn Python online at www.DataCamp.com

> Importing Data in Python

Most of the time, you'll use either NumPy or pandas to import your data:

```
>>> import numpy as np
>>> import pandas as pd
```

> Help

```
>>> np.info(np.ndarray.dtype)
>>> help(pd.read_csv)
```

> Text Files

Plain Text Files

```
>>> filename = 'huck_finn.txt'
>>> file = open(filename, mode='r') #Open the file for reading
>>> text = file.read() #Read a file's contents
>>> print(file.closed) #Check whether file is closed
>>> file.close() #Close file
>>> print(text)

Using the context manager with
```

```
>>> with open('huck_finn.txt', 'r') as file:
    print(file.readline()) #Read a single line
    print(file.readline())
    print(file.readline())
```

Table Data: Flat Files

Importing Flat Files with NumPy

```
>>> filename = 'huck_finn.txt'
>>> file = open(filename, mode='r') #Open the file for reading
>>> text = file.read() #Read a file's contents
>>> print(file.closed) #Check whether file is closed
>>> file.close() #Close file
>>> print(text)
```

Files with one data type

```
>>> filename = 'mnist.txt'
>>> data = np.loadtxt(filename,
    delimiter=',', #String used to separate values
    skiprows=2, #Skip the first 2 lines
    usecols=[0,2], #Read the 1st and 3rd column
    dtype=str) #The type of the resulting array
```

Files with mixed data type

```
>>> filename = 'titanic.csv'
>>> data = np.genfromtxt(filename,
    delimiter=',',
    names=True, #Look for column header
    dtype=None)
>>> data_array = np.recfromcsv(filename)
#The default dtype of the np.recfromcsv() function is None
```

Importing Flat Files with Pandas

```
>>> filename = 'winequality-red.csv'
>>> data = pd.read_csv(filename,
    nrows=5, #Number of rows of file to read
    header=None, #Row number to use as col names
    sep='\t', #Delimiter to use
    comment='#', #Character to split comments
    na_values=['']) #String to recognize as NA/Nan
```

> Exploring Your Data

NumPy Arrays

```
>>> data_array.dtype #Data type of array elements
>>> data_array.shape #Array dimensions
>>> len(data_array) #Length of array
```

Pandas DataFrames

```
>>> df.head() #Return first DataFrame rows
>>> df.tail() #Return last DataFrame rows
>>> df.index #Describe index
>>> df.columns #Describe DataFrame columns
>>> df.info() #Info on DataFrame
>>> data_array = data.values #Convert a DataFrame to an a NumPy array
```

> SAS File

```
>>> from sas7bdat import SAS7BDAT
>>> with SAS7BDAT('urbanpop.sas7bdat') as file:
    df_sas = file.to_data_frame()
```

> Stata File

```
>>> data = pd.read_stata('urbanpop.dta')
```

> Excel Spreadsheets

```
>>> file = 'urbanpop.xlsx'
>>> data = pd.ExcelFile(file)
>>> df_sheet2 = data.parse('1960-1966',
    skiprows=[0],
    names=['Country',
    'AAM: War(2002)'])
>>> df_sheet1 = data.parse(0,
    parse_cols=[0],
    skiprows=[0],
    names=['Country'])

To access the sheet names, use the sheet_names attribute:
>>> data.sheet_names
```

> Relational Databases

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///Northwind.sqlite')
Use the table_names() method to fetch a list of table names:
>>> table_names = engine.table_names()
```

Querying Relational Databases

```
>>> con = engine.connect()
>>> rs = con.execute("SELECT * FROM Orders")
>>> df = pd.DataFrame(rs.fetchall())
>>> df.columns = rs.keys()
>>> con.close()

Using the context manager with
```

```
>>> with engine.connect() as con:
    rs = con.execute("SELECT OrderID FROM Orders")
    df = pd.DataFrame(rs.fetchmany(size=5))
    df.columns = rs.keys()
```

Querying relational databases with pandas

```
>>> df = pd.read_sql_query("SELECT * FROM Orders", engine)
```

> Pickled Files

```
>>> import pickle
>>> with open('pickled_fruit.pkl', 'rb') as file:
    pickled_data = pickle.load(file)
```

> Matlab Files

```
>>> import scipy.io
>>> filename = 'workspace.mat'
>>> mat = scipy.io.loadmat(filename)
```

> HDF5 Files

```
>>> import h5py
>>> filename = 'H-H1_LOSC_4_v1-815411200-4096.hdf5'
>>> data = h5py.File(filename, 'r')
```

> Exploring Dictionaries

Querying relational databases with pandas

```
>>> print(mat.keys()) #Print dictionary keys
>>> for key in data.keys(): #Print dictionary keys
    print(key)
meta
quality
strain
>>> pickled_data.values() #Return dictionary values
>>> print(mat.items()) #Returns items in list format of (key, value) tuple pairs
```

Accessing Data Items with Keys

```
>>> for key in data['meta'].keys(): #Explore the HDF5
    structure
        print(key)
Description
DescriptionURL
Detector
Duration
GPSstart
Observatory
Type
UTCstart
#Retrieve the value for a key
>>> print(data['meta']['Description'].value)
```

> Navigating Your FileSystem

Magic Commands

```
!ls #List directory contents of files and directories
%cd .. #Change current working directory
%pwd #Return the current working directory path
```

OS Library

```
>>> import os
>>> path = "/usr/tmp"
>>> wd = os.getcwd() #Store the name of current directory in a string
>>> os.listdir(wd) #Output contents of the directory in a list
>>> os.chdir(path) #Change current working directory
>>> os.rename("test1.txt", "test2.txt") #Rename a file
    "test2.txt")
>>> os.remove("test1.txt") #Delete an existing file
>>> os.mkdir("newdir") #Create a new directory
```



Python For Data Science

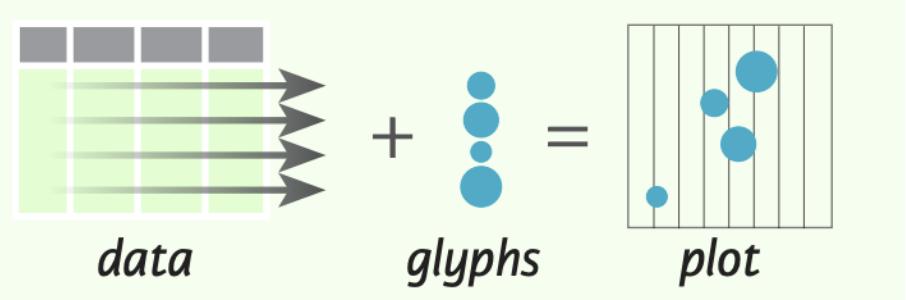
Bokeh Cheat Sheet

Learn Bokeh online at www.DataCamp.com
taught by Bryan Van de Ven, core contributor

Plotting With Bokeh

The Python interactive visualization library **Bokeh** enables high-performance visual presentation of large datasets in modern web browsers.

Bokeh's mid-level general purpose bokeh.plotting interface is centered around two main components: `data` and `glyphs`.



The basic steps to creating plots with the bokeh.plotting interface are:

1. Prepare some data (Python lists, NumPy arrays, Pandas DataFrames and other sequences of values)
2. Create a new plot
3. Add renderers for your data, with visual customizations
4. Specify where to generate the output
5. Show or save the results

```
>>> from bokeh.plotting import figure
>>> from bokeh.io import output_file, show
>>> x = [1, 2, 3, 4, 5] #Step 1
>>> y = [6, 7, 2, 4, 5]
>>> p = figure(title="simple line example", #Step 2
x_axis_label='x',
y_axis_label='y')
>>> p.line(x, y, legend="Temp.", line_width=2) #Step 3
>>> output_file("lines.html") #Step 4
>>> show(p) #Step 5
```

1 Data

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

Under the hood, your data is converted to Column Data Sources.

You can also do this manually:

```
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.DataFrame(np.array([[33.9, 4, 65, 'US'],
[32.4, 4, 66, 'Asia'],
[21.4, 4, 109, 'Europe']]),
columns=['mpg', 'cyl', 'hp', 'origin'],
index=['Toyota', 'Fiat', 'Volvo'])

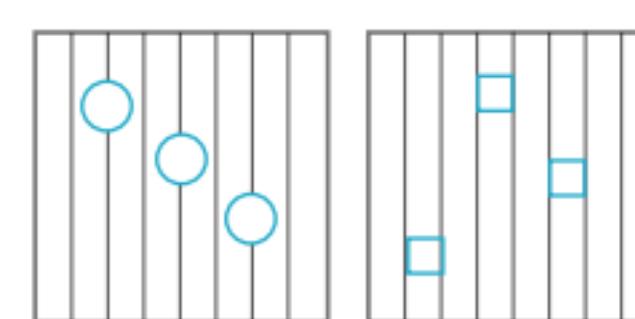
>>> from bokeh.models import ColumnDataSource
>>> cds_df = ColumnDataSource(df)
```

2 Plotting

