Python Reference

by : Dwight

for machine learning

basic python syntax

pandas (dataframes)

matplotlib & plotly (graphs)

scikit-learn (machine learning models)

note: "UCI machine learning repo" has some great datasets to work on and test models

Python Basics

Legend
- notes
- code
- explanation

variable assignment

In python, there is no need to assign a type to a variable. For instance,
code:

```
spam amount = 5
```
(spam amount: variable; 5: value)

Comments

There are two different types of comments:

i) in-line:

```
# this is an inline comment
```

ii) code block comment / docstring

```
""" this is a code block comment """
```

Type

There are 4 basic python types:
int (whole number), float (decimals), string (words), boolean (true / false)

to check the type of a variable,
code:

```
type( word_count )
```

if-statements:

Note, python uses blank space to indicate what statements belong to what block of code.
↳ 4 spaces indicates a block of code

syntax:

```
if <expression> :
    statement
elif <expression> :
    statement
else :
    statement
```
(4 spaces)

numbers & numerical operations

There are two different types of numbers in python: integers and floats. Here are some operations you can do on numbers:

Name	Operator	Description
addition	a+b	sum a & b
subtraction	a-b	difference of a & b
multiply	a * b	product of a & b
division	a/b	quotient of a & b
floor division	a // b	quotient of a & b, removing fractions

modulus	a % b	integer remainder after division
exponent	a**b	a raised to the power of b

order of operations:
all math expressions are evaluated as follows,

Parentheses
Exponent
Multiplication
Division
Addition
Subtraction

Note: floats have interesting methods associated with them. For instance:

```
x = 0.125
x.as_integer_ratio()      ... output: (1,8)
numerator, denominator = x.as_integer_ratio()
```

double variable assignment: numerator = 1 and denominator = 8

aggregations:

- absolute value
  `abs(-32)` ... output: 32
- minimum
  `min(1,2,3)` ... output: 1
- maximum
  `max(1,2,3)` ... output: 3

casting values:
we can force python to change the type of a variable or python

for example,

```
print( float(10) )
output: 10.0

print( int 3) )
output: 3
```

functions

the most basic fcn we can use is the helper function
↳ helps you understand how a python fcn works

code:

```
help(round)
```

name of the fcn and not the function call. For instance, round() is a fcn call

defining a fcn syntax:

```
def <fcn_name> (arg):
    <code block>
    return expression
```

we can also put default values in our fcn so that if user doesn't enter anything it does a calculation based on default

It is best practice to put a docstring in a fcn

fcns with a docstring:

```
def <fcn_name> (arg):
    """ what the fcn does

    >>> fcn_name()
    output
    """
    <code block>
    return <expression>
```

This is a docstring what we see when we call the help fcn

boolean

Booleans are logical statement evaluations; either true or false

Operators:

Name	logical equal	less than	less than equal to	not equal to	greater than	greater than equal to
Operator	a == b	a < b	a <= b	a != b	a > b	a >= b

other operators:
and, or, not

truth tables:

And

Arg1	Arg2	Result
True	True	True
True	False	False
False	True	False
False	False	False

Or

Arg1	Arg2	Result
True	True	True
True	False	True
False	True	True
False	False	False

boolean conversion:
bool(1) . . . all numbers are true, except zero
bool("asf") . . . all string true except empty string

lists

an ordered sequence of values

code
```
prime = [2, 3, 5, 7]
```

indexing:
Note, in python the list starts at index zero

code
```
prime[0]    . . . output: 2    ; first element of list
prime[-1]   . . . output: 7    ; last element of list
```

slicing:
using select parts of a list
code
```
prime[0:3]    . . . first element until the 3rd element
prime[ :3]    . . . leave out the end index but, give me every element until that point
prime[3: ]    . . . all elements from the 4th element
prime[-3:]    . . . last 3 elements of list
```

changing elements in a list:
```
prime[0] = "Alex"
prime
```

Output:
[Alex, 3, 5, 7]

list functions:
len(prime) . . . length of a list
sum(prime) . . . sum the list

list operators

operator	example
in	5 in prime

list methods:
a method is a fun attached to an object

```
prime.append(9)          . . . add element to end of list
prime.pop(9)             . . . removes last element of list
prime.index("Earth")     . . . find the index of an element
prime.count(5)           . . . count number of occurences of 5
<list>.extend(<list>)    . . . add a list to the end of a list
```

Tuples

idea: an immutable/unchangable list

example:

```
t = (1, 2, 3)
t = 1, 2, 3
```

] different ways of doing the same thing

List Comprehensions

A way of doing operations on a list as you are creating it

```
square = [n**2 for n in range(10)]
```

This executes last saying to square the list. This executes first saying: make a list from 0–9. . . output: [0, 1, 4, 9, 16, 25, ... 81]

```
short_planet = [planet for planet in planets if len(planet) < 6]
```

① a list of planets whose names are shorter than 6 characters

② what happens when the condition is met

③

Loops

for loop syntax:

```
for <element> in <list>:
    <code block>
```

range loop syntax: (a range up to but not including this number)

```
for i in range(5):
    <code block>
```

while loop:

```
while <boolean condition>:
    <code block>
```

sets:

idea: a list where order doesn't matter.

downfall: can't rely on the order of info when it is retrieved

```
my_set = {1, 2, 3}
```

→ can only hold immutable data types

Dictionaries

A list of key-value pairs

- we can also do comprehensions similar to list comprehensions, called dictionary comprehensions

```
numbers = {"one": 1, "two": 2}
```

indexing a dictionary:

```
numbers['one']
```

. . . output: 1

adding to a dictionary: (a key not in the dictionary)

```
numbers["eleven"] = 11
```

changing values in a dictionary:

```
numbers['one'] = 'sports'
```

Strings

A way of passing words into python

Similar to lists we can slice strings (look at list section for examples).

String methods:

- `<string>.capitalize()` . . . capitalize the first letter of a string
- `<string>.upper()` . . . capitalize the whole string
- `<string>.lower()` . . . make the whole string lowercase
- `<string>.index()` . . . search for the first index of a substring
- `<string>.split()` . . . turn a string into a list of words
- `<string>.count(arg)` . . . how many times the string <arg> occurs in the larger string
- `<string>.isdigit()` . . . determine if string contains only digits
- `<string>.isalpha()` . . . determine if string contains only words

```
"{} is unhappy on {}".format(name, weekday)
```

① ② predifined variables: insert these values in the string where there are curly brackets

Importing libraries

libraries add more functionality to base python

```
import math
print(dir(math))
```

. . . math is a module (aka a collection of variables)

. . . dir() allows use to see all the variables in a module

we can access variables using dot syntax:

```
pi = math.pi
```

modules also have fcns in them:

```
math.log(32,2)
help(math.log)
```

. . . tells us how to use the log fcn in math module

alias import statements:

```
import math as mt
```

. . . instead of using math we can use mt

`mt.pi` . . . same as calling math.pi bc mt is an alias for math

import all:

```
from math import *
from math import log, pi
from numpy import numpy.random
```

- `from math import *` . . . make all variables from the math module available
- `from math import log, pi` . . . we can also import only specific variables
- `from numpy import numpy.random` . . . we can also import sub libraries held in variables

Pandas

installing pandas

```
pip install pandas
```

or

```
conda install pandas
```

importing pandas

```
import pandas as pd
```

get a summary of data

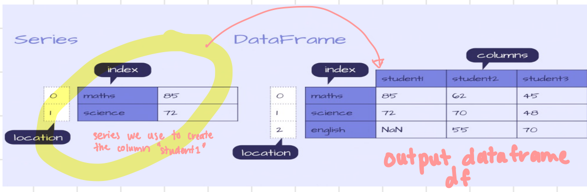
```
df.describe()
```

tells us about the mean, std deviation, min/max, IQR and # of non-missing values for every column of a dataframe

Pandas hosts 2 data structures:

i) series - a list / column of data

ii) dataframes - a table similar to that in spreadsheets



3 methods to create a dataframe:

i) empty dataframe:

```
df = pd.Dataframe()
```

ii) dataframe of series:

```
d = {'student1': pd.Series([85., 72.], index = ['maths', 'science'])
     'student2': pd.Series([62., 70., 55.], index = ['maths', 'science', 'english'])
     'student3': pd.Series([45., 48., 70.], index = ['maths', 'science', 'english'])}
```

