Learn Python

File extension : '.py'

Code (ip stored in a variable) → used to take input

```
name = input('Your name? ')
print('Helo ' + name)
```

(print prints)

```
Your name? Ankit
Helo Ankit
```

(user input)

HOW PYTHON WORKS?

Python gets written as source code. We use this translation service called an interpreter to run this code, and then we'll split out some instructions for our machine, so that the code can be executed.

```
name = input('Hi')
print('Helo' + name)
```

(takes ip from user)

(gives op)

```
print(type(6))
```

returns datatype (int, float, etc)

In python: `2**5 = 2^5 = 2^5`, `5//4` = returns rounded off integer on division

`5%3 = 2` → returns remainder, `round(3.9)` = rounds off value

`abs(-3) = +3` → returns absolute value

{ eg: 5//4 = 1
  eg: 5/4 = 1.25 }

Operator Precedence: `()` > `**` > `*` (or) `/` > `+` (or) `-`

- **Number**
  - **Integer**
    ```
    print(5)
    => 5
    ```
    Also; (binary → int)
    ```
    print(int('0b101', 2))
    => 5
    ```
    (base is) (base of no. inside brackets)
  - **Decimal**
    ```
    a = 3.4
    print(type(a))
    => <class 'float'>
    ```
  - **Complex**
    ```
    print(complex(3,4))
    => 3+4j
    ```
  - **Binary**
    ```
    print(bin(5))
    => 0b101
    ```

```
=> a, b, c = 1, 2, 3
   print(a)
   => 1
```

{ Assigning values to multiple variables at once }

```
=> print('hi')        (String)
=> long_string = '''
   wow
   ooo
   ---
   '''
   print(long_string)
```

WOW
OOO

-: we can use `''` (or) `""` (or) `''' '''` (better for longer sentences)

```
=> print('a' + 'b')   # string concatenation
```

(Comments)

→ str()
↳ (converts to string)

→ a = str(100)
b = int(a)
print(type(b))
↳ op: <class 'int'>
} #type conversion

→ Escape Sequence

weather = "\t It's \"kind of\" sunny \n take care"
print(weather)

It's "kind of" sunny
take care

\t = given tab space
\n = moves to next line (on new)
\ = it tells, "whatever comes after me, it's a string"

→ ex: name = 'Ankit'
0 1 2 3 4 → (index of string)

print(name[0]) → A
- (name[0:3]) → Ank [start:stop] = [start, stop)
- (name[0:5:2]) → Akt [start:stop:stepover]
- (name[1:]) → nkit prints all values after index 1
- (name[:3]) → Ank prints all values before index 3
- (name[::2]) → Akt step by 2 (for whole string)
- (name[-1]) → t prints last letter i.e sequence start from end of string
- (name[::-1]) → tiknA

→ Formatting Sequence:

age = 18
print(f"hi{name}. you are {age} old.") → hiAnkit. you are 18 old.
print('hi{}. you are {} old.'.format(name, age)) → hiAnkit. you are 18 old.

(help in inserting variable values inside a string)

* String is immutable i.e. once declared, it can't be changed.

Ex name[0] = '2' → error
print(name)

{we can only change full string}
ex name = 'yellow'

→ Few features of String

qu = 'to be alive'
print(qu.upper()) → TO BE ALIVE
- (qu.capitalize()) → To be alive
- (len(qu)) → 11
- qu.find('be') → 3
- (qu.replace('be', 'me')) → to me alive
- (qu) → printed original as string is immutable → to be alive

→ Boolean

print(bool(0)) → False
bool(1) → True
print(bool('True')) → True
print(bool('False')) → False

→ if: print('*' * 10) → op: **********

→ year = input("Birth year = ?")
↳ this input will be taken in string form

→ In python, lists are a form of arrays (mutable)
(ordered sequence of obj.'s that can be of any datatype)

ex: cart = ['a', 'b', 'c', 'd']
0 1 2 3 → index

print(cart[1:3]) → ['b', 'c']
new_cart = cart → new_cart = original one!
new_cart[0] = 'g'
print(new_cart) → ['g', 'b', 'c', 'd']
print(cart) → ['g', 'b', 'c', 'd']
(so any changes in new_cart = changes in org. cart)

a copy of cart is made ↑

new_cart = cart[:]
new_cart[0] = 'g'
print(new_cart) → ['g', 'b', 'c', 'd']
print(cart) → ['a', 'b', 'c', 'd']

→ Matrix ≡ 2D array ~ ex: (in py) matrix = [[1,5,1], [0,1,0], [1,0,1]]
print(matrix[0][1]) → 5

→ ex: basket = [1, 2, 3, 4, 5]

basket.append(100)
basket.insert(5, 77)
print(basket)
basket.pop()
basket.remove(3)
basket.clear()

These are few methods!
- append(100) → adds 100 in last of list
- insert(5,77) → adds '77' at index 5
- pop() → removes last element in the list
- remove(3) → removes element whose value is 3
- clear() → removes all element in the list

If we do new_list = basket.append(100)
print(new_list)
→ op: None
as we are just making changes in place (i.e. changes in basket), it's not returning any value

→ ex: bas = ['a', 'b', 'c', 'd', 'e']
find any ele.
print('x' in bas) → op: False
similarly, we can do for string
print('i' in 'yellow Hi') → op: True

# counting ele.
print(bas.count('d'))
↳ op: 1

→ bas.sort() ⟶ sorts list (original list()
(method)
• sorted(bas) ⟶ creates copy of list & then sorts it (no change in orgl)
(function)
• bas_new = bas.copy() ⟶ creates copy of bas/list.
• bas.reverse() ⇒ reverses order of list

→ #list patterns
ex: print(list(range(1,100))) (or) print(list(range(100)))
↓ ↓
[1, 2, 3, 4, 5, ... , 99, 100]

ex: new_sentence = ' '.join(['hello', 'hi', 'bye']) } → hello hi bye
print(new_sentence)
↓
(left) joins to every ele. on (right)

