

Python Reference

for machine learning

by : Dwight

basic python syntax

pandas (dataframes)

matplotlib & plotly (graphs)

scikit-learn (machine learning models)

Python Basics

Legend
O notes
O code
O explanation

variable assignment

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

`spam amount = 5`

variable 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	<code>a+b</code>	sum of a & b
subtraction	<code>a-b</code>	difference of a & b
multiply	<code>a*b</code>	product of a & b
division	<code>a/b</code>	quotient of a & b
floor division	<code>a//b</code>	quotient of a & b , removing fractions

modulus	<code>a % b</code>	integer remainder after division
exponent	<code>a ** b</code>	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()`

`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

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

defining a fcn syntax:

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

It is best practice to put a docstring in a fcn

fcns with a docstring:

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

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

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

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	$a == b$	$a < b$	$a \leq b$	$a != b$	$a > b$	$a \geq b$

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
```

list functions:

```
len(prime) . . . length of a list
```

```
sum(prime) . . . sum the list
```

Output:

```
[Alex, 3, 5, 7]
```

list methods:

a method is a fun attached to an object

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

list operators

operator	example
in	5 in prime

Tuples

idea: an immutable/unchangeable list

example:

`t = (1, 2, 3)` different ways of
`t = 1, 2, 3` doing the same thing

Loops

for loop syntax:

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

range loop syntax:
`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}`
 \rightarrow can only hold immutable data types

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)
 \rightarrow Predefined Variables:
 insert these values to the string where there are curly brackets

Importing libraries

libraries add more functionality to base python

`import math` ... math is a module (aka a collection of variables)
`print(dir(math))` ... dir() allows use to see all the variables in a module

We can access variables using dot syntax:

`pi = math.pi`

modules also have funcs in them:

`math.log(32, 2)`
`help(math.log)` ... tells us how to use the log func in math module

alias import statements:

`import math as mt` ... instead of using math we can use mt

List Comprehensions

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

This executes first saying: make a list from 0-4
square = [n**2 for n in range(10)] ... output: [0, 1, 4, 9, 16, 25, ..., 81]

short_planet = [planet for planet in planets if len(planet) < 6]
 \rightarrow ① a list of planets whose names are shorter than 6 characters
 \rightarrow ② what happens when the condition is met

While loop:

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

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

Dictionaries

A list of key-value pairs

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

indexing a dictionary:

`numbers['one']` ... output: 1

adding to a dictionary:

`numbers["eleven"] = 11`

changing values in a dictionary:

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

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

import all:

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 sublibraries held in variables

Pandas

installing pandas

pip install pandas

or

conda install pandas

importing pandas

import pandas as pd

get a summary of data
dt.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:

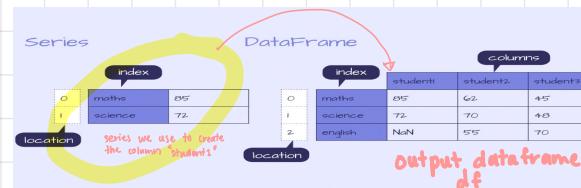
d = {'student1': pd.Series([85., 72.], index = ['maths', 'science']),
'student2': pd.Series([62., 70., 55.], index = ['maths', 'science', 'english']),
'student3': pd.Series([45., 48., 70.], index = ['maths', 'science', 'english'])}

df = pd.DataFrame(d)

print(df.head())

Column values

row names
in-order from top-to-bottom



We can also use
a dictionary of lists

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') 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('select * from penguins', con)

describing query
to see what's in our query

database connection info

query

connect to database

③ read general text files:

df = pd.read_table('filename', sep = '\t')

what separates each column

\t tab symbol

export dataframe to csv file:

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

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 fn

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

option to set

how many columns to max out at

accessing data

get the names of columns in the dataframe

`df.columns`

view a column

- 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]`

integer

removing individual/a group of columns