(column values; row names [in-order from top-to-bottom]; column name)

we can also use a dictionary of lists

```
df = pd.Dataframe(d)
print(df.head())
```

- `df = pd.Dataframe(d)` . . . creates a dataframe from the dictionary of series above
- `print(df.head())` . . . show the first 5 entries of the table

iii) create a dataframe from a file

2 main ways to get a dataframe from a file:

↳ ① csv files:

```
df = pd.read_csv(csv_file_name)
```

(csv_file_name → path to csv file or web address)

if we put the argument index_col = 0 after the filename we can index columns using numbers

② querying a database

```
import pymysql
con = pymysql.connect(host = 'localhost', user = 'test', password = '', db = 'penguinDB')
df = read_sql(f """select * from penguins""", con)
```

(connect → database connection info; """ → docstring quotes! so so we can use quotes in our query; select * from penguins → query; con → connect to database)

③ read general text files:

```
df = pd.read_table(filename, sep = "\t")
```

(sep → what separates each column; \t → tab symbol)

export dataframe to csv file:

```
df.to_csv('output.csv', compression = 'gzip')
```

('output.csv' → filename we want to create)

dimensions of a dataframe

```
len(df)              ... number of rows
len(df.columns)      ... number of columns
df.size              ... number of elements in the table (rows x columns)
df.shape             ... a tuple of format (rows, columns)
df.info()            ... info on column types (str, int, bool, etc.) and number of non-null entries
```

we can also set how many items we see when we call the head fcn

```
pd.set_option('display.max_columns', None)      ... permanently set # of items we see
```

option to set — how many columns to max out at

pandas types:

dtype	Description
object	str or mixed datatypes
int64	Default for integers
float64	Default for floats
bool	True/False used for missing values
datetime64	Datetime values
category	Limited, usually fixed number of possible values

accessing data

get the names of columns in the dataframe

```
df.columns
```

view a column

same thing:

i) `df.column_name`

ii) `df['column_name']`

column name as a string (can also be a list of columns we want to see. For example, `df['column1','column2']`)

get one value in a column

```
df['column_name'][index]
```

index: integer

removing individual/a group of columns

```
df.drop(columns = [column_names])
```

indexing in python (2 types):

↳ index-based selection (selection using #'s)

```
df.iloc[row_index]              ... return this row in df
df.iloc[:, column_index]        ... return this column
df.iloc[:3, column_index]       ... return a column up to this row but not including it
df.iloc[[0,1,2], column_index]  ... return only the rows in the list
```

index range: first element of range is included. Last element of range excluded

row index

↳ location-based selection (select using column names)

```
df.loc[rows, cols]
```

example:

```
keep_cols = ["student1", "student3"]
df = df.loc[:, keep_cols]
df.loc['Apples':'Potatoes']     ... return all the columns alphabetically between 'Apples' & 'Potatoes'
```

index range: first element of range included. Last element of range also included

index of rows

column names we want/ index of columns

conditional selection of rows

we also we can also use the fcn is null to get a table where the values in a column are null. For example:
`df.loc[df.column_name.isnull()]`

Only view rows & columns that satisfy some condition

```
df.country == 'Italy'        ... returns a column of booleans where True means that row satisfies the condition
df.loc[df.country == 'Italy']  ... return a dataframe where all rows satisfy this condition
df.loc[<condition 1> & <condition 2>]   ... dataframe satisfying 2 conditions
df.loc[<condition 1> | <condition2>]    ... dataframe satisfying at least one of two conditions
df.loc[df.column_name.isin(<list>)]     ... only see rows where the value of the given column is in the given list
```

missing values

— in pandas NaN = null

determine # of missing values in each column:

```
df.isna().sum()
```

or

```
df.isnull().sum()
```

remove missing values

```
df.dropna(inplace = True)
```

replace missing values

what to put where there are missing values

```
df['sex'].fillna('unknown', inplace = True)
```

column name

```
df.column_name.fillna('unknown')
```

replacing incorrect values:

```
df.column_name.replace(old_value, new_value)
```
(old_value can be of any type)

when to remove: you have a large dataset with very few (1%) missing values.

when to replace: in cases where we have many missing values. We replace missing values with the column mean

data integrity: what values are in each column?

```
for col in ['Region', 'Island', 'sex']:
    print(f'column: {col} has {df[col].nunique()} unique values: {df[col].unique()}')
```
(# of unique values in the column; unique values in the column)

alternatively, we can call the value_counts method to see unique values in a column, as well as, the number of times each value appears in that column

```
df.column_name.value_counts()
```

making a column categorical

```
df['Species'] = df['Species'].astype('category')
```
('Species': column name; 'category': pandas data type)

advantages of make a column categorical? We save significant space in memory

maps

idea: take one set of values and map them to another set of values
↳ basically a transformation

2 methods to do a map:

i) using the map fcn:

```
df.column_name.map(lambda x: x - 3)
```
(x: input; x - 3: transformation; map fcn; return a new transformed Series/column)

. . . for every row in this column apply this transformation

ii) using the apply fcn:

```
df.apply(function, axis = 'columns')
df.apply(function, axis = 'index')
```
. . . apply a fcn to each row

. . . apply a fcn to each column

(return a new transformed dataframe)

groupwise analysis

Allows us to group rows by common values in a column

```
df.groupby('column_name')
```
. . . returns a dataframe we can do aggregations on

```
df.groupby('points').points.count()
```
. . . count number of unique values in the points column

```
df.groupby('points').price.min()
```
('points': column from dataframe)

. . . return the lowest price for every point rated

```
df.groupby('winery').apply(lambda table: table.title.iloc[0])
```
(table.title: column name, in this case title is the column containing wine names)

. . . the first wine every reviewed by every winery

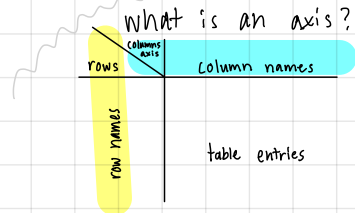
Note, we can also groupby multiple columns; just pass a list in the argument of the groupby fcn

do multiple aggregations on a column at a time

```
df.groupby(['country']).price.agg([len, min, max])
```

. . . returns a dataframe of aggregations given in the list

(fcns we want to apply to the price column)



sort columns

```
df.sort(column= column_name, ascending = True)
```

(sort by increasing value)

(can also use sort_values(column_name, ascending=False))

renaming columns/row indices/axis names

```
df.rename(columns = {'old_column_name': 'new_column_name'})
df.rename(index = {old_index1 : new_index1})
df.rename_axis('wines', axis = 'rows').rename_axis('fields', axis = 'columns')
```

. . . rename a column

. . . rename row indices

('wines': change the name of row axis to this; 'fields': change the name of the column axis to this)

combining dataframes

Concatenation – when we have two tables with the same column names but different data

```
canadian_views = pd.read_csv(...)
british_views = pd.read_csv(...)
pd.concat([canadian_views, british_views])
```

] same column names in each table

join – dataframe must have the same number of rows

```
left = canadian_views.set_index(['title', 'trending_date'])
right = british_views.set_index(['title', 'trending_date'])

left.join(right, lsuffix = '_CAN', rsuffix = '_UK')
```

(make these column indices for our data)

(left: table names; first columns we will see come from here)

(right: column names' second set of column names come from here)

(lsuffix: add this string to the end of all column names from the left table)

(rsuffix: add this string to all column names from the right table)

Tidying a dataframe

→ melt = pivot longer
pivot = pivot wider
wide_to_long = seperate

https://pandas.pydata.org/docs/user_guide/reshaping.html

These are functions used to tidy our data. What does it mean to tidy a dataframe? Each observation has its own row, each variable has its own column and each type of observation is in one table.

pivot longer (aka melt):

the pivot_longer fcn from R is called melt in python

for example:

pigs dataframe

pig \ feed	feed1	feed2	feed3	feed4
1	60.8	68.7	92.6	87.9
2	57.0	67.7	92.1	84.2
3	65.0	74.0	90.2	83.1
4	58.6	66.3	96.2	85.7
5	61.7	64.8	99.1	90.3

table of pig weights by feed

```
pigs = pd.read_csv(...)
pigs['id'] = df.index
pigs.melt(id_vars =['id'], var_name = 'feed', value_name='weight')
```

. . . add a new column to the dataframe of row indices

(id_vars: name of column(s) we don't want to put in new column; a list)