→ 'None' → keyword
↳ like 0 or Null in C++

→ #list unpacking
ex: a,b,c,*other = [1,2,3,4,5,6,7,8,9]
print(a) ⟹ 1
print(c) ⟹ 3
print(other) ⟹ [4,5,6,7,8,9]

{ if we want only '1', '2', '3' values, then we write *<variable_name> & this will store all the remaining items of the list.

if we also declare 'd' after 'other'
↳ then print(other) → [4,5,6,7,8]
print(d) ⇒ 9

→ Dictionary is an unordered key-value pair! (It also gives o/p in unordered form) (a string for us to grab the value)
ex: dictionary = {'a': [1,2,3], 'x': 'hello', 'b': True}
print(dictionary['a'][0]) ⇒ 1 → o/p
(if we try to do with list)
↳ my_list = [{'a': [1,2,3], 'b': 'hello', 'c': True}, {'a': [4,5], 'b': 4, 'c': False}]
print(my_list[0]['a'][2]) ⇒ 3 → o/p

→ A 'key' needs to be immutable
ex: dictionary = {123: [1,2,3], True: 'hello', '[100]': True}
print(dictionary[True]) ⇒ hello

If {'123': [1,2,3], '123': 'hello'}, print(dictionary['123']) → o/p: hello
↓
(∵ 123 will be overwritten if we have same key name)

→ user = {'basket': [1,2,3], 'greet': 'hello', 'age': 20}
print('size' in user) → o/p: false
• ('hello' in user.keys()) → # finds 'hello' in keys (left one) → o/p: False
• ('hello' in user.values()) # finds 'hello' in values (right one) → o/p: True
print(user.items()) → # o/p is tuple.
↳ dict_items([('basket', [1,2,3]), ('greet', 'hello'), ('age', 20)])
user.clear() # clears whole dictionary
user2 = user.copy() # allows us to copy user in user2
print(user.pop('age')) # pop returns the value whatever got removed
↳ Now 'age' wont exist in user
user.update({'age': 55}) # To update: make another dict{} & write what to update
print(user)
↳ o/p: (above/first line with 'age': 55)
↳ If key is not present in user, it will add a new one

→ Tuples are basically immutable lists. They're faster than lists
ex: tup = (1,2,3,4,5) # creating tuple
print(tup[3]) → 4
print(5 in tup) → True
(if) tup[1] = '2'
print(tup)
} error! as we are trying to change!

In above 'user', tuples are valid keywords in python
user = {(1,2): [1,2,3], ... }
# if we were to print
print(user[(1,2)]) → o/p: [1,2,3]

{ tuple that has a single element, tends to have a 'comma'

→ tup = (1,2,3,4,5,5)
print(tup.count(5)) → # counts total elements having value = '5'
• (tup.index(5)) → # prints index of element whose value = '5'
• (len(tup)) → # prints length of tuple
If we want diff. values in a tuple:
x,y,z,*other = (1,2,3,4,5)
print(x) → o/p: 1

→ Sets are ordered collection of unique items

ex: s = {2,1,2,4,5,5}
print(s) → op: {1,2,4,5}

s.add(100) # to add an element '100' in set : <set_name>.add(<value>)

We can also convert a list to set using 'set' keyword & vice versa

ex: lis = [—]
print(set(lis))
print(list(s))

print(len(s)) # to find length of set → op: 5
print(1 in s) # to find '1' in set → op: True

s1 = s.copy() → copying set
s.clear() → emptying set
print(s1) → op: {1,2,4,5,100}
print(s) → op: set()

→ ms = {1,2,3,4,5} # my-set
ys = {4,5,6,7,8,9,10} # your-set

print(ms.difference(ys)) → finds diff. btw both sets
→ op: {1,2,3}

print(ms.discard(5)) → removes that ele. from set
→ op: None
print(ms) → op: {1,2,3,4} → since we have discarded 5

print(ms.difference_update(ys)) → finds diff both sets & modifies it
→ op: None
print(ms) → op: {1,2,3}

<commented>

print(ms.intersection(ys)) → finds & prints common elements in both
(or) print(ms & ys) → op: {4,5}

print(ms.isdisjoint(ys)) → checks if sets are not overlapping (never doing)
→ op: False

print(ms.union(ys)) → uniting the sets
(or) print(ms | ys) → op: {1,2,3,4,5,6,7,8,9,10}

ex: ms = {4,5}
ys = {4,5,6,7,8,9,10}

print(ms.issubset(ys)) → checks if 'ms' is subset of 'ys'
→ op: True

print(ms.issuperset(ys)) → checks if 'ms' is superset of 'ys'
→ op: False

Learn Python Part 2

```
a = False
b = True
if(a and b):
    print('c1')
elif(a):
    print('c2')
else:
    print('c3')
print('done')
```

→ op: c3, done

→ We can use many elif's in our code!

Indentation is a syntactical requirement in Python
→ (white spaces)
ex: you can we have 4 spaces before: print('c1')

If indentation is not given, it will give me indentation error

if I do:
```
a = 'hello'
b = 5
#(if, elif, statement)
```
→ op: c1, done

because our 'if' basically convert conditions to boolean val

{ ex: bool('hello') → True, bool(5) → True }

(similar we have Falsey value) ← (Truthy value)

→ Shortcut for if else: "Ternary Operator"
→ syntax: <condition_if_true> if <condition> else <condition_if_false>

ex: a = False
d = "allow" if a else "not allow"
print(d) → op: not allow

Now we can also use 'short-circuiting' in 'if-else'!

ex: if (a and b) → T/F
a = False
'if a = false, then it won't check 'b'' → "and" (both)

ex: if (a or b) → F/T
a = True
'if a = true, then it won't check 'b'' → "or" (either (this) or (that))

→ ASCII of A = 65, B = 66, a = 97, b = 98 ; (a != b means a ≠ b)