`df.drop(columns = [column-names])`

conditional selection of rows

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[<condition1> & <condition2>]` ... dataframe satisfying 2 conditions

`df.loc[<condition1> | <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()`

pandas types:

Type	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

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

↳ 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 rows alphabetically between 'Apples' and 'Potatoes'

we also can also use the fn `is null` to get a table where the values in a column are null. For example:

`df.loc[df.column-name.isnull()]`

remove missing values

`df.dropna(inplace = True)`

replace missing values

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

what to put where there are missing values

`df.column-name.fillna('unknown')`

replacing incorrect values:

`df.column-name.replace(old-value, new-value)`

can be
of any
type

when to remove: you have a large dataset with very few (%) 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')`

Column name

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)`

return a new transformed Series/
column

input

transformation

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

map fcn

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 datframe we can do aggregations on

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

`df.groupby('points').price.min()` ... return the lowest price for every point rated

`df.groupby('winery').apply(lambda table: table.title.iloc[0])` ... the first wine every reviewed by every winery

Column name
in this case title is
the column containing wine
names

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

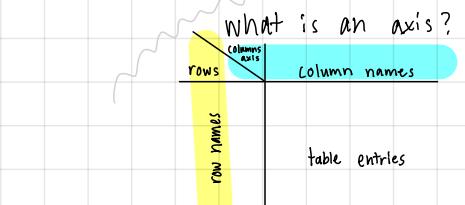
sort columns

for the want to apply column

sort by increasing value

`df.sort(column=column-name, ascending=True)`

(can also use `sort_values(column-name, ascending=False)`)



renaming columns/row indices/axis names

`df.rename(columns={old-column-name: 'new-column-name'})` . . . rename a column

`df.rename(index={old-index1: new-index1})` . . . rename row indices

`df.rename_axis('wines', axis='rows').rename_axis('fields', axis='columns')`

change of row axis to this

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(...)`] same column names in each table

`british-views = pd.read_csv(...)`

`pd.concat([canadian-views, british-views])`

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

table names & first column with the column names

column second set by default from right table

add this string to the end of all column names from the left table

add this string to all column names from the right table

melt = pivot longer
pivot = pivot wider
wide-to-long = separate

Tidying a dataframe

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	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	69.8	99.1	90.3

Table of pig weights by feed

`pigs = pd.read_csv(...)`

`pigs['id'] = df.index`

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

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

name of column(s) put in new column

a list

what we will call the new column that has the old column names

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

list of columns we want to combine into one column

Output:

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

QED □

Pivot longer & separate

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

tb data frame

country_code	year	males between ages 0-4	males between ages 5-14	females between ages 0-4
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='|')

→ index of the columns
names we want to
separate from; can also
be a character like '-'

this is also where
values from these columns
of the old table will be stored

→ characters to
use as an
id variable

→ name of new
column being created
then will contain
the first part of the column
name

outpnt:

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 = tb2.melt(id_vars=['iso2', 'year', 'age'],

value_vars=['m', 'f'], var_name='gender',
value_name='frequency')

what columns we
want to combine

what remains
constant ('in
terms of columns')

new column name
where values will be stored

→ new column
contains old
names

tb-clean data frame

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

QED □

Pivot wider

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

table "df"

countries	metrics	values
A	population_in_million	100
B	population_in_million	200
C	population_in_million	120
A	gdp_per capita	2000
B	gdp_per capita	7000
C	gdp_per capita	15000

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

table "df2"

countries	gdp_per capita	population_in_million
A	2000	100
B	7000	200
C	15000	120

each variable
has its own
column

common statistical funcs

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 'color: %s' % colour
```

```
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%}")
```