(var_name: what we will call the new column that has the old column names)

(value_name: what to call the new column that has the values of the old column)

alternatively,

```
pigs.melt(id_vars=['id'], value_vars = ['feed1', 'feed2', 'feed3', 'feed4'],
          var_name = 'feed', value_name = 'weight')
```

(store in a variable if you want to keep this)

(value_vars: list of columns we want to combine into one column)

output:

id	feed	weight
0	feed 1	60.8
1	feed1	57.0
2	feed1	65.0
3	feed1	58.6
4	feed1	61.7
0	feed2	68.7
⋮	⋮	⋮

QED □

pivot longer & seperate

Alternatively, it could be the case that a wide table has many variables that represent two items. For example:

tb dataframe

country code iso2	year	males between ages 0 - 4 m04	males between ages 5 & 14 m514	females between ages 0 and 4 f04
AD	1996	0	0	1
CA	1996	3	5	8
US	1994	1	10	2

untidy data: should have different column for age & gender

```
tb = read_csv( ... )
tb2 = pd.wide_to_long(tb, ['m', 'f'], i='iso2', j='age',
      sep = 1 )
```

- this is also where values from these columns of the old table will be stored
- column to use as id variable
- name of new column(s) being created that will contain the part of the column title we cut off
- the characters the columns we want to split up starts with
- index of the column name we want to seperate from; can also be a character like "_"

output:

iso2	year	m	f	age
AD	1996	0	1	04
AD	1996	0	null	514
CA	1996	3	8	04
CA	1996	5	null	514
US	1994	1	2	04
US	1994	10	null	514

Note, this is not yet tidy but we are close

tb_clean dataframe

iso2	year	gender	age	frequency
AD	1996	m	04	0
		f	04	1
		m	514	0
		f	514	null
CA		m	04	3
		f	04	8
		m	514	5
		f	514	null
US	1994	m	04	1
⋮	⋮	⋮	⋮	⋮

```
tb_clean = tb2.melt(id_vars=['iso2', 'year', 'age'],
           value_vars=['m', 'f'], var_name = 'gender',
           value_name = 'frequency')
```

- what remains constant (in terms of columns)
- what columns we want to combine
- name of new column that contains old column names
- new column name where values will be stored

QED □

pivot wider

Similar to seperate, used when one column holds multiple variables.

table "df"

	countries	metrics	values
0	A	population_in_million	100
1	B	population_in_million	200
2	C	population_in_million	120
3	A	gdp_percapita	2000
4	B	gdp_percapita	7000
5	C	gdp_percapita	15000

one column holds 2 variables

```
df2 = df.pivot(index = "countries")
```

table "df2"

each variable has its own column

countries	metrics gdp_percapita	population_in_million
A	2000	100
B	7000	200
C	15000	120

common statistical fcns

- `count()` - # of non-null observations
- `sum()` - sum of values
- `mean()` - mean
- `median()` - median
- `std()` - standard deviation
- `var()` - variance
- `skew()` - skew (3rd moment)
- `kurt()` - kurtosis (4th moment)
- `quantile()` - sample quantile [value at %]
- `cov()` - covariance
- `corr()` - correlation

table styling

we can style a table by looking at individual values and applying sum style if a specific criteria is met:

```
def turn_number_red(val):
    """ takes a scalar and returns a
    a red string if scalar is negative;
    and black otherwise """
    colour = 'red' if val < 0 else 'black'
    return 'colour: %s' % colour
```

```
def highlight_max(column):
    """ highlight the largest number
    in a column """
    is_max = (column == column.max())
    return ['background-colour: yellow' if
            v else '' for v is_max]
```

```
df.style.apply(turn_number_red)
df.style.apply(highlight_max)
```

We can also format numbers to a specific decimal place

```
df.style.format("{:.2%}")
```

(`{:.2%}`: format # to 2 decimal places; add a percentage sign)

We can even add a caption to our table:

```
df.style.caption('the words we want to caption our table')
```

hide row indices:

```
df.style.hide_index()
```

Matplotlib (graphs and visualizations)

A visualization library used in python

The type of graph we plot depends on the number and type of variables.
What chart to plot and why:

# of categorical variables	# of quantitative variables	graph
1	0	bar graph
0	1	histogram
2	0	grouped bar graph
1	1	side-by-side boxplots
0	2	scatterplot
2	1	grouped boxplot
1	2	scatterplot with points identified by group

To use matplotlib for making graphs we need to import the pyplot module and numpy

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

anatomy of a plot

components of a plot:

i) figure - the overall window or page everything is drawn on. Think of it like a container we can add things to (i.e. subtitle, legend, colour bar, etc.).

ii) axes - a componet of we can add to a figure. It's the area where the data is plotted. We can add components to it such as ticks or labels

iii) x-axis / y-axis - contains major ticklines and minor ticklines, axis labels, axis scales and axis gridlines.

iv) spines - lines that connect the tick marks on the x and y-axis. These are what we would traditionally think of visually when we see the x-axis or the y-axis

v) artist objects - anything drawn by matplotlib is part of the Artist module

To have our plots embedded inside our jupyter notebook we use the command:

```
%matplotlib inline
```

what is a subplot?

Used to setup and place our axes on a regular grid. In most cases, axes and subplot are the same thing. But note, there are differences between add_subplot() and add_axes()

example of a basic plot:

```
# create a figure
fig = plt.figure()
```

```
# setup our axes
ax = fig.add_subplot(1 1 1)        # ... an axes object
# scatter the data
ax.scatter(np.linespace(0,1,5), np.linespace(0,5,5))

#show the plot
plt.show()
```

represents 3 arguments: # of rows, # of columns, plot number (in that order)

can also be written 1,1,1

thus, a subplot of (2,2,1) is a 2 x 2 grid where each grid can hold a graph. The last # in the list indicates which graph you're working on, in our case, 1 means we are working on graph #1

change the size of our plot

we pass the argument figsize into the figure method to change the size of a plot

```
#initialize the plot
fig = plt.figure(figsize=(20, 10))

ax1 = fig.add_subplot(121)     # ... plotting area for graph 1
ax2 = fig.add_subplot(122)     # ... plotting area for graph 2
```

make an area to plot 2 graphs in one row

width of graph in inches

height of graph in inches

first graph

basic customization

```
ax1.axhline(value_on_yaxis)    # ... horizontal line at y = value_on_yaxis
ax2.axvline(value_on_xaxis)    # ... vertical line at x = value_on_xaxis
fig.delaxes(axes_name)         # ... delete an axes
fig.add_axes(axes_name)        # ... add a deleted axes

ax.legend(loc='center')        # ... edit the location of the legend
ax.set(title='plot title', xlabel='x', ylabel='y')   # ... set title, ylabel and xlabel for the graph
figure.suptitle('figure title')   # ... add a title to the figure
plt.tight_layout()             # ... makes plot fit nicely in figure. Call right before you call the plt.show() fun
plt.subplot_adjust()           # ... manually set the width and height of blank space between subplots
plt.style.use("ggplot")        # ... style the plots similar to ggplot
print(plt.style.available)     # ... look at a list of other style options
```

specific axes we are working on

we can also use the argument bbox_to_anchor

can also take the values: left and right

saving a plot

save an image of our plot:

```
plot.savefig("file_name", transparent=False)
```

example: image.png

do we want a transparent image

optional argument

save a pdf of our plot:

```
from matplotlib.backend_pdf import PdfPages
pp = PdfPages("filename.pdf")   # ... create a file
pp.savefig()                    # ... save the figure to the file
pp.close()                      # ... close the file
```

clear the plotting area

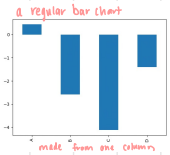
```
plt.cla()      # ... clear an axis
plt.clf()      # ... clear a figure
plt.close()    # ... close a window that popped up to show a plot
```

bar graphs (for one column)

using matplotlib with pandas

```
plt.figure();
df.iloc[5].value_counts().plot.bar()
```

a single column — Count the frequency of all the values in the column — a method of the dataframe