→ '==' checks equality of value.

ex: print(True is 1) → False
" ("1" is "1") → True
" (10 is 10.0) → False

(is) → keyword
checks if location in memory where value is stored is same or not.

ex: print(True == 1) → True
" ("1" == 1) → False
" ([] == 1) → False
" (10 == 10.0) → True
" ([1,2,3] == [1,2,3]) → True

ex: print([] is []) → False
" ([1,2] is [1,2]) → False
(these 2 list are stored in diff. locations)

→ For loops

```
for i in 'Ankit Verma':
    print(i)
```

(can be anything → list / tuple / set)

```
for i in [1,2,3,4,5]
    print(i)
    print(i)
    print(i)
```

(this points to end = 5)

A N K I T V E R M A

Nested loops

```
for i in (1,2,3):
    for j in ['a','b','c']:
        print(i,j)
```

1a 1b 1c 2a 2b 2c 3a 3b 3c

Iterable simply means it is an object/collection that can be iterated over (one by one check item in the collection)

(like dictionary, tuple, set, string (prints keys))

ex: user = {
'name': 'Golem',
'age': 500
}

- for i in user: print(i) → name, age
- for i in user.values(): print(i) → Golem, 500
- for i in user.keys(): print(i) → name, age
- for i in user.items(): print(i) → ('name', 'Golem'), ('age', 500) (op in tuple form)

(if we don't want commas)

for key, val in user.items(): print(key, val) → name Golem, age 500

→ print(range(100)) → op: range(0, 100)

```
for i in range(5):
    print(i)
```

0 1 2 3 4 ✓

```
for j in range(0,10,2):
    print(j)
```

0 2 4 6 8

```
for k in range(10,0,-1):
    print(k)
```

10 9 8 7 6 5 4 3 2 1

```
for _ in range(2):
    print(list(range(10)))
```

op: [0,1,2,3,4,5,6,7,8,9] [0,1,2,3,4,5,6,7,8,9]

→

```
for i,char in enumerate('Heslloo'):
    print(i, char)
```

(can also work on tuple/list)

0 H 1 e 2 s 3 l 4 o 5 o

- Enumerate is very useful if we need the index of counter of the item that you're looping through.

→

```
i=0
while i<50:
    print(i)
    i+=1
else: # break
    print('done')
```

op: 0 1 2 ... 49 done

- If 'break' is present, then 'else' won't work
- When 'while' loop done, then 'else' will run.

→

```
for item in [1,2,3]:
    continue
    print(item)
```

op: ...

- "continue" makes the current iteration stops & moves to next iteration, so restarting for/while loop.

→

```
for item in [1,2,3]:
    pass
    print(item)
```

op: 1 2 3

- "pass" basically tells python to move to next line of code

(Q) picture = [[0,0,0,1,0,0,0], [0,0,1,1,1,0,0], [0,1,1,1,1,1,0], [1,1,1,1,1,1,1], [0,0,0,1,0,0,0]]

print → 0 = " " , 1 = "*"

Soln.

```
for i in picture:
    for j in i:
        if (j==0):
            print(' ', end=' ')
        else:
            print('*', end=' ')
    print()
```

The default value of → end = \n → so everytime code runs, it goes to next line

Here we changed from '\n' to ' ' & after end of inner for loop we print() → (end = \n)!

(Q) Check for duplicates in the list: ['a','b','c','b','d','m','n','n']

(A)

```
some_list = ['a','b','c','b','d','m','n','n']
duplicates = []
for i in some_list:
    if some_list.count(i) > 1:
        if i not in duplicates:
            duplicates.append(i)
print(duplicates)
```

op: ['b','b','n','n']

(if this line not present (keyword))

op: ['b','n']

Functions → def <func. name>: } ex: def hi():
<it's code> print ('Helloo')
hi() ↝ op: Helloo

But these func. can only be used after the declaration line.

Now we can also pass values!

ex: def hi(name, emoji): ← parameters → variables that we receive
print(f'Hello {name}{emoji}') } op: Hello ankit 😊
hi('ankit', 😊) → arguments → actual values we provide to func.

→ default parameters

→ def hello(name='Chimp', emoji='🐒'):
print(f'Hello {name}{emoji}')

hello('Dan', '😊') → positional arguments
hello(name='Bi', emoji='+') → keyword argument
hello('Jonny') } default
hello()

→ **Returns** def sum(num1, num2):
num1+num2
print(sum(10,5)) ↝ op: None
(Since it's not returning anything)

def sum(n1,n2):
return n1+n2
print(sum(10,5)) ↝ op: 15

We can also do func inside func. →
def sum(num1, num2):
def add(n1,n2):
return n1+n2
return add(num1, num2)
print(sum(10,5)) ↝ op: 15

Functions: max(), min(), ... { **Methods**: → (we use dot notation)
ex: "hello". capitalize
count
find ...

Docstrings ex: def test(a):
"""
Info: this func. tests & prints param a
"""
print(a)
test('!!!') ↝ op: !!!

Now docstrings are helpful, because if you hover over test, you will come to know it's use written in: """ """

We can also get info on user-based func. by
→ help(test) (or) print(test.__doc__)
↓ ↓
Keyword

ex: def super(*args):
print(args)
print(*args)
return sum(args)
print(super(1,2,3,4,5)) ↝ op: (1,2,3,4,5)
1 2 3 4 5
15

(stores in tuple form)

'args' → let us grab positional elements

ex: def super(*args, **kwargs):
total = 0
print(kwargs) # stores key values
print(*kwargs)
for i in kwargs.values():
total += i
return sum(args) + total
print(super(1,2,3,4, num1=5, num2=6)) ↝ op: {'num1': 5, 'num2': 6}
num1 num2
21

(stores in dictionary form)

'**kwargs' → keyword args allows us to grab any no of keyword arguments & get a dictionary.