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()
```

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

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 component 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(111)
```

represents 3 arguments: # of rows, # of columns, plot number
(in that order)
can also be written
2,2,1

... an axes object

```
# Scatter the data
```

```
ax.scatter(np.linspace(0,1,5), np.linspace(0,5,5))
```

```
# show the plot
```

```
plt.show()
```

thus, a subplot of (2,2,1) is a 2x2 grid where each grid can hold a graph. The last 1 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))
```

make an area to put 2 graphs in one row

width of graph in inches

height of graph in inches

```
ax1 = fig.add_subplot(121) ... plotting area for graph 1
```

```
ax2 = fig.add_subplot(122) ... plotting area for graph 2
```

basic customization

specific axes we are working on

```
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
```

we can also use the argument blank=True

can also take the values: left and right

```
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
```

Saving a plot

save an image of our plot:

```
plot.savefig("file-name", transparent=False)
```

do we want a transparent image?
example: image.png
optional argument

save a pdf of our plot:

```
from matplotlib.backends.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
```

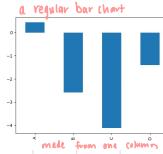
```
plt.close() ... close a window that popped up to show a plot
```

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 dataframne

histogram (for one column)

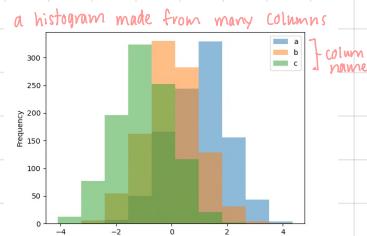
using matplotlib with pandas

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

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

Other arguments:
bins, orientation, cumulative
@ dt bins: horizontal / vertical

do we want a cumulative histogram?



Column names

boxplots

using matplotlib with pandas

```
df.boxplot(column = 'column_name', by = 'column2-name')
```

y-variable

x-variable

scatterplot

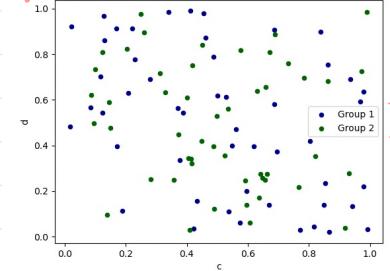
a regular scatterplot

```
df.plot.scatter(x = "x-variable", y = "y-variable")
```

Name of column we want to put on the x-axis

Name of column we want to put on the y-axis

grouped scatterplot - method # 1



label names

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
Group 2
put points on same scatter plot
grouped by 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)
```

filter rows: show only rows that meet this condition

Column name

Column name

Plotly (graphing library)

An alternative to Matplotlib

to install the library:

!pip install plotly == 4.0.2

setup a plotly account:

<https://plotly.settings/api/#/>

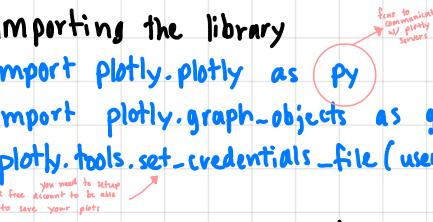
importing the library

import plotly.plotly as py

import plotly.graph_objects as go

plotly.tools.set_credentials_file(username='Your Account', api_key='Your API Key')

you need to return
to 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 up to 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": "#3859b5", "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)
```

What will our subgroup

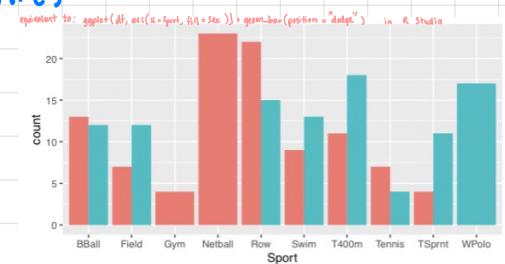
female_sport_freq = []

for sport in df.Sport.unique():

```
    count = df[ (df.Sport == sport) & (df.Sex == 'female') ].count()
    female_sport_freq.append(count)
```

don't show duplicate entries

a list/column of how many values are in each sport (in same order as x-variable)



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.

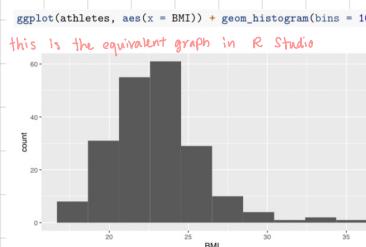
visit: plotly.com/python/bar-charts/

histogram (for continuous data or "dist" like it)

basic histogram

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

trace



normalized / probability histogram

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

instead of y-axis being count/frequency,
y-axis is probability of each value
in the dataset

multiple histograms on one plot

x0 = np.randomn(500) ... generate a sample size of 500

x1 = np.randomn(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)
```

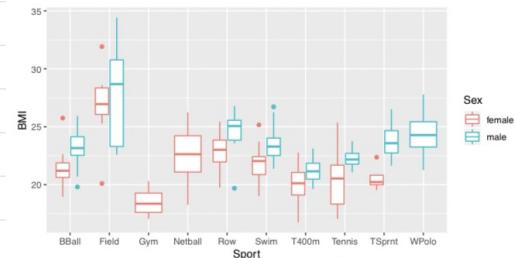
grouped boxplot

```
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 column are went for subgroup to be from