```
>>> from bokeh.plotting import figure
>>> p1 = figure(plot_width=300, tools='pan,box_zoom')
>>> p2 = figure(plot_width=300, plot_height=300,
x_range=(0, 8), y_range=(0, 8))
>>> p3 = figure()
```

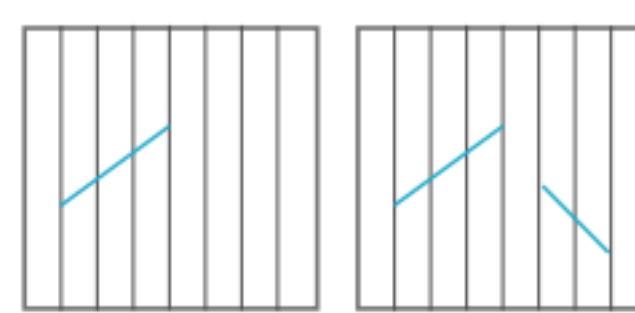
3 Renderers & Visual Customizations

Glyphs



Scatter Markers

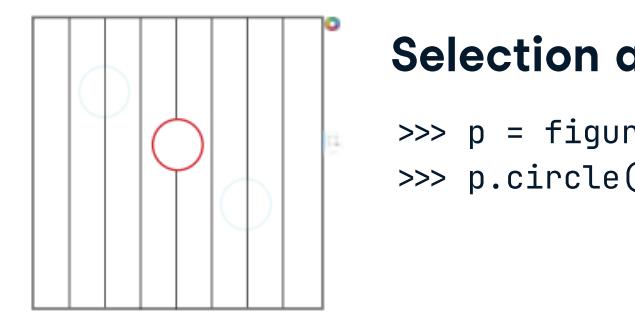
```
>>> p1.circle(np.array([1,2,3]), np.array([3,2,1]),
fill_color='white')
>>> p2.square(np.array([1.5,3.5,5.5]), [1,4,3],
color='blue', size=1)
```



Line Glyphs

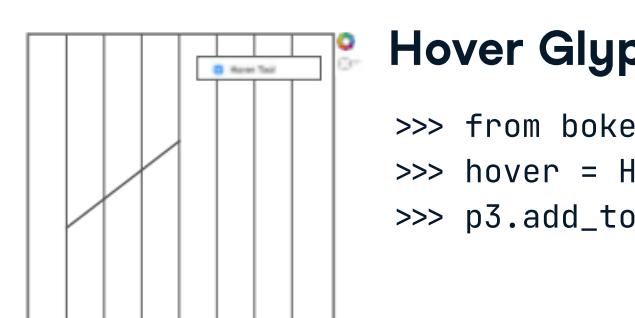
```
>>> p1.line([1,2,3,4], [3,4,5,6], line_width=2)
>>> p2.multi_line(pd.DataFrame([[1,2,3],[5,6,7]]),
pd.DataFrame([[3,4,5],[3,2,1]]),
color="blue")
```

Customized Glyphs



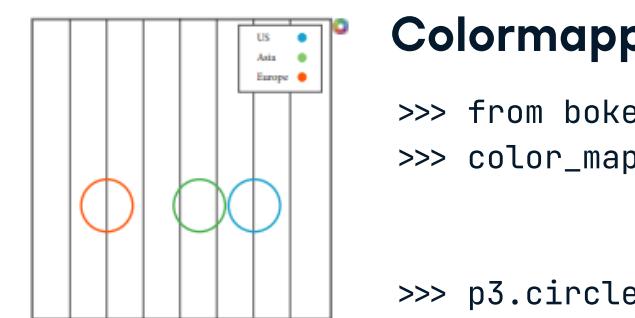
Selection and Non-Selection Glyphs

```
>>> p = figure(tools='box_select')
>>> p.circle('mpg', 'cyl', source=cds_df,
selection_color='red',
nonselection_alpha=0.1)
```



Hover Glyphs

```
>>> from bokeh.models import HoverTool
>>> hover = HoverTool(tooltips=None, mode='vline')
>>> p3.add_tools(hover)
```



Colormapping

```
>>> from bokeh.models import CategoricalColorMapper
>>> color_mapper = CategoricalColorMapper(
factors=['US', 'Asia', 'Europe'],
palette=['blue', 'red', 'green'])
>>> p3.circle('mpg', 'cyl', source=cds_df,
color=dict(field='origin',
transform=color_mapper),
legend='Origin')
```

Also see [Data](#)

Legend Location

Inside Plot Area

```
>>> p.legend.location = 'bottom_left'
```

Outside Plot Area

```
>>> from bokeh.models import Legend
>>> r1 = p2.asterisk(np.array([1,2,3]), np.array([3,2,1]))
>>> r2 = p2.line([1,2,3,4], [3,4,5,6])
>>> legend = Legend(items=[("One", [p1, r1]), ("Two", [r2])],
location=(0, -30))
>>> p.add_layout(legend, 'right')
```

Legend Orientation

```
>>> p.legend.orientation = "horizontal"
>>> p.legend.orientation = "vertical"
```

Legend Background & Border

```
>>> p.legend.border_line_color = "navy"
>>> p.legend.background_fill_color = "white"
```

Rows & Columns Layout

Rows

```
>>> from bokeh.layouts import row
>>> layout = row(p1,p2,p3)
```

Columns

```
>>> from bokeh.layouts import column
>>> layout = column(p1,p2,p3)
```

Nesting Rows & Columns

```
>>> layout = row(column(p1,p2), p3)
```

Grid Layout

```
>>> from bokeh.layouts import gridplot
>>> row1 = [p1,p2]
>>> row2 = [p3]
>>> layout = gridplot([[p1,p2],[p3]])
```

Tabbed Layout

```
>>> from bokeh.models.widgets import Panel, Tabs
>>> tab1 = Panel(child=p1, title="tab1")
>>> tab2 = Panel(child=p2, title="tab2")
>>> layout = Tabs(tabs=[tab1, tab2])
```

Linked Plots

Linked Axes

```
>>> p2.x_range = p1.x_range
>>> p2.y_range = p1.y_range
```

Linked Brushing

```
>>> p4 = figure(plot_width = 100, tools='box_select,lasso_select')
>>> p4.circle('mpg', 'cyl', source=cds_df)
>>> p5 = figure(plot_width = 200, tools='box_select,lasso_select')
>>> p5.circle('mpg', 'hp', source=cds_df)
>>> layout = row(p4,p5)
```

4 Output & Export

Notebook

```
>>> from bokeh.io import output_notebook, show
>>> output_notebook()
```

HTML

Standalone HTML

```
>>> from bokeh.embed import file_html
>>> from bokeh.resources import CDN
>>> html = file_html(p, CDN, "my_plot")

>>> from bokeh.io import output_file, show
>>> output_file('my_bar_chart.html', mode='cdn')
```

Components

```
>>> from bokeh.embed import components
>>> script, div = components(p)
```

PNG

```
>>> from bokeh.io import export_png
>>> export_png(p, filename="plot.png")
```

SVG

```
>>> from bokeh.io import export_svgs
>>> p.output_backend = "svg"
>>> export_svgs(p, filename="plot.svg")
```

5 Show or Save Your Plots

```
>>> show(p1)
>>> show(layout)
>>> save(p1)
>>> save(layout)
```



Learn Data Skills Online at www.DataCamp.com



Python For Data Science

Keras Cheat Sheet

Learn Keras online at www.DataCamp.com

Keras

Keras is a powerful and easy-to-use deep learning library for Theano and TensorFlow that provides a high-level neural networks API to develop and evaluate deep learning models.

A Basic Example

```
>>> import numpy as np
>>> from tensorflow.keras.models import Sequential
>>> from tensorflow.keras.layers import Dense
>>> data = np.random.random((1000,100))
>>> labels = np.random.randint(2,size=(1000,1))
>>> model = Sequential()
>>> model.add(Dense(32,
    activation='relu',
    input_dim=100))
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile(optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['accuracy'])
>>> model.fit(data,labels,epochs=10,batch_size=32)
>>> predictions = model.predict(data)
```

Data

Your data needs to be stored as NumPy arrays or as a list of NumPy arrays. Ideally, you split the data in training and test sets, for which you can also resort to the `train_test_split` module of `sklearn.cross_validation`.

Keras Data Sets

```
>>> from tensorflow.keras.datasets import boston_housing, mnist, cifar10, imdb
>>> (x_train,y_train),(x_test,y_test) = mnist.load_data()
>>> (x_train2,y_train2),(x_test2,y_test2) = boston_housing.load_data()
>>> (x_train3,y_train3),(x_test3,y_test3) = cifar10.load_data()
>>> (x_train4,y_train4),(x_test4,y_test4) = imdb.load_data(num_words=20000)
>>> num_classes = 10
```

Other

```
>>> from urllib.request import urlopen
>>> data =
np.loadtxt(urlopen("http://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data"),delimiter=",")
>>> X = data[:,0:8]
>>> y = data[:,8]
```

Preprocessing

Sequence Padding

```
>>> from tensorflow.keras.preprocessing import sequence
>>> x_train4 = sequence.pad_sequences(x_train4,maxlen=80)
>>> x_test4 = sequence.pad_sequences(x_test4,maxlen=80)
```

One-Hot Encoding

```
>>> from tensorflow.keras.utils import to_categorical
>>> Y_train = to_categorical(y_train, num_classes)
>>> Y_test = to_categorical(y_test, num_classes)
>>> Y_train3 = to_categorical(y_train3, num_classes)
>>> Y_test3 = to_categorical(y_test3, num_classes)
```

> Model Architecture

Sequential Model

```
>>> from tensorflow.keras.models import Sequential
>>> model = Sequential()
>>> model2 = Sequential()
>>> model3 = Sequential()
```

Multilayer Perceptron (MLP)

Binary Classification

```
>>> from tensorflow.keras.layers import Dense
>>> model.add(Dense(12,
    input_dim=8,
    kernel_initializer='uniform',
    activation='relu'))
>>> model.add(Dense(8,kernel_initializer='uniform',activation='relu'))
>>> model.add(Dense(1,kernel_initializer='uniform',activation='sigmoid'))
```

Multi-Class Classification

```
>>> from tensorflow.keras.layers import Dropout
>>> model.add(Dense(512,activation='relu',input_shape=(784,)))
>>> model.add(Dropout(0.2))
>>> model.add(Dense(512,activation='relu'))
>>> model.add(Dropout(0.2))
>>> model.add(Dense(10,activation='softmax'))
```

Regression

```
>>> model.add(Dense(64,activation='relu',input_dim=train_data.shape[1]))
>>> model.add(Dense(1))
```

Convolutional Neural Network (CNN)

```
>>> from tensorflow.keras.layers import Activation,Conv2D,MaxPooling2D,Flatten
>>> model2.add(Conv2D(32,(3,3),padding='same',input_shape=x_train.shape[1:]))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(32,(3,3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))
>>> model2.add(Conv2D(64,(3,3), padding='same'))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(64,(3,3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))
>>> model2.add(Flatten())
>>> model2.add(Dense(512))
>>> model2.add(Activation('relu'))
>>> model2.add(Dropout(0.5))
>>> model2.add(Dense(num_classes))
>>> model2.add(Activation('softmax'))
```

Recurrent Neural Network (RNN)

```
>>> from tensorflow.keras.layers import Embedding,LSTM
>>> model3.add(Embedding(20000,128))
>>> model3.add(LSTM(128,dropout=0.2,recurrent_dropout=0.2))
>>> model3.add(Dense(1,activation='sigmoid'))
```

> Prediction