In python, scope follows: Local
↓
Parent Local
↓
Global

If we want to use global func. in local use 'global' keyword.

ex: total = 0
def count():
global total
total += 1
return total
count()
print(count) ↝ op: 2

non-local:

ex: def outer():
x = "local"
def inner():
nonlocal x ← Keyword
x = "nonlocal"
print("inner:", x)
inner()
print("outer:", x)
outer() ↝ op: inner : nonlocal
outer : nonlocal

'nonlocal' helps us refer to the parent variable! So in the next line, we are making changes to parent variable 'x'.

Once func. is called, it's destroyed by garbage collector of Python. So we can't access it anymore!

```
ex: li = [1,2,3]
    def multby2(item):
        return item*2
    print(list(map(multby2, li)))
    print(li)
```

op: [2,4,6]
[1,2,3]
(total ips = total ops)

"map" → we give it some data & it will act upon it. It iterates automatically.

(if 'list' not mentioned, then we will get address of resultant map)

(no need to mention '()')

```
ex: li = [1,2,3]
    def onlyodd(item):
        return item%2!=0
    print(list(filter(onlyodd, li)))
```

op: [1,3]

(basically filters)

(its going to try & receive a boolean value)

(& then it checks whether it needs to be filtered or not)

if filtered → doesn't get add in final list

```
ex: li = [1,2,3]
    li2 = (10,20,30)
    print(list(zip(li, li2)))
```

op: [(1,10), (2,20), (3,30)]

(it takes the first ele., zips it & then goes to next ele.)

(if 'li2' has only 2 ele.s op: [(1,10), (2,20)])

ex: "reduce" → for this we need to import;

```
from functools import reduce
li = [1,2,3]
def accumulator(acc, item):
    print(acc, item)
    return (acc+item)
print(reduce(accumulator, li, 0))
```

op:
0 1
1 2
3 3
6

(initial value of 'acc')

(doesn't support op in list format)

syntax: reduce(function, sequence, intial)
(reduces to single value) (or iterable)

List Comprehensions: are used to make list in an easier way

Previously:

```
my_list = []
for char in 'hello':
    my_list.append(char)
print(my_list)
```

op: ['h','e','l','l','o']

But using list compre. :→ [<parameter> for <parameter> in <iterable>]

```
ex: li = [char for char in 'hello']
    print(li)
```

(we can name anything)

['h','e','l','l','o']

```
ex: li2 = [num for num in range(0,100)]
    print(li2)
```

[0, 1, 2, 3, ... 98, 99]

If we want numbers to be multiplied by 2.

```
ex: li3 = [num*2 for num in range(0,100)]
    print(li3)
```

[0, 2, 4, 6, ... 196, 198]

If we want $(num)^2$ → with only even no.'s

```
ex: li4 = [num**2 for num in range(0,100) if num%2==0]
    print(li4)
```

op: [0, 4, 16, ..., 9604]

Set Comprehension: Just change '[]' to '{}'

```
ex. li = {c for c in 'tikkiankit'}
    print(li)
```

op: {'i','t','a','k','n'}

Dictionary Comprehension:

```
di = {'a':1, 'b':2}
my_di = {k:v**2 for k,v in di.items()}
```

(grabs whole dictionary; k: key, v: value)

op: {'a':1, 'b':4}

We can also add 'if' at end;

```
my_di2 = {k:v**2 for k,v in di.items() if v%2==0}
print(my_di2)
```

op: {'b':4}

Suppose we want a dict. of num & (num)*2

```
di = {num:num*2 for num in [1,2,3]}
print(di)
```

op: {1:2, 2:4, 3:6}

Q) Find duplicates using comprehensions: ['a','b','c','b','d','m','n','n']

```
li = ['a','b','c','b','d','m','n','n']
li2 = [char for char in li if li.count(char)>1]
print(li2)
li3 = set([char for char in li if li.count(char)>1])
print(li3)
```

op: ['b','b','n','n']
{'b','n'}

(or we change li2 brackets from '[]' to '{}' → same op.)

→ Syntax for type of variable in Py: print(type(x))

→ Syn. to return 1st char of string: x="Hello"[0]

→ Method to replace parts of string: replace()

→ List is ordered, changeable & allows duplicate members

→ Multiline comment: """ ... """

→ Print length of any string
eg. x="Hello World"
print(len(x))
op: 11

→ Get char. from index 2 to index 4
txt="Hello World" → x=txt[2:5]

→ Replace char 'H' with 'J'
txt="Hello World"
txt=txt.replace("H","J")

→ Return String without whitespaces
eg txt=" Hello World " → x=txt.strip()

→ Insert "lemon" as 2nd item
fruits=["apple","banana"]
fruits.insert(1,"lemon")

→ Use (-)ve indexing & print last item
print(fruits[-1])

→ No. of items in a list: print(len(<list_name>))

→ Add multiple_items to a set;
fruits={"apple","mango"}
more_fruits={"kiwi","grapes"}
fruits.update(more_fruits)
print(fruits)
op: {'apple','grapes','mango','kiwi'} (in any order)

→ Print value of "model" in car dictionary, change year to 2020
car={"brand":"Ford",
"model":"Mustang",
"year":1964
}
print(car.get("model"))
car["year"]=2020

→ If we don't know the no. of arguments that will be passed into my func, use can use '*' prefix in func. definition
eg: def my_function(*kids):

→ Create class → "Hi"
class Hi:
x=5
p1=Hi() # object "p1" of class "Hi"
print(p1.x) # using object to print value of 'x'

→ If we don't know the no. of keyword arguments that will be passed into my func, we can use '**' prefix in func. definition
eg: def my_function(**hi):

→ Syntax to define "init" func. to a class : eg class Person:
def __init__(self,name,age):
self.name=name
self.age=age

→ Create class Student that inherits from class Person:
class Student(Person):

→ Print all variables & func.'s names of "mymodule" module;
import mymodule
print(dir(mymodule))

→ Syntax of importing only person1 dictionary of "mymodule" module
from mymodule import person1