```
ggplot(athletes, aes(x = Sport, y = BMI, colour = Sex)) +  
geom_boxplot()
```

this is the R Studio equivalent of what we just wrote



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 column are went for subgroup 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=dfAAPL_High))  
fig.update_layout(xaxis_range=[2010-07-01, "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 scikit-learn

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
 $x = \text{dataset}[:, :-1].values$... all columns except the last column
- target / outcome / dependent: what we want to predict or understand
 $y = \text{dataset}[:, -1].values$... values of the last column

Data Preprocessing

replacing missing values

from sklearn.impute import SimpleImputer

imputer = SimpleImputer(missing_values=np.nan, strategy='mean') ... replace missing values with column mean
imputer.fit(X[:, 1:3]) ... determines where missing values exist and what the column means are

$X[:, 1:3] = \text{imputer.transform}(X[:, 1:3])$... update the columns that had missing values by replacing missing values with column means
print(X) ... to check

other strategies
include using and
most common
value in the
column

encoding categorical data

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

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

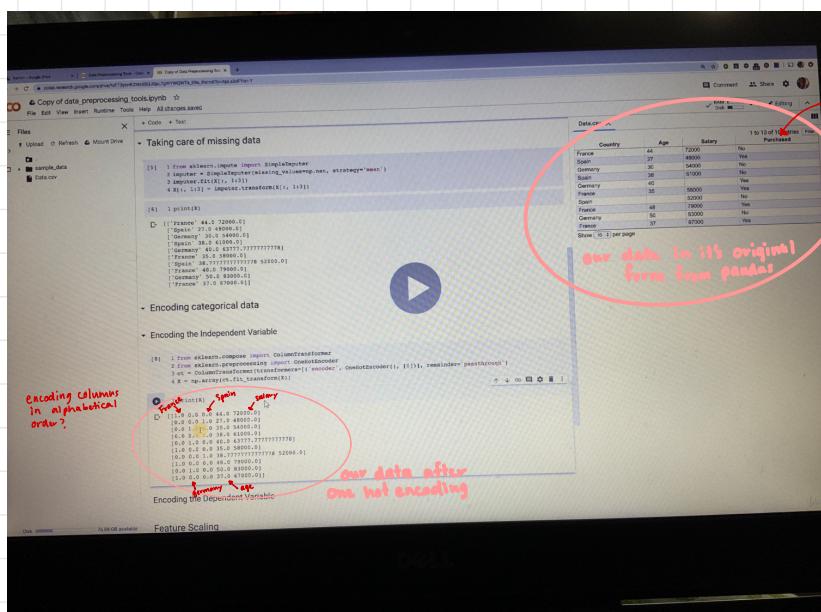
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [0])], remainder='passthrough')
 $X = \text{np.array}(ct.fit_transform(X))$... hot encode our matrix of features

which columns do we want to change / but encode?

what type of transformation

what class will do the transformation

list of indices for columns we want to transform
don't do anything to the other columns but return them in the new table



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)
```

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)
```

The diagram shows the `train_test_split` function call with four parameters: `x`, `y`, `test_size=0.2`, and `random_state=1`. `x` and `y` are labeled as 'matrix of features' and 'target variable' respectively. A bracket under `x` is divided into two parts: 'matrix of features and target variable for training set' and 'matrix of features and target variable for test set'. The `test_size` parameter is annotated with 'percent of dataset to turn into training set'. The `random_state` parameter is annotated with 'optional argument: takes random seed out of our samples; if we run this code again we get different splits at each split'.

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 feature scaling on binary data or columns that have been one-hot encoded

Two forms of feature scaling:

i) standardisation

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

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

ii) normalisation

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

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

Note, we do feature scaling on test dataset and training set separately. 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
```

```
sc = StandardScaler()
```

```
x_train[:, 3:] = sc.fit_transform(x_train[:, 3:]) . . . standardising the training set
```

```
x_test[:, 3:] = sc.transform(x_test[:, 3:]) . . . standardising test set
```

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

Regression

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

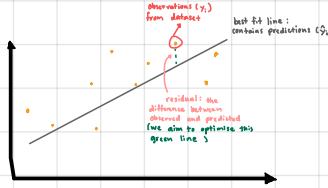
types of regression

- 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 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_1 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 $\min[(y - \hat{y})^2]$

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 numpy as np
```

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