```
>>> model3.predict(x_test4, batch_size=32)
>>> model3.predict_classes(x_test4,batch_size=32)
```

Also see NumPy & Scikit-Learn

Train and Test Sets

```
>>> from sklearn.model_selection import train_test_split
>>> X_train5,X_test5,y_train5,y_test5 = train_test_split(x, y,
    test_size=0.33,
    random_state=42)
```

Standardization/Normalization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(x_train2)
>>> standardized_X = scaler.transform(x_train2)
>>> standardized_X_test = scaler.transform(x_test2)
```

> Inspect Model

```
>>> model.output_shape #Model output shape
>>> model.summary() #Model summary representation
>>> model.get_config() #Model configuration
>>> model.get_weights() #List all weight tensors in the model
```

> Compile Model

MLP: Binary Classification

```
>>> model.compile(optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy'])
```

MLP: Multi-Class Classification

```
>>> model.compile(optimizer='rmsprop',
    loss='categorical_crossentropy',
    metrics=['accuracy'])
```

MLP: Regression

```
>>> model.compile(optimizer='rmsprop',
    loss='mse',
    metrics=['mae'])
```

Recurrent Neural Network

```
>>> model3.compile(loss='binary_crossentropy',
    optimizer='adam',
    metrics=['accuracy'])
```

> Model Training

```
>>> model3.fit(x_train4,
    y_train4,
    batch_size=32,
    epochs=15,
    verbose=1,
    validation_data=(x_test4,y_test4))
```

> Evaluate Your Model's Performance

```
>>> score = model3.evaluate(x_test,
    y_test,
    batch_size=32)
```

> Save/ Reload Models

```
>>> from tensorflow.keras.models import load_model
>>> model3.save('model_file.h5')
>>> my_model = load_model('my_model.h5')
```

> Model Fine-tuning

Optimization Parameters

```
>>> from tensorflow.keras.optimizers import RMSprop
>>> opt = RMSprop(lr=0.0001, decay=1e-6)
>>> model2.compile(loss='categorical_crossentropy',
    optimizer=opt,
    metrics=['accuracy'])
```

Early Stopping

```
>>> from tensorflow.keras.callbacks import EarlyStopping
>>> early_stopping_monitor = EarlyStopping(patience=2)
>>> model3.fit(x_train4,
    y_train4,
    batch_size=32,
    epochs=15,
    validation_data=(x_test4,y_test4),
    callbacks=[early_stopping_monitor])
```

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[0,0].barh([0.5,1,2.5],[0,1,2]) #Plot horizontal 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

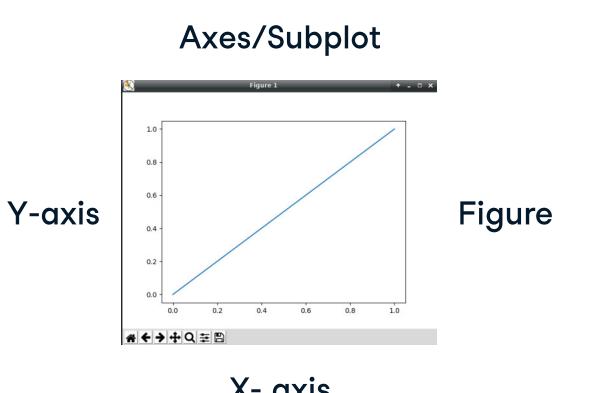
```
>>> axes[0,1].arrow(0,0,0.5,0.5) #Add an arrow to the axes
>>> axes[1,1].quiver(y,z) #Plot a 2D field of arrows
>>> axes[0,1].streamplot(X,Y,U,V) #Plot a 2D field of arrows
```

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**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
```

# Python For Data Science

## NumPy Cheat Sheet

Learn NumPy online at [www.DataCamp.com](http://www.DataCamp.com)

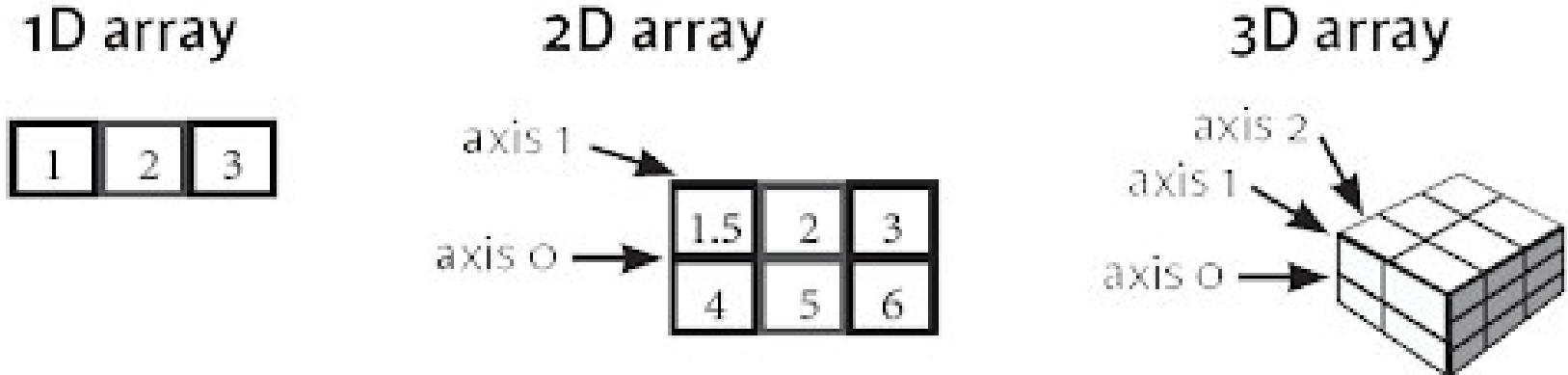
## Numpy

The NumPy library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:

```
>>> import numpy as np
```

## NumPy Arrays



## Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([(1.5,2,3), (4,5,6)],[(3,2,1), (4,5,6)]), dtype = float)
```

## Initial Placeholders

```
>>> np.zeros((3,4)) #Create an array of zeros
>>> np.ones((2,3,4),dtype=np.int16) #Create an array of ones
>>> d = np.arange(10,25,5) #Create an array of evenly spaced values (step value)
>>> np.linspace(0,2,9) #Create an array of evenly spaced values (number of samples)
>>> e = np.full((2,2),7) #Create a constant array
>>> f = np.eye(2) #Create a 2x2 identity matrix
>>> np.random.random((2,2)) #Create an array with random values
>>> np.empty((3,2)) #Create an empty array
```

## I/O

### Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

### Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

## Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

## Inspecting Your Array

```
>>> a.shape #Array dimensions
>>> len(a) #Length of array
>>> b.ndim #Number of array dimensions
>>> e.size #Number of array elements
>>> b.dtype #Data type of array elements
>>> b.dtype.name #Name of data type
>>> b.astype(int) #Convert an array to a different type
```

## Data Types

```
>>> np.int64 #Signed 64-bit integer types
>>> np.float32 #Standard double-precision floating point
>>> np.complex #Complex numbers represented by 128 floats
>>> np.bool #Boolean type storing TRUE and FALSE values
>>> np.object #Python object type
>>> np.string_ #Fixed-length string type
>>> np_unicode_ #Fixed-length unicode type
```

## Array Mathematics

### Arithmetic Operations

```
>>> g = a - b #Subtraction
array([[-0.5, 0. , 0.],
 [-3. , -3. , -3.]])
>>> np.subtract(a,b) #Subtraction
>>> b + a #Addition
array([[2.5, 4. , 6.],
 [5. , 7. , 9.]])
>>> np.add(b,a) Addition
>>> a / b #Division
array([[0.66666667, 1. , 1.],
 [0.25 , 0.4 , 0.5]])
>>> np.divide(a,b) #Division
>>> a * b #Multiplication
array([[1.5, 4. , 9.],
 [4. , 10. , 18.]])
>>> np.multiply(a,b) #Multiplication
>>> np.exp(b) #Exponentiation
>>> np.sqrt(b) #Square root
>>> np.sin(a) #Print sines of an array
>>> np.cos(b) #Element-wise cosine
>>> np.log(a) #Element-wise natural logarithm
>>> e.dot(f) #Dot product
array([[7., 7.],
 [7., 7.]])
```

### Comparison

```
>>> a == b #Element-wise comparison
array([[False, True, True],
 [False, False, False]], dtype=bool)
>>> a < 2 #Element-wise comparison
array[[True, False, False], dtype=bool]
>>> np.array_equal(a, b) #Array-wise comparison
```

### Aggregate Functions

```
>>> a.sum() #Array-wise sum
>>> a.min() #Array-wise minimum value
>>> b.max(axis=0) #Maximum value of an array row
>>> b.cumsum(axis=1) #Cumulative sum of the elements
>>> a.mean() #Mean
>>> np.median(b) #Median
>>> np.correlcoef(a) #Correlation coefficient
>>> np.std(b) #Standard deviation
```

## Copying Arrays

```
>>> h = a.view() #Create a view of the array with the same data
>>> np.copy(a) #Create a copy of the array
>>> h = a.copy() #Create a deep copy of the array
```

## Sorting Arrays

```
>>> a.sort() #Sort an array
>>> c.sort(axis=0) #Sort the elements of an array's axis
```

## Subsetting, Slicing, Indexing

### Subsetting

```
>>> a[2] #Select the element at the 2nd index
3
>>> b[1,2] #Select the element at row 1 column 2 (equivalent to b[1][2])
6.0
```

|     |   |   |
|-----|---|---|
| 1   | 2 | 3 |
| 1.5 | 2 | 3 |
| 4   | 5 | 6 |

### Slicing

```
>>> a[0:2] #Select items at index 0 and 1
array([1, 2])
>>> b[0:2,1] #Select items at rows 0 and 1 in column 1
array([2., 2.])
>>> b[:,1] #Select all items at row 0 (equivalent to b[0:, 1])
array([[1.5, 2., 3.]])
>>> c[1,...] #Same as [1,:,:]
array([[3., 2., 1.],
 [4., 5., 6.]])
>>> a[: :-1] #Reversed array a array([3, 2, 1])
```

|     |   |   |
|-----|---|---|
| 1   | 2 | 3 |
| 1.5 | 2 | 3 |
| 4   | 5 | 6 |

### Boolean Indexing

```
>>> a[a<2] #Select elements from a less than 2
array([1])
```

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
|---|---|---|

### Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]] #Select elements (1,0),(0,1),(1,2) and (0,0)
array([4. , 2. , 6. , 1.5])
>>> b[[1, 0, 1, 0]][:, [0,1,2,0]] #Select a subset of the matrix's rows and columns
array([[4. , 5. , 6. , 4.],
 [1.5, 2. , 3. , 1.5],
 [4. , 5. , 6. , 4.],
 [1.5, 2. , 3. , 1.5]])
```

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
|---|---|---|

## Array Manipulation

### Transposing Array