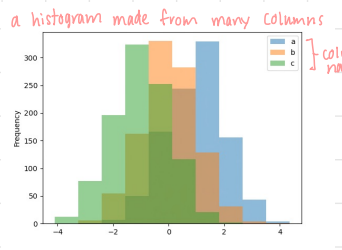
histogram (for one column)

using matplotlib with pandas

```
plt.figure();
df.plot.hist(alpha = 0.5)
```

create a df containing only one column to see a regular histogram instead of a grouped-histogram

other arguments: bins (# of bins), orientation (horizontal/vertical), cumulative (true/false) — do we want a cumulative histogram



boxplots

using matplotlib with pandas

```
df.boxplot(column = 'column_name', by = 'column2_name')
```

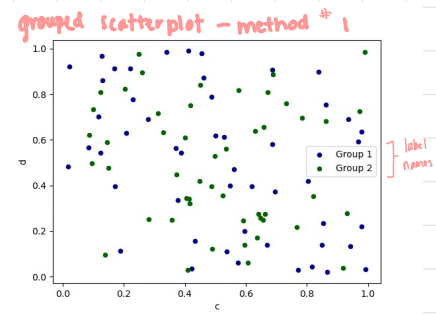
'column_name' — y-variable; 'column2_name' — x-variable

scatterplot

a regular scatterplot

```
df.plot.scatter(x = "x_variable", y = "y_variable")
```

"x_variable" — name of column we want to put on x-axis; "y_variable" — name of column we want to put on the y-axis



grouped scatterplot

```
ax = df.plot.scatter(x = "column1", y = "column2", color = "DarkBlue", label = "Group 1")
df.plot.scatter(x = "column3", y = "column4", color = "DarkGreen", label = "Group 2", ax = ax)
```

"Group 1" — name of group; ax = ax — put points on the same graph as data in group 1

alternatively, we can filter and plot data from the same column

```
ax = df[df.Sport == "BBall"].plot.scatter(x = "BMI", y = "Wt", color = "DarkBlue", label = "Basketball")
df[df.Sport == Swim].plot.scatter(x = "BMI", y = "Wt", color = "Green", label = "Swim", ax = ax)
```

df.Sport == Swim — filter rows: show only rows that meet this condition

"BMI" — column name; "Wt" — column name

Plotly (graphing library)

setup a plotly account:
https://plot.ly/settings/api#/

An alternative to Matplotlib

to install the library:
```
!pip install plotly == 4.8.2
```

importing the library
```
import plotly.plotly as py
import plotly.graph_objects as go
plotly.tools.set_credentials_file(username='Your Accont', api_key='Your API key')
```
(py: free to communicate w/ plotly servers; username: account username; api_key: account password; you need to setup a free account to be able to save your plots)

when displaying visualizations in plotly, both the data and the graph are saved to your plotly account (for upto 25 graphs).

displaying plots:
```
py.iplot()   ... Jupyter Notebooks: display plots in the cell below
py.plot()    ... save plot to a url page
```

save all plots offline:
```
plotly.offline.iplot()   ... save plot to current Jupyter Notebook
plotly.offline.plot()    ... save url page graphs locally
```

general graph structures

graph objects have several components:

- trace – contains data and specifications for how data should be plotted. Basically a dictionary of parameters for how data should be plotted

  example:
```
trace1 = {"x": ["2017-09-30", "2017-10-31", ...],
          "y": [327900, 329100, 331030, ...],
          "line": {"color": "#385965", width = 1.5},
          "mode": "lines"
          "name": "Hawaii"
          "type": "scatter",
         }
```
we can even put several traces in a list to plot as data

- layout – handles how data in a graph looks, as well as, how axis and titles are displayed. Also a dictionary of parameters
```
layout = {"showlegend": True,
          "title": {"text": "Zillow Home Value"},
          "xaxis": {"rangslider": {"visible": True}, "title": {"text": "Year (1996-2017)"},
                    "zeroline": False},
          "yaxis": "title": {"text": "Home Prices"}, "zeroline": False}
         }
```

What graph to make and when:

# of categorical variables	# of quantitative variables	graph
1	0	bar graph
0	1	histogram
2	0	grouped bar graph
1	1	side-by-side boxplots
0	2	scatterplot
2	1	grouped boxplot
1	2	scatterplot with points identified by group

bar graph (for categorical variables)

a simple bar graph:

```
frequency = df.loc['column'].value_counts().sort_values(by="column", ascending=True)
trace = go.Bar(x=df.column, y=frequency)
fig = go.Figure(data=trace)
py.iplot(fig)
```

grouped bar graph:

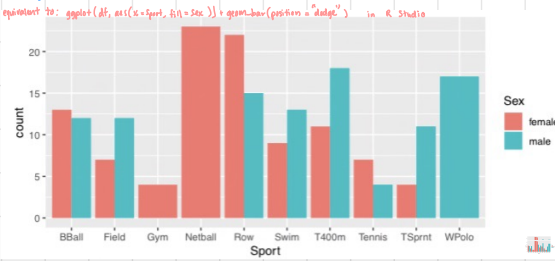
```
male_sport_freq = []
for sport in df.Sport.unique():
    count = df[(df.Sport == sport) & (df.Sex == 'male')].count()
    male_sport_freq.append(count)

female_sport_freq = []
for sport in df.Sport.unique():
    count = df[(df.Sport == sport) & (df.Sex == 'female')].count()
    female_sport_freq.append(count)

trace1 = go.Bar(x=df.Sport.unique, y=male_sport_freq, name='male')   ... a plot
trace2 = go.Bar(x=df.Sport.unique, y=female_sport_freq, name='female')   ... a plot

Sport_by_Sex = [trace1, trace2]   ... combine 2 groups into one plot

layout = go.Layout(barmode="group", title="How many male/female are in each Sport?")
fig = go.Figure(data=Sport_by_Sex, layout=layout)
py.iplot(fig)
```



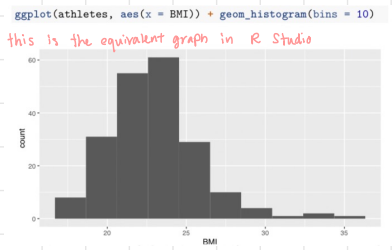
we can add other arguments to go.Bar() besides x, y and name to customize our plots even more.

visit: plotly.com/python/bar-charts/

histogram (for continuous data or look at a distn)

basic histogram

```
fig = Figure(data=[go.Histogram(x=df.BMI)])
py.iplot(fig)
```



normalized / probability histogram

```
fig = go.Figure(data=[go.Histogram(x=df.BMI, histnorm='probability')])
py.iplot(fig)
```

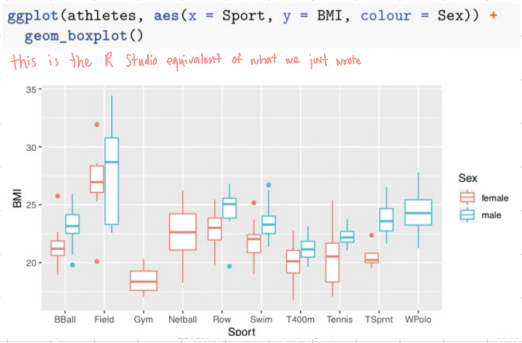
multiple histograms on one plot

```
x0 = np.random.randn(500)   ... generate a sample size of 500
x1 = np.random.randn(500)+1   ... generate one sample of size 500, then shift the mean by 1

fig = go.Figure()   ... an empty figure
fig.add_trace(go.Histogram(x=x0))
fig.add_trace(go.Histogram(x=x1))
```

```
fig.update_layout(barmode='overlay')   ... put one histogram on the other
fig.update_traces(opacity=0.75)
py.iplot(fig)
```

boxplot



basic boxplot

```
fig = go.Figure()
fig.add_trace(go.Box(x=df.Sex, y=df.BMI))
py.iplot(fig)
```

grouped boxplot

```
fig = go.Figure()
fig.add_trace(go.Box(x=df[df.Sex=="male"].Sport, y=df[df.Sex=="male"].BMI, name='male',
              marker_color='blue'))   ... plot each value of our subgroup independently
fig.add_trace(go.Box(x=df[df.Sex=="female"].Sport, y=df[df.Sex=="female"].BMI, name="female",
              marker_color='pink'))
fig.update_layout(yaxis_title='BMI values', boxmode='group')
py.iplot(fig)
```