→ Execute 'printname' method of obj x: class Person:
def __init__(self,fname):
self.printname=fname
def printname(self):
print(self.firstname)
class Student(Person):
pass
x=Student("Mike")
x.printname()

Modules are nothing but python files. We generally communicate btw diff files.

main.py
import utility # this is how we access other files
print(utility)
print(utility.mul(4,5))

utility.py
def mul(num1,num2):
return num1*num2

Packages are folders in which .py files are there

shopping
shopping_cart.py → def buy(item):
cart=[]
cart.append(item)
return cart
__init__.py
(It indicates that this is a Python package)

using in main.py
<package>.<module>
import shopping.shopping_cart
print(shopping.shopping_cart.buy('apple'))
op: ['apple']

Now we can import in many different ways
i) eg from shopping.shopping_cart import buy # importing a func from a package
print(buy('apple'))
ii) eg from shopping import shopping_cart # importing module from a package
print(shopping_cart.buy('apple'))

If there is a func. defined in a package & it's a keyword with diff use, then it gets overwritten if we call it. (like max → keyword → func. not use as user-defined word)

Data Science Environment Setup

A collection of tools/packages is called environment.
We use "Conda" to create the environment & then use it again to install different packages. It also allows us to share.

- .ipynb = i python notebook = jupyter notebook
  - (runs in order of execution of cells)

Example: Heart_Disease_Project

import pandas as pd (becomes its shortform)

df = pd.read_csv("heart-disease.csv")
- df = dataframe
- ("heart-disease.csv") = (comma seperated values)
- (press "tab" for autofill)

df.head() → shows first 5 rows of table
- (if we write any no. here, it will show those many rows of table)

import matplotlib.pyplot as plt → (to make graphs)

df.target.value_counts().plot(kind="bar")
- df = means df access 'target'
- target = (column name)
- value_counts() = counts no. of occurences of diff. values in that column
- plot = means create a plot
- kind="bar" = means "bar graph"

If we want to upload an image (markdown mode) (type image name)

```
![](6-step-ml-framework.png)
```

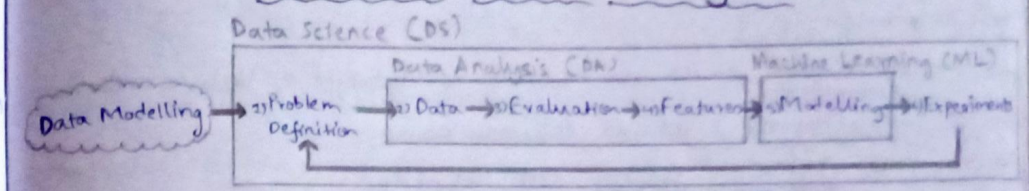
→ now: shift+enter

ls → it works in the same way as if we did 'ls' in command prompt. It shows us all files

In order to close the notebook → go to Jupyter notebook
- (press "ctrl+c" to exit)
- (you can see all updates which you did)

- Shift + Enter = Run
- esc = to remove edit opt.
- enter = to enter edit
- m = to enter 'markdown'
- y = to enter 'code'
- esc+b = for new cell below
- a = for new cell above
- esc+d+d = to delete cell
- select cell → type 'y' ← change to markdown (to save)
- '1' = to enter 'heading'

Pandas: Data Analysis



DA → pandas, matplotlib, numpy
ML → tensorflow, Pytorch, scikit learn, CatBoost, dmlc XGBoost
DS → jupyter, anaconda, miniconda

import pandas as pd

In this section, we will learn:
- → Most useful func.'s
- → pandas Datatypes
- → Importing & exporting data
- → Describing data
- → Viewing & selecting data
- → Manipulating data

How to open Jupyter Notebook:

→ Open "Jupyter" → click on its link

- Series = 1-D (takes "List" as input)
- Dataframe = 2-D (takes "dictionary" as input)

ex: check1 = pd.Series(["Hi", "Bye"])
check1

```
0    Hi
1   Bye
dtype: object
```

ex: car_data = pd.DataFrame({"Car": check1})
- "Car" = column name
- check1 = column values

car_data

	Car
0	Hi
1	Bye

row aka axis(0)
Column aka axis(1)
row-number starts by 0 by default

- Importing data:

car_sales = pd.read_csv("car-sales.csv")
car_sales
- read_csv = (Keyword)
- "car-sales.csv" = file name
- file extension
- # (print the csv data)

---------- CHANGES MADE ----------

- Exporting dataframe:

car_sales.to_csv("exported-car-sales.csv", index=False)
- "exported-car-sales.csv" = exported file name
- index=False = if we don't write, it will make index as new column if we try to open it. So we would have index of index! To avoid it, we keep "index=False"

new_sales = pd.read_csv("exported-car-sales.csv")
new_sales
- "exported-car-sales.csv" = file name to be read

* We can also read by using (" ") github link!

car = pd.read_csv("data/car.csv")
- "data/car.csv" = file location if in same directory

DESCRIBING DATA

- Attributes → no brackets
  - → it's just some meta info which is stored in our file (ex: size)
- Functions → brackets there
  - → performs some steps

eg: we have "car_sales.csv" file!

ex: (Attribute)

→ car_sales.dtypes : returns columns & datatype of values under it

→ car_sales.columns : returns columns name

→ car_sales.index : returns start, stop, step

ex: (function)

→ car_sales.describe() : returns info like; mean, min of numeric columns

→ car_sales.info() : its basically ((.index) + (.dtypes))

→ car_sales.mean() : returns mean value of numeric columns

→ car_sales.sum() : → adds the numeric values
↳ appends the string values

→ car_sales["Doors"].sum() : returns 'sum' of specific column named "Doors".

→ len(car_sales) : returns length of our data

→ car_prices = pd.Series([3000, 1500, 111250])
car_prices.mean()
↳ eg: 38583.336

SELECTING AND VIEWING DATA

→ car_sales.head() : returns first 5 values
↳ if we write any no. here, then it returns those many values from top/bottom.