```
>>> i = np.transpose(b) #Permute array dimensions
>>> i.T #Permute array dimensions
```

### Changing Array Shape

```
>>> b.ravel() #Flatten the array
>>> g.reshape(3,-2) #Reshape, but don't change data
```

### Adding/Removing Elements

```
>>> h.resize((2,6)) #Return a new array with shape (2,6)
>>> np.append(h,g) #Append items to an array
>>> np.insert(a, 1, 5) #Insert items in an array
>>> np.delete(a,[1]) #Delete items from an array
```

### Combining Arrays

```
>>> np.concatenate((a,d),axis=0) #Concatenate arrays
array([[1., 2., 3., 10.],
 [2., 15.],
 [3., 20.]])
>>> np.vstack((a,b)) #Stack arrays vertically (row-wise)
array([[1., 2., 3.],
 [1.5, 2., 3.],
 [4., 5., 6.]]))
>>> np.r_[e,f] #Stack arrays vertically (row-wise)
>>> np.hstack((e,f)) #Stack arrays horizontally (column-wise)
array([[7., 7., 1., 0.],
 [7., 7., 0., 1.]])
>>> np.column_stack((a,d)) #Create stacked column-wise arrays
array([[1, 10],
 [2, 15],
 [3, 20]])
>>> np.c_[a,d] #Create stacked column-wise arrays
```

### Splitting Arrays

```
>>> np.hsplit(a,3) #Split the array horizontally at the 3rd index
[array([1]),array([2]),array([3])]
>>> np.vsplit(c,2) #Split the array vertically at the 2nd index
[array([[1.5, 2. , 1.]]),
 [4. , 5. , 6.]])
array([[[3., 2., 3.]],
 [4., 5., 6.]])
```



# Python For Data Science

## Pandas Basics Cheat Sheet

Learn Pandas Basics online at [www.DataCamp.com](http://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

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

|         | Country | Capital   | Population |
|---------|---------|-----------|------------|
| Index → | Belgium | Brussels  | 11190846   |
| 0       | India   | New Delhi | 1303171035 |
| 1       | Brazil  | Brasilia  | 207847528  |

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
 'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
 'Population': [11190846, 1303171035, 207847528]}
>>> df = pd.DataFrame(data,
 columns=['Country', 'Capital', 'Population'])
```

### Dropping

```
>>> s.drop(['a', 'c']) #Drop values from rows (axis=0)
>>> df.drop('Country', axis=1) #Drop values from columns(axis=1)
```

### Asking For Help

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

### Sort & Rank

```
>>> df.sort_index() #Sort by labels along an axis
>>> df.sort_values(by='Country') #Sort by the values along an axis
>>> df.rank() #Assign ranks to entries
```

### > 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.xlsx')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

#### 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)
```

### > Selection

Also see NumPy Arrays

#### Getting

```
>>> s['b'] #Get one element
-5
>>> df[1:] #Get subset of a DataFrame
 Country Capital Population
1 India New Delhi 1303171035
2 Brazil Brasilia 207847528
```

#### Selecting, Boolean Indexing & Setting

##### By Position

```
>>> df.iloc[[0],[0]] #Select single value by row & column
'Belgium'
>>> df.iat[[0],[0]]
'Belgium'
```

##### By Label

```
>>> df.loc[[0], ['Country']] #Select single value by row & column labels
'Belgium'
>>> df.at[[0], ['Country']]
'Belgium'
```

##### By Label/Position

```
>>> df.ix[2] #Select single row of subset of rows
Country Brazil
Capital Brasilia
Population 207847528
>>> df.ix[:, 'Capital'] #Select a single column of subset of columns
0 Brussels
1 New Delhi
2 Brasilia
>>> df.ix[1, 'Capital'] #Select rows and columns
'New Delhi'
```

##### Boolean Indexing

```
>>> s[~(s > 1)] #Series s where value is not >1
>>> s[(s < -1) | (s > 2)] #s where value is <-1 or >2
>>> df[df['Population']>1200000000] #Use filter to adjust DataFrame
```

##### Setting

```
>>> s['a'] = 6 #Set index a of Series s to 6
```

### > Retrieving Series/DataFrame Information

#### Basic Information

```
>>> df.shape #(rows,columns)
>>> df.index #Describe index
>>> df.columns #Describe DataFrame columns
>>> df.info() #Info on DataFrame
>>> df.count() #Number of non-NA values
```

#### Summary

```
>>> df.sum() #Sum of values
>>> df.cumsum() #Cumulative sum of values
>>> df.min()/df.max() #Minimum/maximum values
>>> df.idxmin()/df.idxmax() #Minimum/Maximum index value
>>> df.describe() #Summary statistics
>>> df.mean() #Mean of values
>>> df.median() #Median of values
```

### > Applying Functions

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

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

# Data Visualization with Plotly Express in Python

Learn Plotly online at [www.DataCamp.com](http://www.DataCamp.com)

## > What is plotly?

Plotly Express is a high-level data visualization package that allows you to create interactive plots with very little code. It is built on top of Plotly Graph Objects, which provides a lower-level interface for developing custom visualizations.

## > Interactive controls in Plotly



Plotly plots have interactive controls shown in the top-right of the plot. The controls allow you to do the following:

- Download plot as a png:** Save your interactive plot as a static PNG.
- Zoom:** Zoom in on a region of interest in the plot.
- Pan:** Move around in the plot.
- Box Select:** Select a rectangular region of the plot to be highlighted.
- Lasso Select:** Draw a region of the plot to be highlighted.
- Autoscale:** Zoom to a "best" scale.
- Reset axes:** Return the plot to its original state.
- Toggle Spike Lines:** Show or hide lines to the axes whenever you hover over data.
- Show closest data on hover:** Show details for the nearest data point to the cursor.
- Compare data on hover:** Show the nearest data point to the x-coordinate of the cursor.

## > Plotly Express code pattern

The code pattern for creating plots is to call the plotting function, passing a data frame as the first argument. The x argument is a string naming the column to be used on the x-axis. The y argument can either be a string or a list of strings naming column(s) to be used on the y-axis.

```
px.plotting_fn(dataframe, # Dataframe being visualized
 x=["column-for-x-axis"], # Accepts a string or a list of strings
 y=["columns-for-y-axis"], # Accepts a string or a list of strings
 title="Overall plot title", # Accepts a string
 xaxis_title="X-axis title", # Accepts a string
 yaxis_title="Y-axis title", # Accepts a string
 width=width_in_pixels, # Accepts an integer
 height=height_in_pixels) # Accepts an integer
```

## > Common plot types

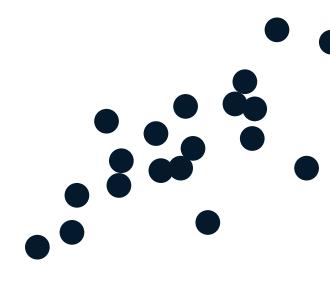
### Import plotly

```
import plotly express as px
import plotly.express as px
```

### Scatter plots

```
Create a scatterplot on a DataFrame named clinical_data
px.scatter(clinical_data, x="experiment_1", y="experiment_2")
```

Set the size argument to the name of a numeric column to control the size of the points and create a bubble plot.



### Line plots

```
Create a lineplot on a DataFrame named stock_data
px.line(stock_data, x="date", y=["FB", "AMZN"])
```

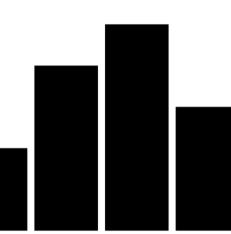
Set the line\_dash argument to the name of a categorical column to have dashes or dots for different lines.



### Bar plots

```
Create a barplot on a DataFrame named commodity_data
px.bar(commodity_data, x="nation", y=["gold", "silver", "bronze"],
 color_discrete_map={"gold": "yellow",
 "silver": "grey",
 "bronze": "brown"})
```

Swap the x and y arguments to draw horizontal bars.



### Histograms

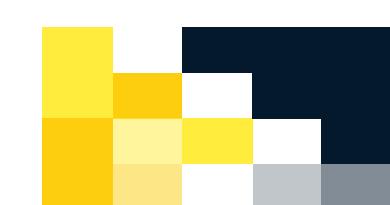
```
Create a histogram on a DataFrame named bill_data
px.histogram(bill_data, x="total_bill")
```

Set the nbins argument to control the number of bins shown in the histogram.



### Heatmaps

```
Create a heatmap on a DataFrame named iris_data
px.imshow(iris_data.corr(numeric_only=True),
 zmin=-1, zmax=1, color_continuous_scale='rdbu')
```



Set the text\_auto argument to True to display text values for each cell.

## > Customizing plots in plotly

The code pattern for customizing a plot is to save the figure object returned from the plotting function, call its .update\_traces() method, then call its .show() method to display it.