(filter individual values of the columns we want our subgroups to be from)

scatterplot

basic scatterplot

```
fig = go.Figure(data=go.Scatter(x=df.Ht, y=df.Wt, mode="markers"))
py.iplot(fig)
```

grouped scatterplot

```
fig = go.Figure()
fig.add_trace(go.Box(x=df[df.Sex=="male"].Ht, y=df[df.Sex=="male"].Wt, name='male',
              mode="markers", marker_color='blue'))
fig.add_trace(go.Box(x=df[df.Sex=="female"].Ht, y=df[df.Sex=="female"].Wt, name="female",
              mode="markers", marker_color='pink'))   ... plot each value of our subgroup independently
py.iplot(fig)
```

(filter individual values of the columns we want our subgroups to be from)

time series

basic time series

```
df = pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/finance-charts-apple.csv")
fig = go.Figure(data=go.Scatter(x=df.Date, y=df.AAPL_High))
fig.update_layout(xaxis_range=["2016-07-01","2016-12-31"])   ... manually set the range of dates shown
fig.update_xaxis(rangeslider_visible=True)   ... have a slide to adjust the range of dates shown
py.iplot(fig)
```

making models with scikitlearn

the goal of statistics: to understand the relationships that exist between variables. This is done through p-values and levels of significance.

the goal of machine learning: to get accurate predictions of data using the info we have at hand. We don't care about the relationships between variable and we don't want to understand them

useful terms:

- matrix of features / covariates: the columns / variables we will use to make predictions or model our data

  columns we want

  `x = dataset.iloc[:, :-1].values` ... all columns except the last column

- target / outcome / dependent: what we want to predict or understand

  `y = dataset.iloc[:, -1].values` ... values of the last column

- hyperparameters: variables in the model eqn (i.e. choosing if we want an intercept in linear regression)

Data Preprocessing

replacing missing values

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values = np.nan, strategy = 'mean')
imputer.fit(X[:, 1:3])
x[:,1:3] = imputer.transform(x[:, 1:3])
print(x)
```

other strategies include using the most common value in the column

... replace missing values with column mean

... determines where missing values exist and what the column means are

covariates/matrix of features

all numerical columns why? The 'mean' strategy only works on numbers

... update the columns that had missing values by replacing missing values with column means

... to check

encoding categorical data

one hot encoding - for each category of a variable, we turn the category into it's own column. For instance, if we had a country column that had values Canada, USA, Mexico, we would turn Canada into it's own column, USA into its own column and Mexico into it's own column. Each column is basically a (binary) indicator fcn.

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers = [("encoder", OneHotEncoder(), [0])], remainder = 'passthrough')
x = np.array(ct.fit_transform(x))
```

which column do we want to change / hot encode

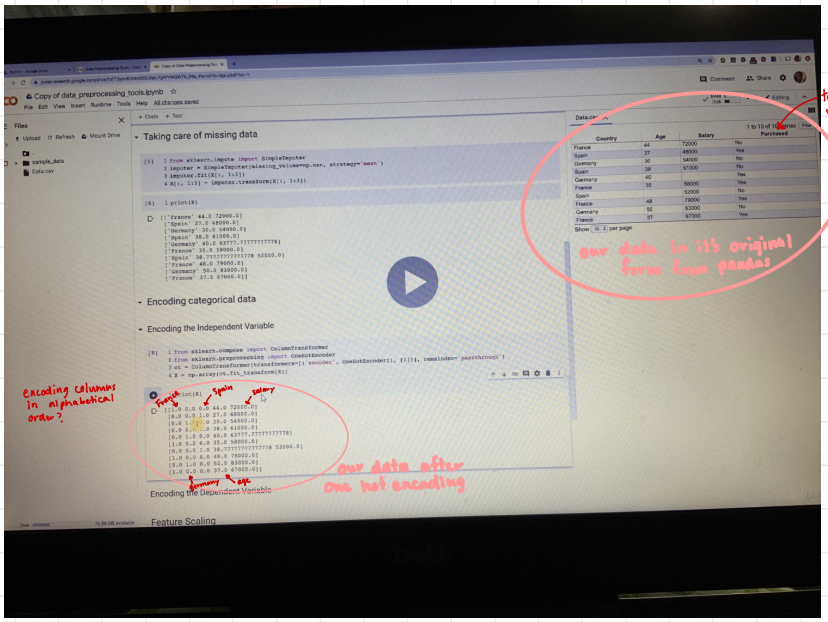
what type of transformation

what class will do the transformations

list of indexes for columns we want to transform

don't do anything to the other columns but return them in the new table

... hot encode our matrix of features



Think of OneHotEncoding as encoding categorical variables where order doesn't matter. Meaning the order variables of a column appear in the dataset has no affect / outcome / doesn't add info to our analysis or model.

Note, our target variable can also be a categorical variable of yes and no. We can simply change yes and no to zero and one respectively

```
import sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
```

LabelEncoder can also be used to encode categorial variables where order matters. Like the months in a year

We could also use this on our features column if it had a binary outcome instead of multiple outcomes

splitting the dataset

training set - for building the model
test set - used to evaluate the performance of the model

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 1)
```

X: matrix of features; y: target variable; test_size = 0.2: percent of dataset to turn into training set; random_state = 1: optional argument: takes randomness out of our samples; if we run this code again we get the same split of data

X_train, y_train: matrix of features and target variable for training set
X_test, y_test: matrix of features and target variable for training set

feature scaling

(important: this isn't needed for all machine learning models)

a method of putting all our features on the same scale.

why would you feature scale?
to prevent some features dominating others, such that all features are equally considered in the machine learning model.

some models such as regression don't need feature scaling bc it handles this problem in the model itself.

which machine learning models need feature scaling:

warning: don't use featuring scaling on binary data or columns that have been one-hot encoded

two forms of feature scaling:

i) standardisation

$$x_{stand} = \frac{x - mean(x)}{standard\ deviation(x)}$$

interval

. . . all features take values between (-3, 3) of standard deviation usually
↳ good in all cases

ii) normalisation

$$x_{norm} = \frac{x - min(x)}{max(x) - min(x)}$$

interval

. . . all features take value between (0, 1)
↳ good for normal data only

note, we do feature scaling on test dataset and training set seperately. If you don't you get information leakage, which is bad bc the training and test set must be independent

```
from sklearn.preprocessing import StandardScaler

stc = StandardScaler()
x_train[:, 3:] = stc.fit_transform(x_train[:, 3:])      ... standardising the training set
x_test[:, 3:] = stc.transform(x_test[:, 3:])            ... standardising test set
```

numerical columns that aren't indicator fans

standardise the test set using the mean and standard deviation of the training set

We get means and standard deviation values from the fit fun called on the train set prior

interpretation: we get values "in" standard deviations away from the mean

Regression

Used to **predict continuous real values** (like salary data).

types of regression

- simple regression - linear model for one column/variable of continuous data
- multiple linear regression - linear model for multiple columns/variables of continuous data
- polynomial regression - regression for non-linear relationships of continuous data
- support vector for regression (SVR)
- decision tree classification
- random forest classification

Simple linear regression

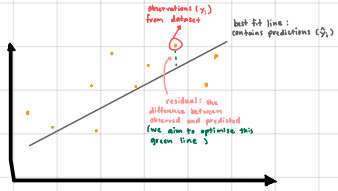
$y = b_0 + b_1 x_1$

dependent variable/ what we are trying to explain

independent variable/ what causes or is associated with the dependent variable

intercept

coefficient of independent variable; how a unit change in "x" affects a unit change in "y"



Note, we optimise our regression using the ordinary least squares method where we try to minimise the distance between observed and predicted values

$\min(\text{sum}[(y - \hat{y})^2]) = \min(ss_{residual})$

look at Evaluating Regression Models section for more info

idea: make a line that best fits the data (or approximates it well)

dataset: a dataset with 2 columns - salary (in $) and YearsExperience. Each row represents one employee

import libraries
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

import dataset
```
dataset = pd.read_csv('Salary_Data')
x = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values      ... last column - Salary - is what we want to predict
```

split dataset
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2)      ... split data into training set and test set
```

fit the model
```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()      ... create a empty linear regression model
```

`regressor.fit(x_train, y_train)` ... fit the model to our data

to get our intercept and coefficient for the independent variable (aka our estimates):

`print(regressor.intercept_)` ... estimate of intercept

`print(regressor.coef_)` ... estimate of coefficient for independent variable

not available in this package `print(regressor.summary())` ... get p-values, R^2, R^2 adjusted, standard errors. Basically what we get in R-Studio

making predictions

`y_pred = regressor.predict(x_test)` ... make predictions for these values from our training set

visualizing our estimates

visualising our training set against the regression line

`plt.scatter(x_train, y_train, color='red')` ... plot the data we used to train the model