→ car_sales.tail() : returns last 5 values

• .loc → refers to index no.s (↳ location) & .iloc → refers to position

eg: animals = pd.Series(["cat", "dog", "bird", "panda", "snake"], index=[0,3,9,8,3])
animals

0	cat
3	dog
9	bird
8	panda
3	snake

If we don't want custom index, no need to mention it

• animals.loc[3]

3	dog
3	snake

→ animals.loc[9] → 'bird'

→ animals.iloc[3] → 'panda'

→ animals.iloc[:3] → 0 1 2 → car_sales.loc[:3] (↳ calling head with val 4)

0	cat
3	dog
9	bird

If we want a particular column!

→ car_sales["Make"] —① or) car_sales.Make —② (Make = column name)

In ① → if we have spaces in column, it will run ✓
② → if we have spaces in column, it will not run ✗

→ car_sales[car_sales["Make"] == "Toyota"] ↳ boolean indexing
↳ eg: returns all values whose column name = Toyota

→ car_sales[car_sales["Odometer (KM)"] > 100000] ↳ eg: selects rows with over 100000 on odometer

→ pd.crosstab(car_sales.Make, car_sales.Doors)
↳ eg: makes a dataframe of 'Make' & 'Doors'

→ car_sales.groupby(["Make"]).mean() # groupby → similar to crosstab but focusing on more columns at a time
↳ eg: prints mean of diff. columns (using "Make" as reference) (only numeric)

→ car_sales["Odometer (KM)"].plot()
↳ eg: plots line graph of odometer!

→ if not running; (type ↓)
%matplotlib inline
#import matplotlib.pyplot as plt

→ car_sales["Odometer (KM)"].hist()
↳ eg: plots histogram

Now we have "Price" column in our car-sales.csv! It's dtype = 'O' (object)
↳ (₹4500.00)

Now to convert string to integer!
↳ Syntax → dataframe["amount"] = dataframe["amount"].str.replace(['₹', '.'], '').astype(int)
(↓ name)

→ car_sales["Price"] = car_sales["Price"].str.replace('[₹,.]', '').astype(int)

car_sales
↳ changes price values → (450000)
access car_sales["Price"] → we want to keep it as it is but access the string & replace the dollar sign, commas & dots with nothing (i.e empty string) & then change it from string to integer

Now to ignore decimal values → car_sales["Price"] = car_sales["Price"].astype(int)//100
↳ eg: 4500

MANIPULATING DATA

→ car_sales["Make"].str.lower() : returns lowered alphabets/words of column "Make"

But if we print → "car_sales", our lowered words won't be visible as change[so to change our original table, we can reassign it!

∴ car_sales["Make"] = car_sales["Make"].str.lower()

↳ Now our original table, make column changed to lowered words.

New file → car_sales_missing (has few blocks with no values)

(Shown with "NaN" in pandas)

If we want to fill those empty values, one way is reassigning }

2nd way

→ car_sales_missing["Odometer"].fillna(car_sales_missing["Odometer"].mean(), inplace = True)

helps us make changes in our table without reassigning

by default = False

car_sales_missing

↳ returns the table with 'NaN' values now filled with mean of Odometer

→ car_sales_missing.dropna() : returns only those rows which doesn't have 'NaN' values

If we have to reaccess our prev. data after 'inplacing', we have to import again.

→ car_sales_missing_dropped = car_sales_missing.dropna()
car_sales_missing_dropped

↳ returns fully filled rows!

Now exporting it!

car_sales_missing_dropped.to_csv("car-sales-missing-dropped.csv")

(file name)

Making column from series

seats_column = pd.Series([5,5,5,5,5])
car_sales["Seats"] = seats_column
car_sales

↳ fills first '5' values with '5' & rest all 'NaN'

↓ new column name

car_sales.fillna(5, inplace = True) → fills all values by 5
car_sales

Making column from Python List

fuel_economy = [7.5, 9.2, 5.0, 9.6, 8.7, 4.5, 6.0, 1.2, 7.0, 3.0]
car_sales["Fuel per 100km"] = fuel_economy
car_sales

↳ new column name

↳ list size = dataframe size

Making column from another column

car_sales["Total fuel (L)"] = car_sales["Odometer (km)"]/100 * car_sales["Fuel per 100km"]
car_sales

(column name)

Creating column from a single value

car_sales["Num of wheels"] = 4 → creates column "Num of wheels" with all its value = 4
car_sales

Now if we want to remove a column,

→ car_sales.drop("Total fuel used", axis = 1, inplace = True)
car_sales

(column name which is to be removed)

If we want to randomize our sample input:

→ car_sales_shuffle = car_sales.sample(frac = 1)
car_sales_shuffle

↳ randomizes rows & prints table

↳ '1' means shuffle 100% of data, 0.5 means 50% of data

↳ if we want to make changes in "car_sales" only, we have to do 'inplace'!

If we want to re-order/reset our index;

→ car_sales_shuffle.reset_index(drop = True, inplace = True)
car_sales_shuffle

↳ for permanent changes

↳ if we didn't mention this, then it will give new index for those random indexes, so to avoid it, we write!

↳ (eg: suppose → idx: 4, 3, 1, 2 → now 0, 1, 2, 3 ∴ 4→0, 3→1, 1→2, 2→3)

1 mile = 1.6 km

Now if we want to change km to miles in Odometer:

→ car_sales["Odometer (km)"] = car_sales["Odometer (km)"].apply(lambda x : x/1.6)
car_sales

↳ (eg: prints table with odometer values divided by 1.6)

↳ it says: apply → reset x = x/1.6

apply() → lets you apply some kind of func. whether it be numpy or lambda to a certain column

lambda : is a keyword in python which is short for anonymous func.