```
Create a plot with plotly (can be of any type)
fig = px.some_plotting_function()
Customize and show it with .update_traces() and .show()
fig.update_traces()
fig.show()
```

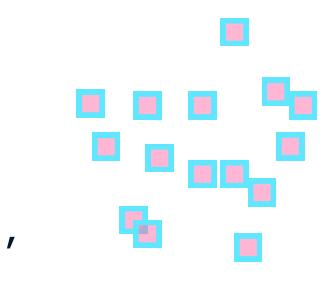
## Customizing markers in Plotly

When working with visualizations like scatter plots, lineplots, and more, you can customize markers according to certain properties. These include:

- size: set the marker size
- color: set the marker color
- opacity: set the marker transparency
- line: set the width and color of a border
- symbol: set the shape of the marker

```
In this example, we're updating a scatter plot named fig_sct
fig_sct.update_traces(marker={"size": 24,
 "color": "magenta",
 "opacity": 0.5,
 "line": {"width": 2, "color": "cyan"},
 "symbol": "square"})

fig_sct.show()
```



## Customizing lines in Plotly

When working with visualizations that contain lines, you can customize them according to certain properties. These include:

- color: set the line color
- dash: set the dash style ("solid", "dot", "dash", "longdash", "dashdot", "longdashdot")
- width: set the line width
- shape: set how values are connected ("linear", "spline", "hv", "vh", "hvvh", "vhv")

```
In this example, we're updating a scatter plot named fig_ln
fig_ln.update_traces(patch={"line": {"dash": "dot",
 "shape": "spline",
 "width": 6}})

fig_ln.show()
```



## Customizing bars in Plotly

When working with barplots and histograms, you can update the bars themselves according to the following properties:

- size: set the marker size
- color: set the marker color
- opacity: set the marker transparency
- line: set the width and color of a border
- symbol: set the shape of the marker

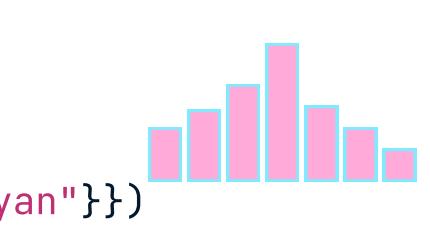
```
In this example, we're updating a scatter plot named fig_bar
fig_bar.update_traces(marker={"color": "magenta",
 "opacity": 0.5,
 "line": {"width": 2, "color": "cyan"})

fig_bar.show()
```



```
In this example, we're updating a histogram named fig_hst
fig_hst.update_traces(marker={"color": "magenta",
 "opacity": 0.5,
 "line": {"width": 2, "color": "cyan"})

fig_hst.show()
```



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

# Python For Data Science

## PySpark RDD Cheat Sheet

Learn PySpark RDD online at [www.DataCamp.com](http://www.DataCamp.com)

### Spark



PySpark is the Spark Python API that exposes the Spark programming model to Python.

### > Initializing Spark

#### SparkContext

```
>>> from pyspark import SparkContext
>>> sc = SparkContext(master = 'local[2]')
```

#### Inspect SparkContext

```
>>> sc.version #Retrieve SparkContext version
>>> sc.pythonVer #Retrieve Python version
>>> sc.master #Master URL to connect to
>>> str(sc.sparkHome) #Path where Spark is installed on worker nodes
>>> str(sc.sparkUser()) #Retrieve name of the Spark User running SparkContext
>>> sc.appName #Return application name
>>> sc.applicationId #Retrieve application ID
>>> sc.defaultParallelism #Return default level of parallelism
>>> sc.defaultMinPartitions #Default minimum number of partitions for RDDs
```

#### Configuration

```
>>> from pyspark import SparkConf, SparkContext
>>> conf = (SparkConf()
... .setMaster("local")
... .setAppName("My app")
... .set("spark.executor.memory", "1g"))
>>> sc = SparkContext(conf = conf)
```

#### Using The Shell

In the PySpark shell, a special interpreter-aware SparkContext is already created in the variable called `sc`.

```
$./bin/spark-shell --master local[2]
$./bin/pyspark --master local[4] --py-files code.py
```

Set which master the context connects to with the `--master` argument, and add Python .zip, .egg or .py files to the runtime path by passing a comma-separated list to `--py-files`.

### > Loading Data

#### Parallelized Collections

```
>>> rdd = sc.parallelize([('a',7),('a',2),('b',2)])
>>> rdd2 = sc.parallelize([('a',2),('d',1),('b',1)])
>>> rdd3 = sc.parallelize(range(100))
>>> rdd4 = sc.parallelize([('a',["x", "y", "z"]),
... ('b',["p", "r"])]))
```

#### External Data

Read either one text file from HDFS, a local file system or any Hadoop-supported file system URI with `textFile()`, or read in a directory of text files with `wholeTextFiles()`

```
>>> textFile = sc.textFile("/my/directory/*.txt")
>>> textFile2 = sc.wholeTextFiles("/my/directory/")
```

### > Retrieving RDD Information

#### Basic Information

```
>>> rdd.getNumPartitions() #List the number of partitions
>>> rdd.count() #Count RDD instances 3
>>> rdd.countByKey() #Count RDD instances by key
defaultdict(<type 'int'>,{'a':2,'b':1})
>>> rdd.countByValue() #Count RDD instances by value
defaultdict(<type 'int'>,{'b',2}:1,{'a',2}:1,{'a',7}:1)
>>> rdd.collectAsMap() #Return (key,value) pairs as a dictionary
{'a': 2, 'b': 2}
>>> rdd3.sum() #Sum of RDD elements 4950
>>> sc.parallelize([]).isEmpty() #Check whether RDD is empty
True
```

#### Summary

```
>>> rdd3.max() #Maximum value of RDD elements
99
>>> rdd3.min() #Minimum value of RDD elements
0
>>> rdd3.mean() #Mean value of RDD elements
49.5
>>> rdd3.stdev() #Standard deviation of RDD elements
28.86607004772218
>>> rdd3.variance() #Compute variance of RDD elements
833.25
>>> rdd3.histogram(3) #Compute histogram by bins
([0,33,66,99],[33,33,34])
>>> rdd3.stats() #Summary statistics (count, mean, stdev, max & min)
```

### > Applying Functions

```
#Apply a function to each RDD element
>>> rdd.map(lambda x: x+(x[1],x[0])).collect()
[('a',7,7,'a'),('a',2,2,'a'),('b',2,2,'b')]
#Apply a function to each RDD element and flatten the result
>>> rdd5 = rdd.flatMap(lambda x: x+(x[1],x[0]))
>>> rdd5.collect()
['a',7,7,'a','a',2,2,'a','b',2,2,'b']
#Apply a flatMap function to each (key,value) pair of rdd4 without changing the keys
>>> rdd4.flatMapValues(lambda x: x).collect()
[('a','x'),('a','y'),('a','z'),('b','p'),('b','r')]
```

### > Selecting Data

**Getting**

```
>>> rdd.collect() #Return a list with all RDD elements
[('a', 7), ('a', 2), ('b', 2)]
>>> rdd.take(2) #Take first 2 RDD elements
[('a', 7), ('a', 2)]
>>> rdd.first() #Take first RDD element
('a', 7)
>>> rdd.top(2) #Take top 2 RDD elements
[('b', 2), ('a', 7)]
```

**Sampling**

```
>>> rdd3.sample(False, 0.15, 81).collect() #Return sampled subset of rdd3
[3,4,27,31,40,41,42,43,60,76,79,80,86,97]
```

**Filtering**

```
>>> rdd.filter(lambda x: "a" in x).collect() #Filter the RDD
[('a',7),('a',2)]
>>> rdd5.distinct().collect() #Return distinct RDD values
[('a',2),('b',7)]
>>> rdd.keys().collect() #Return (key,value) RDD's keys
[('a', 'a', 'b')]
```

### > Iterating

```
>>> def g(x): print(x)
>>> rdd.foreach(g) #Apply a function to all RDD elements
('a', 7)
('b', 2)
('a', 2)
```

### > Reshaping Data

#### Reducing

```
>>> rdd.reduceByKey(lambda x,y : x+y).collect() #Merge the rdd values for each key
[('a',9),('b',2)]
>>> rdd.reduce(lambda a, b: a + b) #Merge the rdd values
('a',7,'a',2,'b',2)
```

#### Grouping by

```
>>> rdd3.groupBy(lambda x: x % 2) #Return RDD of grouped values
.mapValues(list)
.collect()
>>> rdd.groupByKey() #Group rdd by key
.mapValues(list)
.collect()
[('a',[7,2]),('b',[2])]
```

#### Aggregating

```
>>> seqOp = (lambda x,y: (x[0]+y,x[1]+1))
>>> combOp = (lambda x,y:(x[0]+y[0],x[1]+y[1]))
#Aggregate RDD elements of each partition and then the results
>>> rdd3.aggregate((0,0),seqOp,combOp)
(4950,100)
#Aggregate values of each RDD key
>>> rdd.aggregateByKey((0,0),seqOp,combOp).collect()
[('a',(9,2)), ('b',(2,1))]
#Aggregate the elements of each partition, and then the results
>>> rdd3.fold(0,add)
4950
#Merge the values for each key
>>> rdd.foldByKey(0, add).collect()
[('a',9),('b',2)]
#Create tuples of RDD elements by applying a function
>>> rdd3.keyBy(lambda x: x*x).collect()
```

### > Mathematical Operations

```
>>> rdd.subtract(rdd2).collect() #Return each rdd value not contained in rdd2
[('b',2),('a',7)]
#Return each (key,value) pair of rdd2 with no matching key in rdd
>>> rdd2.subtractByKey(rdd).collect()
[('d', 1)]
>>> rdd.cartesian(rdd2).collect() #Return the Cartesian product of rdd and rdd2
```

### > Sort

```
>>> rdd2.sortBy(lambda x: x[1]).collect() #Sort RDD by given function
[('d',1),('b',1),('a',2)]
>>> rdd2.sortByKey().collect() #Sort (key, value) RDD by key
[('a',2),('b',1),('d',1)]
```

### > Repartitioning

```
>>> rdd.repartition(4) #New RDD with 4 partitions
>>> rdd.coalesce(1) #Decrease the number of partitions in the RDD to 1
```

### > Saving

```
>>> rdd.saveAsTextFile("rdd.txt")
>>> rdd.saveAsHadoopFile("hdfs://namenodehost/parent/child",
... 'org.apache.hadoop.mapred.TextOutputFormat')
```

### > Stopping SparkContext

```
>>> sc.stop()
```

### > Execution

```
$./bin/spark-submit examples/src/main/python/pi.py
```

# Python For Data Science

## PySpark SQL Basics Cheat Sheet

Learn PySpark SQL online at [www.DataCamp.com](http://www.DataCamp.com)

## PySpark & Spark SQL



Spark SQL is Apache Spark's module for working with structured data.

### > Initializing SparkSession

A SparkSession can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files.