`plt.plot(x_train, regressor.predict(x_train), color='blue')` ... a regression line for the data in the training set

the individual estimates from the observed values in the training set

```
plt.title("Salary vs. Experience (Training Set)")
plt.xlabel("Years of Experience")
plt.ylabel("salary")
plt.show()
```

Here we see visually, how well our model fits the training set

code output:

we can also do a visualization for our test set / predictions

`plt.scatter(x_test, y_test, color='red')` ... plot the data from the dataset where our predictions exists

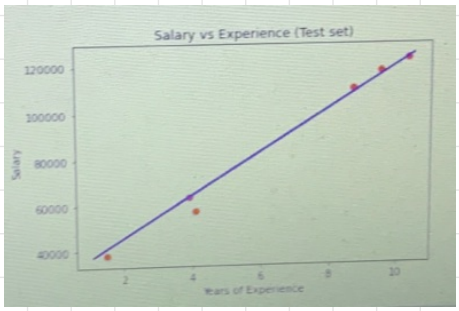
`plt.plot(x_train, regressor.predict(x_train), color='blue')` ... same regression line as above

the individual estimates from predicting values in the test set

↳ we can put any of the x-values from either dataset and get the same line. Why? All x-values exist on our line.

```
plt.title("Salary vs. Experience (Test Set)")
plt.xlabel("Years of Experience")
plt.ylabel("salary")
plt.show()
```

Here we see visually, how well our data fits the test set



model evaluation:

```
from sklearn.metrics import mean_square_error, r2_score
print("mean square error : %.2f" % mean_square_error(y_test, y_pred))
print("coefficient of determinant (R2): %.2f" % r2_score(y_test, y_pred))
```

... mean square error

... R^2 score

format the string and put in this variable everywhere there is a percent sign

checkout: A beginner guide to linear regression in python for scikit-learn

multiple linear regression

$$y = b_0 + b_1 x_1 + b_2 x_2 + \ldots + b_n x_n$$

(x_1, x_2: independent variables)

assumptions of linear regression:

i) **linearity** - there is a linear (additive) relationship between the dependent variable (response) and the independent variable (predictor/covariate).

ii) **homoscedasticity** - error terms must have constant variance.

iii) **multivariate normality** - error terms must be normally distributed

iv) **independent errors** - there should be no correlation between residual (error) terms. Basically look at a residual plot for no obvious patterns.

v) **lack of collinearity** - independent variables should not be correlated.

5 methods to build a model (idea: not all the variables will be great predictors of what we want so we have to reduce the variable we use to the essentials.):

1. **all-in** - we use all the variables given to us. When to use: when we know from prior knowledge these variables predict our response, or we were tasked with studying the interaction of all variables
2. **backwards elimination** - remove non-significant variables one at a time and re-evaluate the model (best method)
3. **forward selection** - add one variable at a time and test for statistical significance in that model and in the variable
4. **bidirectional elimination** - fit all possible regressions at the same time
5. **score comparison**

dataset:

we are given the columns "R&D Spend" (how much money was spent on R&D in one year by a company), "Administration" (how much money was spent on admin tasks like paying employees in one year), "Marketing Spend" (how much the company spent on marketing), "State" (where is the company) and "Profit" (profit made in one year). There are 50 rows in the dataset, each row represents a startup company.

We want to understand which what makes a company a good investment given the above columns. In other words, we are predicting profit.

In our case, x_1 = R&D, x_2 = administration, x_3 = marketing but we also have a categorical variable in the State column that we need to take care of. Thus, we will do One Hot Encoding on the state variable to create dummy/indicator variables. We leave one dummy variable out of the model to preserve interpretability of the data. The dummy variable we leave out becomes a reference for the other dummy variables.

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

dataset = pd.read_csv("50_Startups.csv")
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
```

alternative using pandas

```
X = dataset[['Petrol_tax', 'Average_income', 'Paved_Highways',
       'Population_Driver_licence(%)']]
y = dataset['Petrol_Consumption']
```

dummy variable: California, Florida, New York

dummy variables encoded alphabetically

```
[[0.0 0.0 1.0 165349.2 136897.8 471784.1]
 [1.0 0.0 0.0 162597.7 151377.59 443898.53]
 [0.0 1.0 0.0 153441.51 101145.55 407...54]
 [0.0 0.0 1.0 144372.41 118671.85 383...9.62]
 [0.0 1.0 0.0 142107.34 91391.77 366168.42]
 [0.0 0.0 1.0 131876.9 99814.71 362861.36]
 [1.0 0.0 0.0 134615.46 147198.87 127716.82]
 [0.0 1.0 0.0 130298.13 145530.06 323876.68]
 [0.0 0.0 1.0 120542.52 148718.95 311613.29]
 [1.0 0.0 0.0 123334.88 108679.17 304981.62]
 [0.0 1.0 0.0 101913.08 110594.11 229160.95]
 [1.0 0.0 0.0 100671.96 91790.61 249744.55]
 [0.0 1.0 0.0 93863.75 127320.38 249839.44]
 [1.0 0.0 0.0 91992.39 135495.07 252664.93]
 [0.0 1.0 0.0 119943.24 156547.42 256512.92]
 [0.0 0.0 1.0 114523.61 122616.84 261776.23]
 [1.0 0.0 0.0 78013.11 121597.55 264346.06]
 [0.0 0.0 1.0 94657.16 145077.58 282574.31]
 [0.0 1.0 0.0 91749.16 114175.79 294919.57]
 [0.0 0.0 1.0 86419.7 153514.11 0.0]
 [1.0 0.0 0.0 76253.86 113867.3 298664.47]
 [0.0 0.0 1.0 78389.47 153773.43 299737.29]
 [0.0 1.0 0.0 73994.56 122782.75 303319.26]
 [0.0 1.0 0.0 67532.53 105751.03 304768.73]
 [0.0 0.0 1.0 77044.01 99281.34 140574.81]
 [1.0 0.0 0.0 64664.71 139553.16 137962.62]
 [0.0 1.0 0.0 75328.87 144135.98 134050.07]
```

We can know what dummy variables represent what country by comparing X.head() before and after one hot encoding

X after one hot encoding

encoding categorical data: One Hot Encoder

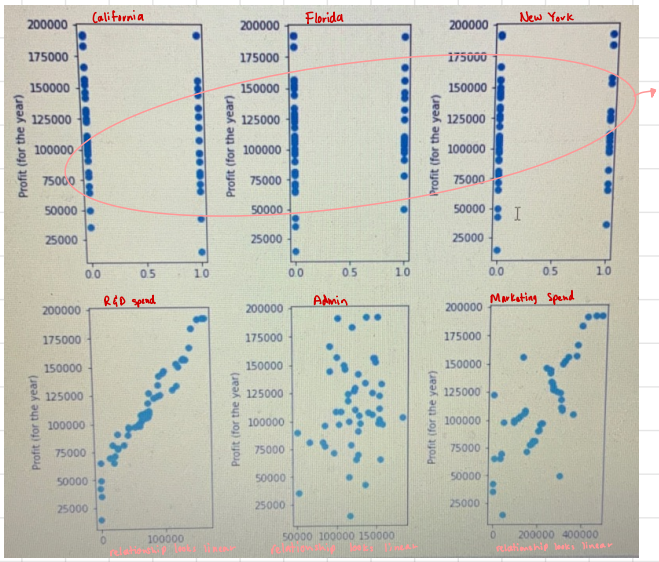
```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[("encoder", OneHotEncoder(), [3])], remainder="passthrough")
X = np.array(ct.fit_transform(X))

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

checking linearity

```
# build a graph to look for linear relationships in all columns when plotted against our response/
# dependent variable
col_index = 0
plt.figure(figsize=(9,9))
for column in X_train.T:
    plot_pos = int("23" + str(col_index + 1))   ... parameter to create a grid of plots
    plt.subplot(plot_pos)
    plt.scatter(X_train.T[col_index], y_train.T)   ... plot one column at a time against our response variable
    plt.ylabel("Profit (for the year)")
    col_index += 1

plt.subplot_adjust(wspace=0.7)   ... amount of white space between plots
plt.show()
```



fitting the model

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)   ... builds the linear model & selects statistical significant variables/columns
```

making predictions