Now if want to rename a column,

→ car_sales = car_sales.rename(columns={"Odometer (KM)": "Odometer (Miles)"})
car_sales
(current name) (new name)

Imp point

Previously - when we changed "Price"! {rewriting in better way}
(Price → $4000.00)

remove punctuation from price column
car_sales["Price"] = car_sales["Price"].str.replace("[$,.]", "")
checking changes in Price column
car_sales["Price"] ↳ o/p 400000
removing the 2 extra zeroes at the end
car_sales["Price"] = car_sales["Price"].str[:-2]
checking changes
car_sales["Price"]
↳ o/p 4000
changing datatype of Price column to integers
car_sales["Price"] = car_sales["Price"].astype(int)

NumPy

Numerical Python

In this section, we will learn: → most useful func.'s
→ Numpy datatypes & attributes (ndarray)
→ Creating arrays → Viewing arrays & matrices
→ Manipulating & comparing arrays
→ Sorting arrays → Use cases

{import numpy as np}

DATATYPES AND ATTRIBUTES

Numpy's main datatype is "ndarray" ~ N-dimentional array

a1 = np.array([1,2,3])
a1
↳ o/p array([1,2,3])

a2 = np.array([[1,2,3.3],
[4,5,6.5]])

a2 → float becomes its datatype by default
↳ o/p returns above array

a3 = np.array([[[1,2,3],
[4,5,6],
[7,8,9]],
[[10,11,12],
[13,14,15],[16,17,18]]])

a3

→ a1.shape , a2.shape , a3.shape ↳ o/p (3,) , (2,3) , (2,3,3))
↳ keeping comman will also print our ans. in comman

→ a1.ndim , a2.ndim , a3.ndim ↳ o/p (1, 2, 3) → no. of dimensions

→ a1.dtype a2.dtype ↳ o/p int, float } → returns datatype of array

→ a1.size , a2.size , a3.size ↳ o/p (3, 6, 18) } → returns no. of elements in array

→ type(a1), type(a2), type(a3) ↳ o/p (numpy.ndarray, ,)

.Creating a dataframe from Numpy array:

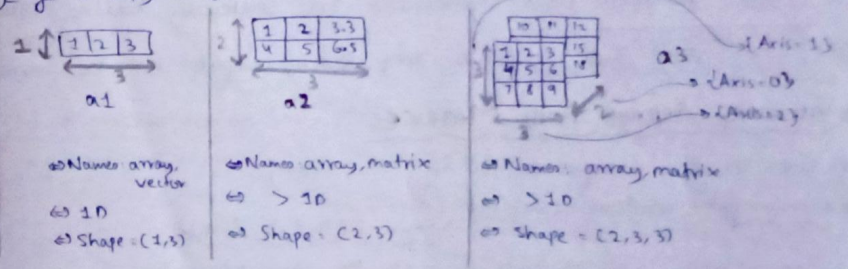
import pandas as pd
df = pd.DataFrame(a2)
df

o/p

	0	1	2
0	1.0	2.0	3.3
1	4.0	5.0	6.5

.Anatomy of Numpy array:

	1	2	3
Data	a1	a2	a3
Numpy	Names: array, vector 1D Shape: (1,3)	Names: array, matrix > 1D Shape: (2,3)	Names: array, matrix > 1D Shape = (2,3,3)
Details			



CREATING ARRAYS

• np.[func_name]
↳ (press 'Shift + tab' → opens docstring which gives us info. about that func.

→ ones = np.ones((2,3))
ones
↳ o/p array([[1., 1., 1.],
[1., 1., 1.]])
↳ returns new array of given shape & type, filled with ones

→ zeros = np.zeros((2,3))
R C
zeros
↳ o/p array([[0., 0., 0.],
[0., 0., 0.]])
↳ returns new array of given shape & type, filled with zeroes

→ np.arange(0, 10, 2)
start stop step
↳ o/p ([0, 2, 4, 6, 8])
↳ returns evenly spaced values with given interval
↳ (not saved in anything)

Now making random arrays!

→ random_array = np.random.randint(0, 10, size=(3,5))
random_array
↳ (returns random integers from low to high)

o/p 3x5 matrix with random values between 0 to 10

→ random_array.size, random_array.shape

→ np.random.random((5,3)) → returns random floats in half open interval [0.0, 1.0)

(eg: random 5×3 matrix with random values btw [0.0 & 1.0))

→ np.random.rand(5,3)

→ np.random.randn(5,3) → creates an array of normally distributed no. & size = 5

Now whatever the above op is generated is not random but pseudo random!

→ np.random.seed(seed=0) → any no. (for diff seed, diff. random no.s will be saved)

random_array_4 = np.random.randint(10, size=(2,3))

random_array_4

op: [[5,0,3], [3,7,9]]

Now no matter how many times we run this code, output will be same unless until seed value is changed

So those random no.s generated are basically having seed = (random value)

you can run the code in any device, output will be same!

□ VIEWING ARRAYS AND MATRICES

If we want unique no.s in an array:

→ np.unique(random_array_4) op: [0,3,5,7,9]

We can also use slicing concept to view arrays!

a3[:1,:2,:2] # means 1st matrix → 2 rows + 2 column elements

op: [[[1,2], [4,5]]]

a3[:2,:,:1] # means both matrix → all rows + 1st column elements (in 2nd matrix)

op: [[[1], [4], [7]], [[10], [13], [16]]]

- Furthest element on the right is furthest number of the shape gets displayed on the innermost section.

□ MANIPULATING AND COMPARING ARRAYS

- Arithmetic

→ a1+ones op: [2.,3.,4.] → a1-ones → a1*ones → a1*a2

→ a1/ones → a2/a1 (divide)

→ a2//a1 # (floor division - removes decimals & rounds down)

→ a2**2 # power 2

Above are simple python operations, but we can do numpy operations too

→ np.add(a1, ones) → np.square(a2) = a2**2

→ np.exp(a1) (e^a1, exponential of elements) → np.log(a1) (log of elements)

- Aggregation = performing same operation on number of things!