```
>>> from pyspark.sql import SparkSession
>>> spark = SparkSession \
 .builder \
 .appName("Python Spark SQL basic example") \
 .config("spark.some.config.option", "some-value") \
 .getOrCreate()
```

### > Creating DataFrames

#### From RDDs

```
>>> from pyspark.sql.types import *
```

#### Infer Schema

```
>>> sc = spark.sparkContext
>>> lines = sc.textFile("people.txt")
>>> parts = lines.map(lambda l: l.split(","))
>>> people = parts.map(lambda p: Row(name=p[0],age=int(p[1])))
>>> peopledf = spark.createDataFrame(people)
```

#### Specify Schema

```
>>> people = parts.map(lambda p: Row(name=p[0],
 age=int(p[1].strip())))
>>> schemaString = "name age"
>>> fields = [StructField(field_name, StringType(), True) for
field_name in schemaString.split()]
>>> schema = StructType(fields)
>>> spark.createDataFrame(people, schema).show()
```

|  | name     | age |
|--|----------|-----|
|  | Mine     | 28  |
|  | Filip    | 29  |
|  | Jonathan | 30  |

#### From Spark Data Sources

##### JSON

```
>>> df = spark.read.json("customer.json")
>>> df.show()
+-----+-----+
| address|age|firstName|lastName|phoneNumber|
+-----+-----+
|[New York,10021,N...| 25| John | Smith|[212 555-1234,ho...
|[New York,10021,N...| 25| Jane | Doe|[321 555-1234,ho...
+-----+-----+
```

```
>>> df2 = spark.read.load("people.json", format="json")
```

##### Parquet files

```
>>> df3 = spark.read.load("users.parquet")
```

##### TXT files

```
>>> df4 = spark.read.text("people.txt")
```

### > Filter

#Filter entries of age, only keep those records of which the values are >24

```
>>> df.filter(df["age"]>24).show()
```

## > Duplicate Values

```
>>> df = df.dropDuplicates()
```

## > Queries

```
>>> from pyspark.sql import functions as F
```

#### Select

```
>>> df.select("firstName").show() #Show all entries in firstName column
>>> df.select("firstName","lastName") \
 .show()
>>> df.select("firstName", #Show all entries in firstName, age and type
 "age",
 explode("phoneNumber") \
 .alias("contactInfo")) \
 .select("contactInfo.type",
 "firstName",
 "age") \
 .show()
>>> df.select(df["firstName"],df["age"]+ 1) #Show all entries in firstName and age,
 .show() add 1 to the entries of age
>>> df.select(df['age'] > 24).show() #Show all entries where age >24
```

#### When

```
>>> df.select("firstName", #Show firstName and 0 or 1 depending on age >30
 F.when(df.age > 30, 1) \
 .otherwise(0)) \
 .show()
>>> df[df.firstName.isin("Jane","Boris")] #Show firstName if in the given options
 .collect()
```

#### Like

```
>>> df.select("firstName", #Show firstName, and lastName is TRUE if lastName is like Smith
 df.lastName.like("Smith")) \
 .show()
```

#### Startswith - Endswith

```
>>> df.select("firstName", #Show firstName, and TRUE if lastName starts with Sm
 df.lastName \
 .startswith("Sm")) \
 .show()
>>> df.select(df.lastName.endswith("th")) #Show last names ending in th
 .show()
```

#### Substring

```
>>> df.select(df.firstName.substr(1, 3) \ #Return substrings of firstName
 .alias("name")) \
 .collect()
```

#### Between

```
>>> df.select(df.age.between(22, 24)) \ #Show age: values are TRUE if between 22 and 24
 .show()
```

## > Add, Update & Remove Columns

#### Adding Columns

```
>>> df = df.withColumn('city',df.address.city) \
 .withColumn('postalCode',df.address.postalCode) \
 .withColumn('state',df.address.state) \
 .withColumn('streetAddress',df.address.streetAddress) \
 .withColumn('telephoneNumber', explode(df.phoneNumber.number)) \
 .withColumn('phoneType', explode(df.phoneNumber.type))
```

#### Updating Columns

```
>>> df = df.withColumnRenamed('telephoneNumber', 'phoneNumber')
```

#### Removing Columns

```
>>> df = df.drop("address", "phoneNumber")
>>> df = df.drop(df.address).drop(df.phoneNumber)
```

## > Missing & Replacing Values

```
>>> df.na.fill(50).show() #Replace null values
>>> df.na.drop().show() #Return new df omitting rows with null values
>>> df.na \ #Return new df replacing one value with another
 .replace(10, 20) \
 .show()
```

## > GroupBy

```
>>> df.groupBy("age")\ #Group by age, count the members in the groups
 .count() \
 .show()
```

## > Sort

```
>>> peopledf.sort(peopledf.age.desc()).collect()
>>> df.sort("age", ascending=False).collect()
>>> df.orderBy(["age", "city"], ascending=[0,1])\
 .collect()
```

## > Repartitioning

```
>>> df.repartition(10)\ #df with 10 partitions
 .rdd \
 .getNumPartitions()
>>> df.coalesce(1).rdd.getNumPartitions() #df with 1 partition
```

## > Running Queries Programmatically

#### Registering DataFrames as Views

```
>>> peopledf.createGlobalTempView("people")
>>> df.createTempView("customer")
>>> df.createOrReplaceTempView("customer")
```

#### Query Views

```
>>> df5 = spark.sql("SELECT * FROM customer").show()
>>> peopledf2 = spark.sql("SELECT * FROM global_temp.people")\
 .show()
```

## > Inspect Data

```
>>> df.dtypes #Return df column names and data types
```

```
>>> df.show() #Display the content of df
```

```
>>> df.head() #Return first n rows
```

```
>>> df.first() #Return first row
```

```
>>> df.take(2) #Return the first n rows >> df.schema Return the schema of df
```

```
>>> df.describe().show() #Compute summary statistics >> df.columns Return the columns of df
```

```
>>> df.count() #Count the number of rows in df
```

```
>>> df.distinct().count() #Count the number of distinct rows in df
```

```
>>> df.printSchema() #Print the schema of df
```

```
>>> df.explain() #Print the (logical and physical) plans
```

## > Output

#### Data Structures

```
>>> rdd1 = df.rdd #Convert df into an RDD
```

```
>>> df.toJSON().first() #Convert df into a RDD of string
```

```
>>> df.toPandas() #Return the contents of df as Pandas DataFrame
```

#### Write & Save to Files

```
>>> df.select("firstName", "city")\
```

```
 .write \
 .save("nameAndCity.parquet")
```

```
>>> df.select("firstName", "age")\
```

```
 .write \
 .save("namesAndAges.json", format="json")
```

## > Stopping SparkSession

```
>>> spark.stop()
```

# Python For Data Science

## Scikit-Learn Cheat Sheet

Learn Scikit-Learn online at [www.DataCamp.com](http://www.DataCamp.com)

### Scikit-learn

Scikit-learn is an open source Python library that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms using a unified interface.



#### A Basic Example

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=33)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

### > Loading The Data

Also see NumPy & Pandas

Your data needs to be numeric and stored as NumPy arrays or SciPy sparse matrices. Other types that are convertible to numeric arrays, such as Pandas DataFrame, are also acceptable.

```
>>> import numpy as np
>>> X = np.random.random((10,5))
>>> y = np.array(['M', 'M', 'F', 'F', 'M', 'M', 'M', 'F', 'F', 'F'])
>>> X[X < 0.7] = 0
```

### > Training And Test Data

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X,
y,
random_state=0)
```

### > Model Fitting

**Supervised learning**

```
>>> lr.fit(X, y) #Fit the model to the data
>>> knn.fit(X_train, y_train)
>>> svc.fit(X_train, y_train)
```

**Unsupervised Learning**

```
>>> kmeans.fit(X_train) #Fit the model to the data
>>> pca_model = pca.fit_transform(X_train) #Fit to data, then transform it
```

### > Prediction

**Supervised Estimators**

```
>>> y_pred = svc.predict(np.random.random((2,5))) #Predict labels
>>> y_pred = lr.predict(X_test) #Predict labels
>>> y_pred = knn.predict_proba(X_test) #Estimate probability of a label
```

**Unsupervised Estimators**

```
>>> y_pred = kmeans.predict(X_test) #Predict labels in clustering algos
```

### > Preprocessing The Data

#### Standardization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)
```

#### Normalization

```
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer().fit(X_train)
>>> normalized_X = scaler.transform(X_train)
>>> normalized_X_test = scaler.transform(X_test)
```

#### Binarization

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X)
```

#### Encoding Categorical Features

```
>>> from sklearn.preprocessing import LabelEncoder
>>> enc = LabelEncoder()
>>> y = enc.fit_transform(y)
```

#### Imputing Missing Values

```
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values=0, strategy='mean', axis=0)
>>> imp.fit_transform(X_train)
```

#### Generating Polynomial Features

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly = PolynomialFeatures(5)
>>> poly.fit_transform(X)
```

### > Create Your Model

#### Supervised Learning Estimators

##### Linear Regression

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression(normalize=True)
```

##### Support Vector Machines (SVM)

```
>>> from sklearn.svm import SVC
>>> svc = SVC(kernel='linear')
```

##### Naive Bayes

```
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
```

##### KNN

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
```

#### Unsupervised Learning Estimators

##### Principal Component Analysis (PCA)

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=0.95)
```

##### K Means

```
>>> from sklearn.cluster import KMeans
>>> kmeans = KMeans(n_clusters=3, random_state=0)
```

### > Evaluate Your Model's Performance

#### Classification Metrics

##### Accuracy Score

```
>>> knn.score(X_test, y_test) #Estimator score method
>>> from sklearn.metrics import accuracy_score #Metric scoring functions
>>> accuracy_score(y_test, y_pred)
```

##### Classification Report

```
>>> from sklearn.metrics import classification_report #Precision, recall, f1-score and support
>>> print(classification_report(y_test, y_pred))
```

##### Confusion Matrix

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(y_test, y_pred))
```

#### Regression Metrics

##### Mean Absolute Error

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2]
>>> mean_absolute_error(y_true, y_pred)
```