```
import pandas as pd
```

```
{ 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
```

```
{ 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
```

```
{ from sklearn.linear_model import LinearRegression
```

```
regressor = LinearRegression() . . . create an empty linear regression model
```

L 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

print(regressor.summary()) ... get p-values, R², R² adjusted, standard errors. Basically what we get in R-Studio

not available
in this
package

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

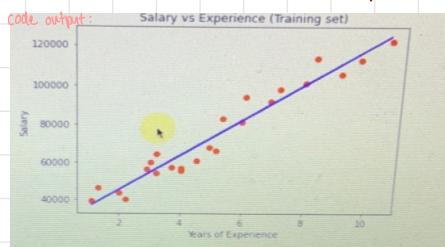
plt.title("Salary vs. Experience (Training Set)")¹⁾

plt.xlabel("Years of Experience")

plt.ylabel("salary")

plt.show()

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



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

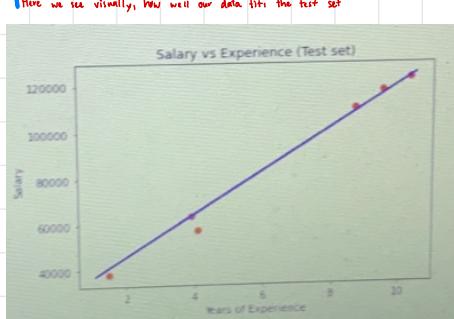
plt.title("Salary vs. Experience (Test Set)")¹⁾

plt.xlabel("Years of Experience")

plt.ylabel("salary")

plt.show()

Here we see visually, how well our data fits 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.

model evaluation:

from sklearn.metrics import mean_squared_error, r2_score

print("mean square error: %.2f" % mean_squared_error(y-test, y-pred)) ... mean square error

print("coefficient of determinant (R2): %.2f" % r2_score(y-test, y-pred)) ... R² score

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

Multiple Linear Regression

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

↑ 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
- 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, Administration, Marketing Spend, State and Profit". 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, $x_1 = \text{R&D}$, $x_2 = \text{administration}$, $x_3 = \text{marketing}$ 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
```

```
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)
```

We can know what dummy variables represent what country by comparing X[heads] before and after one hot encoding.

↳ alternative using pandas

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

The transformed dataset looks like this:

	California	Florida	New York	R&D spend	Marketing spend
[1, 0, 0, 0, 1]	165349.2	136897.8	471784.1		
[1, 0, 0, 0, 1]	162597.7	151377.59	443898.53		
[0, 1, 0, 0, 1]	153441.51	101145.55	407134.54		
[0, 1, 0, 0, 1]	144327.41	118671.85	38319.62		
[0, 1, 0, 0, 1]	142107.34	91391.77	366168.42		
[0, 0, 1, 0, 1]	131876.9	99814.71	362861.36		
[1, 0, 0, 0, 0]	134615.46	147198.87	127716.82		
[0, 0, 1, 0, 0]	130298.13	145530.05	323876.68		
[0, 0, 0, 1, 0]	120542.52	148718.95	311613.29		
[1, 0, 0, 0, 0]	123334.88	108679.17	304981.62		
[0, 1, 0, 0, 0]	101913.08	110594.11	229160.95		
[1, 0, 0, 0, 0]	100671.96	91790.61	249744.55		
[0, 1, 0, 0, 0]	93863.75	127320.38	249839.44		
[0, 0, 0, 0, 0]	91992.39	135495.07	252664.93		
[0, 0, 1, 0, 0]	119943.24	156547.42	256512.92		
[0, 0, 0, 1, 0]	114523.61	122616.84	261776.23		
[1, 0, 0, 0, 0]	78013.11	121597.53	264346.06		
[0, 0, 0, 0, 1]	946557.16	145077.58	282574.31		
[0, 0, 1, 0, 0]	104175.59	141375.59	4919.57		
[0, 0, 0, 1, 0]	86419.59	153851.11	0.0		
[1, 0, 0, 0, 0]	76253.86	113867.3	298664.47		
[0, 0, 0, 1, 0]	78389.47	153773.43	299737.29		
[0, 0, 1, 0, 0]	73994.56	122782.75	303319.26		
[0, 0, 1, 0, 0]	67532.53	105751.03	304768.73		
[0, 0, 0, 1, 0]	77044.01	99281.34	140574.81		
[1, 0, 0, 0, 0]	64664.71	139553.16	137962.62		
[0, 0, 0, 0, 1]	75000.07	130500.49	300500.07		