```
y_pred = regressor.predict(X_test)
np.set_printoptions(precision=2)   ... print all numerical values with two decimal places
print(np.concatenate(y_pred.reshape(len(y_pred), 1), y_test.reshape(len(y_test), 1)), axis=1)
```

we can also suppress scientific notation using: `np.set_printoptions(suppress=True, formatter={'float_kind':'{:0.2f}'.format})`

reshape array to this many rows; reshape array to this many columns; turn this row vector of observations into a column vector; axis=1 means add the rows of vector 2 onto vector 1; combine the columns of these vectors

Output of the concatenate fcn

```
predictions   observations from test set
[[103015.2  103282.38]
 [132582.28 144259.4 ]
 [132447.74 146121.95]
 [ 71976.1   77798.83]
 [178537.48 191050.39]
 [116161.24 105008.31]
 [ 67851.69  81229.06]
 [ 98791.73  97483.56]
 [113969.44 110352.25]
 [167921.07 166187.94]]
```

alternatively, instead of using the concatenate fcn we can also output data using pandas

```
print(pd.DataFrame({"predictions": y_pred, "actual": y_test}))
```

to get our intercept and coefficient for the independent variable (aka our estimates):

```
print(regressor.intercept_)   ... estimate of intercept
print(regressor.coef_)   ... estimate of coefficient for independent variable
```

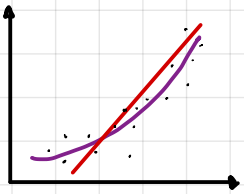
place our coefficients in a dataframe so we know what variables the model used

```
coeff_df = pd.DataFrame(regressor.coef_, X.columns, columns=['Coefficient'])
coeff_df
```

polynomial regression

$$y = b_0 + b_1 x_1 + b_2 x_1^2 + \ldots + b_n x_1^n$$

- b_0: intercept
- $b_1 x_1$: same variable, but different
- $b_2 x_1^2$: as all about just different
- x_1: dependent
- $b_n x_1^n$: regression coefficient
- increasing the exponent of a variable until the relationship is properly captured



- polynomial regression: $y = b_0 + b_1 x_1 + b_2 x_1^2$ (parabola)
- linear regression

when to use?

some times the data we have isn't a linear relationship but rather a polynomial. For example, in the above code we see that a polynomial of degree 2 fits the data better than a straight line.

dataset: a table containing job titles (position), pay-scale (level) and the salary for a given position

	A	B	C
1	Position	Level	Salary
2	Business Analyst	1	45000
3	Junior Consultant	2	50000
4	Senior Consultant	3	60000
5	Manager	4	80000
6	Country Manager	5	110000
7	Region Manager	6	150000
8	Partner	7	200000
9	Senior Partner	8	300000
10	C-level	9	500000
11	CEO	10	1000000

Imagine we are an HR manager at company A who is negotiating the salary of an inbound employee whose last company was company B. This table represents the salaries of people from company B. Imagine the employee was Regional Manager with a few years experience. We want to predict the employees old salary but it won't be exact because the longer you have been in a position the more money you get. To get around this we can use a pay-scale level between Regional Manager (6) and Partner (7).

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

dataset = pd.read_csv("Position_Salaries.csv")
X = dataset.iloc[:, 1].values   . . . note the position column and level column hold the same info
y = dataset.iloc[:, -1].values
```

alternative code:

```
X = dataset[["Level"]]
y = dataset[["Salary"]]
```

note, we don't want to split the dataset as we can't make accurate predictions without every row in this case

fitting the linear model

we are creating a linear model only to compare w/ the polynomial model we are about to build

linear regression model

```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(X, y)
```

fitting a polynomial model

polynomial regression model

```
from sklearn.preprocessing import PolynomialFeatures
poly_var = PolynomialFeatures(degree = 2)   we can adjust this as needed
X_poly = poly_var.fit_transform(X)   . . . increase the power of the x-variable to a power of two
poly_reg = LinearRegression()   . . . we use a linear regression bc even though the variable isn't linear, our output can still be expressed as a linear combination of regression coefficients
poly_reg.fit(X_poly, y)
```

visualizing the linear model

plot the linear model

```
plt.scatter(X, y, color = "red")   . . . plot the points from the dataset (aka the real salaries from our dataset)
plt.plot(X, lm.predict(X), color = "blue")   . . . make the regression line
plt.title("Linear Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()
```

As we can see from the graph made from the code above, the linear model doesn't fit this data.

visualizing the polynomial model

plot the polynomial model

```
plt.scatter(X, y, color = "red")      . . . plot the points from the dataset (aka the real salaries from our dataset)
plt.plot(X, poly_reg.predict(X_poly), color = "blue"). . . make the regression line
plt.title("Polynomial Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()
```

predicting new results

```
lin.predict([[6.5]])      . . . predicted salary from linear regression model
```

an array (table) of one row and one column

```
poly_reg.predict(poly_var.fit_transform([[6.5]]))
```

note: the array

[[6.5, 2], [5.1, 3.7]]

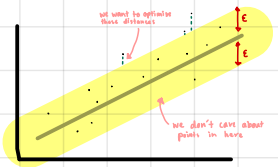
is equivalent to the table

column # 1	column # 2
6.5	2
5.1	3.7

Support vector regression (SVR)

additional resources: core.ac.uk/download/pdf/81523322.pdf
↳ chapter 4: support vector regression by Mariette Awad (2015)

Think of this as a best fit line (aka regression line) with a epsilon-insensitive tube.



what is an ε-insensitivity tube?
It is a region where we don't care about the residual of any points contained in the region.

the difference between SVR and linear regression is that instead of minimizing the residual (linear regression), we want to minimize the distance from any point outside the ε-insensitivity tube to the tube itself.
↳ we call these distances "slack variables". Any distance above the ε-insensitive tube is called ξ_i and any distance below the ε-insensitive tube is called ξ_i^*.

we want to minimize:
$$\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*)$$

Thus, the points outside the tube dictate the best fit line (aka the placement of the line). Therefore, we can use this technique to account for any errors in the model

Why is it called support vector regression?
Each point outside the epsilon-insensitive tube can be thought of as a vector from the origin. It is these vectors that hold/support the epsilon-insensitive tube in place.

Note, SVR models can be both linear and non-linear.
dataset: We will use the same dataset that was used in the polynomial regression section of these notes

building a simple SVR model:

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

dataset = pd.read_csv("Position_Salaries.csv")
X = dataset.iloc[:, 1].values   . . . note the position column and level column hold the same info
y = dataset.iloc[:, -1].values
```

alternative code:
```
X = dataset[["Level"]]
y = dataset[["Salary"]]
```

note for SVR, we do need to do feature scaling on our data because unlike regression, SVR doesn't have (regression) coefficients that balance variables with high and low values.

```
y = y.reshape(len(y), 1)   . . . turn y into a column vector/array; this is need for the feature scaling fxn
                  # of rows we want, # of columns we want

from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X = sc_x.fit_transform(X)   . . . feature scale "x" bc its values are drastically different than y
y = sc_y.fit_transform(y)   . . . feature scale "y" independently from "x" bc we don't want any information bleeding
```

feature scaling

X and y after being feature scaled

```
[8] 1 print(X)

[[-1.5666989 ]
 [-1.21854359]
 [-0.87038828]
 [-0.52223297]
 [-0.17407766]
 [ 0.17407766]
 [ 0.52223297]
 [ 0.87038828]
 [ 1.21854359]
 [ 1.5666989 ]]

  1 print(y)

[[-0.72004253]
 [-0.70243757]
 [-0.66722767]
 [-0.59680786]
 [-0.49117815]
 [-0.35033854]
 [-0.17428902]
 [ 0.17781001]
 [ 0.88200808]
 [ 2.64250325]]
```

fitting the model

fitting the SVR model

```
from sklearn.svm import SVR
svr = SVR(kernel = 'rbf')
svr.fit(X, y)
```

question:

- how do we choose a kernel for SVR?
- how do we choose the parameters to set in our SVR model (i.e. C, degree, epsilon, shrinking, etc.)

making predictions

prediction

```
pred_scaled = svr.predict(sc_x.transform([[6.5]]))
```

... predictions in standardized scalar form