##### Mean Squared Error

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(y_test, y_pred)
```

##### R<sup>2</sup> Score

```
>>> from sklearn.metrics import r2_score
>>> r2_score(y_true, y_pred)
```

#### Clustering Metrics

##### Adjusted Rand Index

```
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(y_true, y_pred)
```

##### Homogeneity

```
>>> from sklearn.metrics import homogeneity_score
>>> homogeneity_score(y_true, y_pred)
```

##### V-measure

```
>>> from sklearn.metrics import v_measure_score
>>> metrics.v_measure_score(y_true, y_pred)
```

#### Cross-Validation

```
>>> from sklearn.cross_validation import cross_val_score
>>> print(cross_val_score(knn, X_train, y_train, cv=4))
>>> print(cross_val_score(lr, X, y, cv=2))
```

### > Tune Your Model

#### Grid Search

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {"n_neighbors": np.arange(1,3),
"metric": ["euclidean", "cityblock"]}
>>> grid = GridSearchCV(estimator=knn,
param_grid=params)
>>> grid.fit(X_train, y_train)
>>> print(grid.best_score_)
>>> print(grid.best_estimator_.n_neighbors)
```

#### Randomized Parameter Optimization

```
>>> from sklearn.grid_search import RandomizedSearchCV
>>> params = {"n_neighbors": range(1,5), "weights": ["uniform", "distance"]}
>>> rsearch = RandomizedSearchCV(estimator=knn, param_distributions=params,
cv=4, n_iter=8, random_state=5)
>>> rsearch.fit(X_train, y_train)
>>> print(rsearch.best_score_)
```

# Python For Data Science

## SciPy Cheat Sheet

Learn SciPy online at [www.DataCamp.com](http://www.DataCamp.com)

### SciPy



The SciPy library is one of the core packages for scientific computing that provides mathematical algorithms and convenience functions built on the NumPy extension of Python.

### > Interacting With NumPy

Also see NumPy

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> b = np.array([(1+5j),2j,3j], [(4j,5j,6j)])
>>> c = np.array([[1.5,2,3], [4,5,6]], [(3,2,1), (4,5,6)])
```

#### Index Tricks

```
>>> np.mgrid[0:5,0:5] #Create a dense meshgrid
>>> np.ogrid[0:2,0:2] #Create an open meshgrid
>>> np.r_[[3,[0]*5,-1:1:10j]] #Stack arrays vertically (row-wise)
>>> np.c_[b,c] #Create stacked column-wise arrays
```

#### Shape Manipulation

```
>>> np.transpose(b) #Permute array dimensions
>>> b.flatten() #Flatten the array
>>> np.hstack((b,c)) #Stack arrays horizontally (column-wise)
>>> np.vstack((a,b)) #Stack arrays vertically (row-wise)
>>> np.hsplit(c,2) #Split the array horizontally at the 2nd index
>>> np.vsplit(d,2) #Split the array vertically at the 2nd index
```

#### Polynomials

```
>>> from numpy import poly1d
>>> p = poly1d([3,4,5]) #Create a polynomial object
```

#### Vectorizing Functions

```
>>> def myfunc(a):
... if a < 0:
... return a*2
... else:
... return a/2
>>> np.vectorize(myfunc) #Vectorize functions
```

#### Type Handling

```
>>> np.real(c) #Return the real part of the array elements
>>> np.imag(c) #Return the imaginary part of the array elements
>>> np.real_if_close(c,tol=1000) #Return a real array if complex parts close to 0
>>> np.cast['f'](np.pi) #Cast object to a data type
```

#### Other Useful Functions

```
>>> np.angle(b,deg=True) #Return the angle of the complex argument
>>> g = np.linspace(0,np.pi,num=5) #Create an array of evenly spaced values(number of samples)
>>> g [3:] += np.pi
>>> np.unwrap(g) #Unwrap
>>> np.logspace(0,10,3) #Create an array of evenly spaced values (log scale)
>>> np.select([c<4],[c*2]) #Return values from a list of arrays depending on conditions
>>> misc.factorial(a) #Factorial
>>> misc.comb(10,3,exact=True) #Combine N things taken at k time
>>> misc.central_diff_weights(3) #Weights for N-point central derivative
>>> misc.derivative(myfunc,1.0) #Find the n-th derivative of a function at a point
```

### > Linear Algebra

You'll use the linalg and sparse modules.

Note that scipy.linalg contains and expands on numpy.linalg.

```
>>> from scipy import linalg, sparse
```

#### Creating Matrices

```
>>> A = np.matrix(np.random.random((2,2)))
>>> B = np.asmatrix(b)
>>> C = np.mat(np.random.random((10,5)))
>>> D = np.mat([[3,4], [5,6]])
```

#### Basic Matrix Routines

##### Inverse

```
>>> A.I #Inverse
>>> linalg.inv(A) #Inverse
>>> A.T #Transpose matrix
>>> A.H #Conjugate transposition
>>> np.trace(A) #Trace
```

##### Norm

```
>>> linalg.norm(A) #Frobenius norm
>>> linalg.norm(A,1) #L1 norm (max column sum)
>>> linalg.norm(A,np.inf) #L inf norm (max row sum)
```

##### Rank

```
>>> np.linalg.matrix_rank(C) #Matrix rank
```

##### Determinant

```
>>> linalg.det(A) #Determinant
```

##### Solving linear problems

```
>>> linalg.solve(A,b) #Solver for dense matrices
>>> E = np.mat(a).T #Solver for dense matrices
>>> linalg.lstsq(D,E) #Least-squares solution to linear matrix equation
```

##### Generalized inverse

```
>>> linalg.pinv(C) #Compute the pseudo-inverse of a matrix (least-squares solver)
```

```
>>> linalg.pinv2(C) #Compute the pseudo-inverse of a matrix (SVD)
```

#### Creating Sparse Matrices

```
>>> F = np.eye(3, k=1) #Create a 2X2 identity matrix
>>> G = np.mat(np.identity(2)) #Create a 2x2 identity matrix
>>> C[C > 0.5] = 0
>>> H = sparse.csr_matrix(C) #Compressed Sparse Row matrix
>>> I = sparse.csc_matrix(D) #Compressed Sparse Column matrix
>>> J = sparse.dok_matrix(A) #Dictionary Of Keys matrix
>>> E.todense() #Sparse matrix to full matrix
>>> sparse.isspmatrix_csc(A) #Identify sparse matrix
```

#### Sparse Matrix Routines

##### Inverse

```
>>> sparse.linalg.inv(I) #Inverse
```

##### Norm

```
>>> sparse.linalg.norm(I) #Norm
```

##### Solving linear problems

```
>>> sparse.linalg.spsolve(H,I) #Solver for sparse matrices
```

#### Sparse Matrix Functions

```
>>> sparse.linalg.expm(I) #Sparse matrix exponential
```

#### Sparse Matrix Decompositions

```
>>> la, v = sparse.linalg.eigs(F,1) #Eigenvalues and eigenvectors
>>> sparse.linalg.svds(H, 2) #SVD
```

### Matrix Functions

##### Addition

```
>>> np.add(A,D) #Addition
```

##### Subtraction

```
>>> np.subtract(A,D) #Subtraction
```

##### Division

```
>>> np.divide(A,D) #Division
```

##### Multiplication

```
>>> np.multiply(D,A) #Multiplication
>>> np.dot(A,D) #Dot product
>>> np.vdot(A,D) #Vector dot product
>>> np.inner(A,D) #Inner product
>>> np.outer(A,D) #Outer product
>>> np.tensordot(A,D) #Tensor dot product
>>> np.kron(A,D) #Kronecker product
```

##### Exponential Functions

```
>>> linalg.expm(A) #Matrix exponential
>>> linalg.expm2(A) #Matrix exponential (Taylor Series)
>>> linalg.expm3(D) #Matrix exponential (eigenvalue decomposition)
```

##### Logarithm Function

```
>>> linalg.logm(A) #Matrix logarithm
```

##### Trigonometric Functions

```
>>> linalg.sinm(D) #Matrix sine
>>> linalg.cosm(D) #Matrix cosine
>>> linalg.tanm(A) #Matrix tangent
```

##### Hyperbolic Trigonometric Functions

```
>>> linalg.sinhm(D) #Hyperbolic matrix sine
>>> linalg.coshm(D) #Hyperbolic matrix cosine
>>> linalg.tanhm(A) #Hyperbolic matrix tangent
```

##### Matrix Sign Function

```
>>> np.sign(A) #Matrix sign function
```

##### Matrix Square Root

```
>>> linalg.sqrtm(A) #Matrix square root
```

##### Arbitrary Functions

```
>>> linalg.funm(A, lambda x: x*x) #Evaluate matrix function
```

### Decompositions

##### Eigenvalues and Eigenvectors

```
>>> la, v = linalg.eig(A) #Solve ordinary or generalized eigenvalue problem for square matrix
>>> l1, l2 = la #Unpack eigenvalues
>>> v[:,0] #First eigenvector
>>> v[:,1] #Second eigenvector
>>> linalg.eigvals(A) #Unpack eigenvalues
```

##### Singular Value Decomposition

```
>>> U,s,Vh = linalg.svd(B) #Singular Value Decomposition (SVD)
>>> M,N = B.shape
>>> Sig = linalg.diagsvd(s,M,N) #Construct sigma matrix in SVD
```

##### LU Decomposition

```
>>> P,L,U = linalg.lu(C) #LU Decomposition
```

### > Asking For Help

```
>>> help(scipy.linalg.diagsvd)
>>> np.info(np.matrix)
```

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

# Python For Data Science Seaborn Cheat Sheet

Learn Seaborn online at [www.DataCamp.com](http://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.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
 {“x tick major size”:8,
 “y tick major size”:8})
#Return a dict of params or use 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.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
```

### Context Functions

```
>>> sns.set_context("talk") #Set context to "talk"
>>> sns.set_context("notebook", #Set context to "notebook",
 font_scale=1.5, #Scale font elements and
 rc={“lines.linewidth”:2.5}) #override param mapping
```