↳ after one hot encoding

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(column_index + 1)) ... parameter to create a grid of plots
plt.subplot(plot_pos)

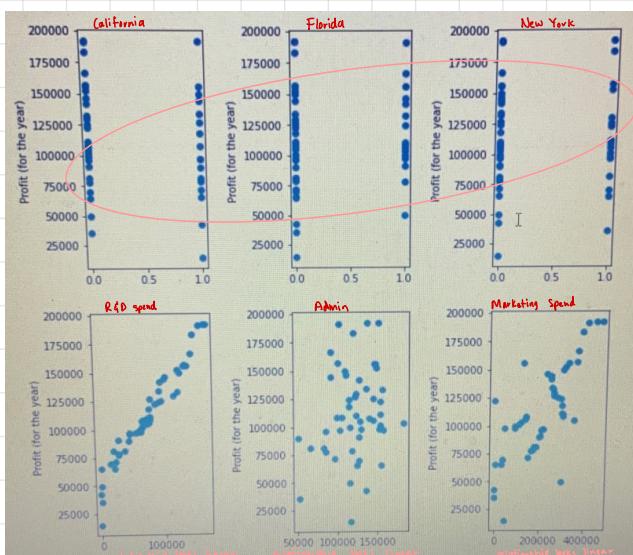
plt.scatter(X_train.T[column_index], y_train.T) ... plot one column at a time against our response variable

plt.ylabel("Profit (for the year)")

column_index += 1

plt.tight_layout(wspace=0.7) ... amount of white space between plots

plt.show()



from sklearn.linear_model import LinearRegression

regressor = LinearRegression()

regressor.fit(X_train, y_train) ... builds the linear model & selects statistically 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))

Output of the concatenate fn

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]]

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 raw vector of observations into a column vector

(combine the columns of these vectors)
axis=1
axis=0 means add the result of vector 1 onto vector 2

alternatively, instead of using the concatenate fn we can also output data using pandas
print(pd.DataFrame({'predictions': y_pred, 'actual': y_test}))

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

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

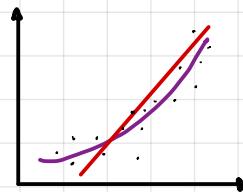
print(regressor.intercept_) ... estimate of intercept

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

polynomial regression

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

Intercept
regression coefficient
a linear variable with different degrees
increasing the exponent of a variable until the relationship is properly captured



— polynomial regression: $y = b_0 + b_1 x_1 + b_2 x_1^2 + \dots + b_n x_1^n$
— linear regression

when to use?

sometimes the data we have isn't a linear relationship but rather a polynomial. For example, in the above code we see that a polynomial at 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
y = dataset.iloc[:, -1].values
```

fitting the linear model

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

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

fitting a polynomial model

```
{ from sklearn.preprocessing import PolynomialFeatures
poly_var = PolynomialFeatures(degree=2)
X_poly = poly_var.fit_transform(X)
poly_reg = LinearRegression()
poly_reg.fit(X_poly, y)
```

... increase the power of the x-variable to a power of two
... 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

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

visualizing 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

```
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

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

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

note: the array

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

is equivalent to the table

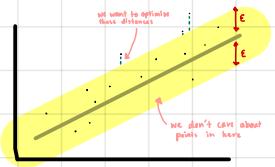
Columns	Columns
0 1	0 2
6.5	2
5.1	3.7

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 -insensitivity tube is called ξ_i and any distance below the ϵ -insensitivity 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 fun

of rows
we want
of columns

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

x and y after being feature scaled

[8] 1 print(X)
[] [-1.5666989] [-1.21854359] [-0.87038288] [-0.52223297] [-0.17407766] [0.17407766] [0.52223297] [0.87038288] [1.21854359] [1.5666989]

1 print(y)
[] [-0.72004253] [-0.79243757] [-0.66722767] [-0.59680786] [-0.49117815] [-0.35033854] [-0.17428902] [-0.17781001] [0.88200808] [2.64250325]

fitting the model

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

making predictions

```
pred_scaled = svr.predict(sc_X.transform([[6.5]]))  
sc_y.inverse_transform(pred_scaled)
```

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.)

graphing our model

```
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()
```

remove the scaling from the features in y

... predictions in standardized scalar form

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

Since our model makes predictions based on scaled data, it must input scaled data into the model for predictions than unscale any predicted values.

Decision Tree Regression