```
sc_y.inverse_transform(pred_scaled)
```

... returns values on the same scale as the original data we started with

graphing our model

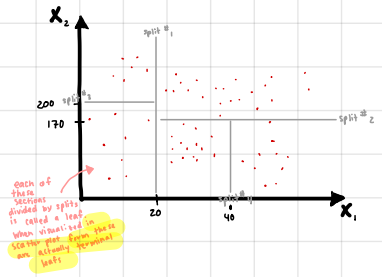
visualizing the SVR model

remove the scaling from the features in y

```
plt.scatter(sc_x.inverse_transform(X), sc_y.inverse_transform(y), color = 'red')
plt.plot(sc_x.inverse_transform(X), sc_y.inverse_transform(svr.predict(X)), color = 'blue')
plt.title('Support Vector Regression')
plt.xlabel('Position Level')
plt.ylabel('Salary')
plt.show()
```

since our model makes predictions based on scaled data, we must input scaled data into the model but predictions then unscale any predicted values

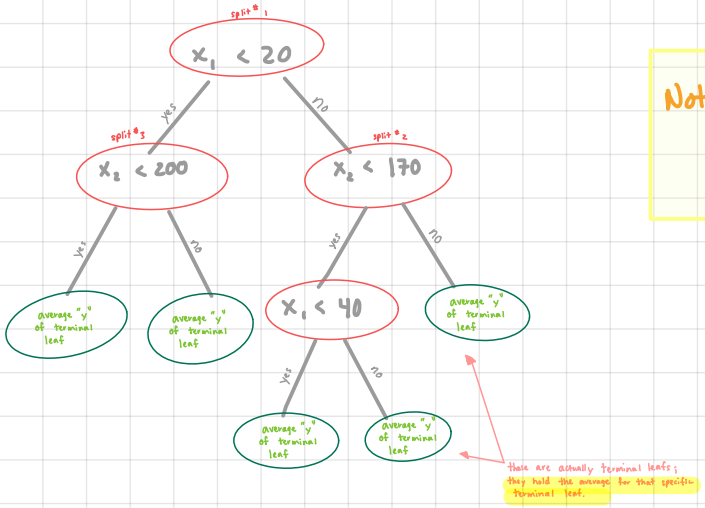
Decision Tree Regression



Imagine you have two independent variables (x_1, x_2) and a dependent variable (y). The algorithm would then split the graph of x_1 vs. x_2 into several sections using the concept of information entropy (think: when you do a split, are you getting more information about a point or are we adding value to the way we want to group our points). When no more info can be added by splitting our data, the algorithms stops splitting the data.

def^n leaf: sections of the data (or graph) that are seperated by splits in the data (as visualized above). The final leaves are called terminal leafs.

decision trees can also be visualized as follows:
This is what the prediction algorithm is doing in the background.



Note: no feature scaling is needed for decision trees

how does the model make predictions?
It takes the average of "y" for all the data in each leaf. Thus, if a new data point we want to estimate falls onto one of these leafs, that point can be estimated using the average of the leaf.

When to use instead of (linear/multiple) regression:
- for non-linear relationships like parabolas
- when you have a cloud of data (like we see in the scatterplot above); in this case linear regression would give us a formula with little predictive power

dataset: we will use the same dataset from the polynomial regression section; note decision trees are best used for datasets with multiple features. Thus, this is probably a bad example of the decision tree model in use but its good as a example to learn from.

building a simple decision tree model:

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

dataset = pd.read_csv("Position_Salaries.csv")
X = dataset.iloc[:, 1].values
y = dataset.iloc[:, -1].values
```

this is where would check for missing values and encode categorical data

building the decision tree model

fitting the model

```
from sklearn.tree import DecisionTreeRegressor
dt = DecisionTreeRegressor()
dt.fit(X, y)
```

making predictions

predictions

```
dt.predict([[6.5]])
```

visualizing the decision tree

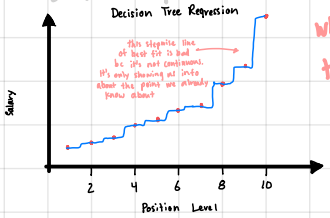
```
X_grid = np.arrange(min(X), max(X), 0.1)
```
. . . make a list starting from the minimum value of X to the max value of X, which increases by a value of 0.1 until it reaches the max. Thus we make an X with more values than the original dataset.

```
X_grid = X_grid.reshape((len(X_grid), 1))
```
. . . make this a column vector

```
plt.scatter(X, y, color = "red")
plt.plot(X_grid, dt.predict(X_grid), color = "blue")
```
. . . makes a more detailed plot than just using the 10 entries we had prior

```
plt.title("Decision Tree Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()
```

graph output



Note, this type of visualization won't be possible when we have data of higher dimensions (i.e. data that have two or more independent variables).

Here we can see why decision trees are best for data with multiple features.

Random Forest Regression

This model is a form of ensemble learning.

defn ensemble learning: when you take multiple algorithms or the same algorithm multiple times and you put the algorithms together to make something much more powerful than the original algorithm.

the random forest algorithm:

1. pick "k" data points at random from the training set
2. build a decision tree associated with these specific "k" data points
3. choose "n" number of trees you want to build. Repeat steps 1 & 2
4. make each tree predict the value of "y" for each new data point. The average of the predictions from all our trees become the prediction we recieve from the model.

Why is this useful?

By doing the algorithm above, we get a more accurate and stable model than a single decision tree. The model is also less prone to error as, any change in the dataset has a higher probability of impacting a single decision tree than a forest of trees.

building a simple random forest regression model:

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

dataset = pd.read_csv("Position_Salaries.csv")
X = dataset.iloc[:,1].values
y = dataset.iloc[:,-1].values
```

building the random forest model

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators = 10)   # n_estimators = # of decision trees
rf.fit(X,y)
```

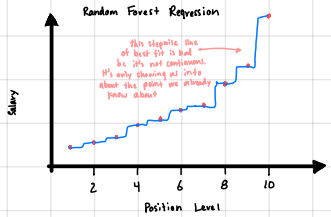
making prediction

```
rf.predict([[6.5]])
```

visualizing random forest regression model:

```
X_grid = np.arrange(min(X), max(X), 0.1)
X_grid = X_grid.reshape((len(X_grid), 1))
plt.scatter(X, y, color = "red")
plt.plot(X_grid, rf.predict(X_grid), color = "blue")
plt.title("Random Forest Regression")
plt.xlabel("Position Level")
plt.ylabel("Salary")
plt.show()
```

Note, this type of visualization won't be possible when we have data of higher dimensions (i.e. data that have two or more independent variables).



Here we can see why decision trees are best for data with multiple features, and bad for data with single features.

Evaluating Regression Models — How to select the best regression model

R^2

recall, in least square regression the aim was to try to minimize $(y-\hat{y})^2$. We call $\text{sum}(y_i-\hat{y}_i)^2$ the sum square of residuals (SS_{Res}). Thus,

$SS_{Res} = \text{sum}(y_i - \hat{y}_i)^2$ ($\hat{y}$: regression estimate / prediction) $=$ sum square residual

$SS_{tot} = \text{sum}(y_i - \bar{y})^2$ ($\bar{y}$: mean y) $=$ total sum of squares

We know that R^2 can be represented as

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = \text{goodness of fit}$$

We want an R^2 value as close to one as possible. Basically, how good is our regression at predicting values as compared to an average line (made of the mean of all data points). Think of this as predictive power or the goodness of fit for the regression model.

The problem:
the more variables we add the higher R^2 becomes (or in some cases it stays the same), meaning R^2 will never decrease (which we know if you add a non-sense variable R^2 should decrease).

solution:
we use a measurement that accounts for the number of variables in our model to assess the goodness of fit. We call this R^2 adjusted.

$$R^2_{adj} = 1 - (1-R^2)\frac{n-1}{n-p-1}$$ where n = sample size, p = number of regressors / independent variables

We say R^2_{adj} has a penalization factor because it penalizes you for adding independent variables that don't help your model.

We can use R^2 to assess any regression model (linear, polynomial, SVR, decision tree, random forest) using the following code:

```
from sklearn.metric import r2_score
r2_score(y_test, y_pred)
```

If you don't know which regression model to choose, build them all and choose the one with the best R^2 score.

note: the metrics module of scikitlearn holds all the functions needed for model evaluation.

Classification

classification is used to predict a categovy and not a continuous variable. There are linear classification models (like logistic regression & SVM) and non-linear models (like K-NN, SVM and random forest).