### Color Palette

```
>>> sns.set_palette("husl",3) #Define the color palette
>>> sns.color_palette("husl") #Use with with to temporarily set palette
>>> flatui = [“#9b59b6”, “#3498db”, “#95a5a6”, “#e74c3c”, “#34495e”, “#2ecc71”]
>>> sns.set_palette(flatui) #Set your own color palette
```

### 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”],
 linestyles=[“-”, “--”])
```

#### 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
```

# Python For Data Science spaCy Cheat Sheet

Learn spaCy online at [www.DataCamp.com](http://www.DataCamp.com)

## spaCy

spaCy is a free, open-source library for advanced Natural Language processing (NLP) in Python. It's designed specifically for production use and helps you build applications that process and "understand" large volumes of text. Documentation: [spacy.io](https://spacy.io)

```
>>> $ pip install spacy
>>> import spacy
```

## > Statistical models

### Download statistical models

Predict part-of-speech tags, dependency labels, named entities and more. See here for available models: [spacy.io/models](https://spacy.io/models)

```
>>> $ python -m spacy download en_core_web_sm
```

### Check that your installed models are up to date

```
>>> $ python -m spacy validate
```

### Loading statistical models

```
>>> import spacy
>>> nlp = spacy.load("en_core_web_sm") # Load the installed model "en_core_web_sm"
```

## > Documents and tokens

### Processing text

Processing text with the `nlp` object returns a `Doc` object that holds all information about the tokens, their linguistic features and their relationships

```
>>> doc = nlp("This is a text")
```

### Accessing token attributes

```
>>> doc = nlp("This is a text")
>>> [token.text for token in doc] #Token texts
['This', 'is', 'a', 'text']
```

## > Label explanations

```
>>> spacy.explain("RB")
'adverb'
>>> spacy.explain("GPE")
'Countries, cities, states'
```

## > Spans

### Accessing spans

Span indices are exclusive. So `doc[2:4]` is a span starting at token 2, up to – but not including! – token 4.

```
>>> doc = nlp("This is a text")
>>> span = doc[2:4]
>>> span.text
'a text'
```

### Creating a span manually

```
>>> from spacy.tokens import Span #Import the Span object
>>> doc = nlp("I live in New York") #Create a Doc object
>>> span = Span(doc, 3, 5, label="GPE") #Span for "New York" with label GPE (geopolitical)
>>> span.text
'New York'
```

## > Linguistic features

Attributes return label IDs. For string labels, use the attributes with an underscore. For example, `token.pos_`.

### Part-of-speech tags

Predicted by Statistical model

```
>>> doc = nlp("This is a text.")
>>> [token.pos_ for token in doc] #Coarse-grained part-of-speech tags
['DET', 'VERB', 'DET', 'NOUN', 'PUNCT']
>>> [token.tag_ for token in doc] #Fine-grained part-of-speech tags
['DT', 'VBZ', 'DT', 'NN', '.']
```

### Syntactic dependencies

Predicted by Statistical model

```
>>> doc = nlp("This is a text.")
>>> [token.dep_ for token in doc] #Dependency labels
['nsubj', 'ROOT', 'det', 'attr', 'punct']
>>> [token.head.text for token in doc] #Syntactic head taken (governor)
['is', 'is', 'text', 'is', 'is']
```

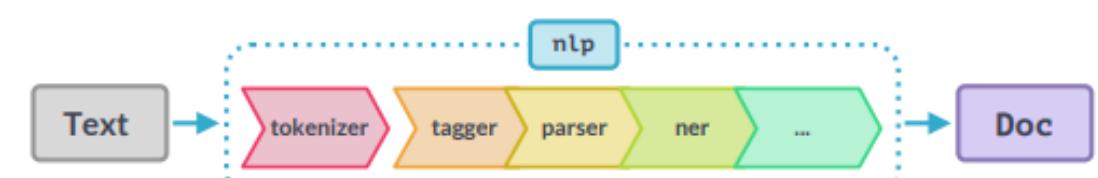
### Named entities

Predicted by Statistical model

```
>>> doc = nlp("Larry Page founded Google")
>>> [(ent.text, ent.label_) for ent in doc.ents] #Text and label of named entity span
[('Larry Page', 'PERSON'), ('Google', 'ORG')]
```

## > Pipeline components

Functions that take a `Doc` object, modify it and return it.



### Pipeline information

```
>>> nlp = spacy.load("en_core_web_sm")
>>> nlp.pipe_names
['tagger', 'parser', 'ner']
>>> nlp.pipeline
[('tagger', <spacy.pipeline.Tagger>),
 ('parser', <spacy.pipeline.DependencyParser>),
 ('ner', <spacy.pipeline.EntityRecognizer>)]
```

### Custom components

```
from spacy.language import Language
@Language.component("custom_component")
def custom_component(doc): #Function that modifies the doc and returns it
 #Do something to the doc here
 return doc
nlp.add_pipe(custom_component, first=True) #Add the component first in the pipeline
```

Components can be added first, last (default), or before or after an existing component.

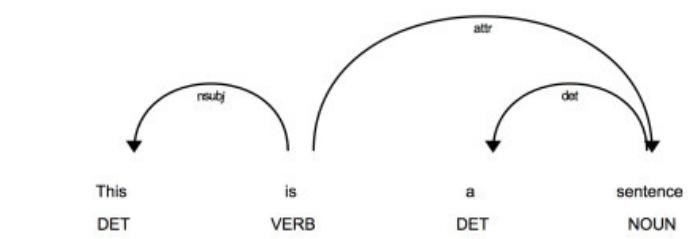
## > Visualizing

If you're in a Jupyter notebook, use `displacy.render` otherwise, use `displacy.serve` to start a web server and show the visualization in your browser.

```
>>> from spacy import displacy
```

### Visualize dependencies

```
>>> doc = nlp("This is a sentence")
>>> displacy.render(doc, style="dep")
```



### Visualize named entities

```
>>> doc = nlp("Larry Page founded Google")
>>> displacy.render(doc, style="ent")
```

Larry Page PERSON founded Google ORG

## > Word vectors and similarity

To use word vectors, you need to install the larger models ending in `md` or `lg`, for example `en_core_web_lg`.

### Comparing similarity

```
>>> doc1 = nlp("I like cats")
>>> doc2 = nlp("I like dogs")
>>> doc1.similarity(doc2) #Compare 2 documents
>>> doc1[2].similarity(doc2[2]) #Compare 2 tokens
>>> doc1[0].similarity(doc2[1:3]) #Compare tokens and spans
```

### Accessing word vectors

```
>>> doc = nlp("I like cats") #Vector as a numpy array
>>> doc[2].vector #The L2 norm of the token's vector
>>> doc[2].vector_norm
```

## > Syntax iterators

### Sentences

Usually needs the dependency parser

```
>>> doc = nlp("This a sentence. This is another one.")
>>> [sent.text for sent in doc.sents] #doc.sents is a generator that yields sentence spans
['This is a sentence.', 'This is another one.']
```

### Base noun phrases

Needs the tagger and parser

```
>>> doc = nlp("I have a red car")
#doc.noun_chunks is a generator that yields spans
>>> [chunk.text for chunk in doc.noun_chunks]
['I', 'a red car']
```

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

## > Extension attributes

Custom attributes that are registered on the global Doc, Token and Span classes and become available as `_.`.

```
>>> from spacy.tokens import Doc, Token, Span
>>> doc = nlp("The sky over New York is blue")
```

### Attribute extensions

With default value

```
Register custom attribute on Token class
>>> Token.set_extension("is_color", default=False)
Overwrite extension attribute with default value
doc[6]_.is_color = True
```

### Property extensions

With getter and setter

```
Register custom attribute on Doc class
>>> get_reversed = lambda doc: doc.text[::-1]
>>> Doc.set_extension("reversed", getter=get_reversed)
Compute value of extension attribute with getter
>>> doc_.reversed
'eulb si kroY weN revo yks ehT'
```

### Method extensions

Callable Method

```
Register custom attribute on Span class
>>> has_label = lambda span, label: span.label_ == label
>>> Span.set_extension("has_label", method=has_label)
Compute value of extension attribute with method
>>> doc[3:5].has_label("GPE")
True
```

## > Rule-based matching

### Using the matcher

```
Matcher is initialized with the shared vocab
>>> from spacy.matcher import Matcher
Each dict represents one token and its attributes
>>> matcher = Matcher(nlp.vocab)
Add with ID, optional callback and pattern(s)
>>> pattern = [{"LOWER": "new"}, {"LOWER": "york"}]
>>> matcher.add("CITIES", [pattern])
Match by calling the matcher on a Doc object
>>> doc = nlp("I live in New York")
>>> matches = matcher(doc)
Matches are (match_id, start, end) tuples
>>> for match_id, start, end in matches:
 # Get the matched span by slicing the Doc
 span = doc[start:end]
 print(span.text)
'New York'
```

### Token patterns

```
"love cats", "loving cats", "loved cats"
>>> pattern1 = [{"LEMMA": "love"}, {"LOWER": "cats"}]
"10 people", "twenty people"
>>> pattern2 = [{"LIKE_NUM": True}, {"TEXT": "people"}]
"book", "a cat", "the sea" (noun + optional article)
>>> pattern3 = [{"POS": "DET", "OP": "?"}, {"POS": "NOUN"}]
```

### Operators and quantifiers

Can be added to a token dict as the "OP" key

- ! Negate pattern and match exactly 0 times
- ? Make pattern optional and match 0 or 1 times
- + Require pattern to match 1 or more times
- \* Allow pattern to match 0 or more times

## > Glossary

### Tokenization

Segmenting text into words, punctuation etc

### Lemmatization

Assigning the base forms of words, for example:  
"was" → "be" or "rats" → "rat".

### Sentence Boundary Detection

Finding and segmenting individual sentences.

### Part-of-speech (POS) Tagging

Assigning word types to tokens like verb or noun.

### Dependency Parsing

Assigning syntactic dependency labels,  
describing the relations between individual  
tokens, like subject or object.

### Named Entity Recognition (NER)

Labeling named "real-world" objects,  
like persons, companies or locations.

### Text Classification

Assigning categories or labels to a whole  
document, or parts of a document.

### Statistical model

Process for making predictions based on examples.

### Training

Updating a statistical model with new examples.

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