→ lis = [1,2,3]
type(lis) → op: list
sum(lis) → op: 6

a1 = np.array([1,2,3])
sum(a1) → op: 6
np.sum(a1) → op: 6

Use Python's method (sum()) or Python datatypes (eg: Numpy's methods on Numpy arrays (np.sum())

→ {Magic func. → has '%' symbol}

ex:
Creating a massive array
massive_array = np.random.random(100000)
massive_array[:10] → op: shows us first 10 values → (random values btw 0 & 1)

%timeit sum(massive_array) → Python's sum()
%timeit np.sum(massive_array) → Numpy's np.sum()

op: 7.79 ms ± 936 µs per loop (mean ± std of 7 runs, 100 loops each)

op: 411 µs ± 34.8 µs per loop (mean ± std dev. of 7 runs, 1000 loops each)

∴ We can see "np.sum()" is a lot faster than "sum()"

→ a2
op: [[1.,2.,3.3], [4.,5.,6.5]]

→ np.mean(a2) : op: 3.6333
→ np.max(a2) : op: 6.5
→ np.min(a2) : op: 1.0

→ np.std(a2) : op: 1.82269
→ np.var(a2) : op: 3.322222
→ np.sqrt(np.var(a2)) : op: 1.82269

- Standard deviation (std) = measure of how spread out of a group of numbers is mean

- Variance (var) = measure of the average degree to which each number is different to the mean
  - High variance: wide range of numbers
  - Low variance: lower range of numbers

- Standard deviation = square root of variance = $\sqrt{\text{variance}}$ (sqrt)

- Understanding std & var:

→ high_var_array = np.array([1, 100, 200, 300, 4000, 5000]),
low_var_array = np.array([2, 4, 6, 8, 10])

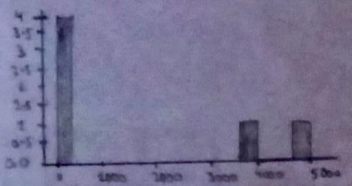
→ np.var(high_var_array), np.var(low_var_array) : (4296133.472222222, 8.0)

→ np.std(high_var_array), np.std(high_var_array) : (2072.711671626984, 2.8284271247461903)

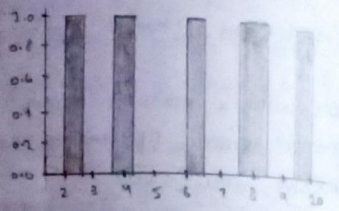
→ np.mean(high_var_array), np.mean(low_var_array) : (1600.1666666666667, 6.0)

→ %matplotlib inline
import matplotlib.pyplot as plt

→ plt.hist(high_var_array)
plt.show()



→ plt.hist(low_var_array)
plt.show()



- Reshaping And Transposing -

→ a2 * a3 : error

→ a2.shape, a3.shape : ((2,3),(2,3,3))

→ a2_new = a2.reshape(2,3,1)
↳ reshapes the dimension

- General Broadcasting Rules:
  2 dimensions are compatible when:
  ① They're equal (OR)
  ② One of them is 1

Now we can multiply!

→ a2_new * a3 : ✓

transpose - switches the axis

→ a2.T : [[1, 4],
[2, 5.],
[3, 6.5]]

→ a2.T.shape : (3, 2)

- Element-wise:

A	B	*	E	F	=	A*E	B*F
C	D		G	H		C*G	D*H
2×2			2×2				

- Dot-product:

A	B	C	. dot	I	K	A*I + B*L + C*N	A*K + B*M + C*O
D	E	F		L	M	D*I + E*L + F*N	D*K + E*M + F*O
G	H	I		N	O	G*I + H*L + I*N	G*K + H*M + I*O
3×3				3×2		3×2	

↳ must be same

Let mat1, mat2 = 2 matrices

→ mat1.shape, mat2.shape = ((5,3),(5,3))

→ mat1 * mat2 : ✓ → element wise multiplication

→ np.dot(mat1, mat2) : error → as shapes not aligned
↳ dot product

This is where we use "Transpose"!

→ mat3 = np.dot(mat1, mat2.T)
mat3
→ ✓

→ mat3.shape : (5,5)

- Nut Butter Sales (example → dot product) → refer jupyter for full code, here only imp. ones are there!

→ weekly_sales = pd.DataFrame(sales_amount (some array), index=["Mon", "Tue", "Wed", "Thur", "Fri"], columns=["Almond", "Peanut", "Cashew"])

- In Numpy, we can use same comparison operators which are in Python

a1 → [1, 2, 3] & a2 → [[1., 2., 3.3],
[4., 5., 6.5]]

→ a1 > a2 : array([[False, False, False],
[False, False, False]])

→ Similarly bool_arr = a1 >= a2
bool_arr

→ type(bool_array), bool_array.dtype
(numpy.ndarray, dtype('bool'))

→ a1 >= 4 → a1 == a1
[True, True, True]

→ a1 == a2
[[True, True, False],
[False, False, False]]

- Sorting arrays:

→ random_array = np.random.randint(10, size=(3,5))
random_array
[[7, 8, 1, 5, 9],
[8, 9, 4, 3, 0],
[3, 5, 0, 2, 3]])

→ np.sort(random_array)
[[1, 5, 7, 8, 9],
[0, 3, 4, 8, 9],
[0, 2, 3, 3, 5]])

(sorts the indexes)

→ np.argsort(random_array)
[[2, 3, 0, 1, 4],
[4, 3, 2, 0, 1],
[2, 3, 0, 4, 1]]
(columns → L2R)

→ a1 = [1, 2, 3]
np.argsort(a1)
[0, 1, 2]

→ np.argmin(a1) → (index of minimum number)
0

→ np.argmax(a1) → (index of maximum number)
2

→ np.argmax(random_array, axis=0)
[1, 1, 1, 0, 0]

→ np.argmax(random_array, axis=1)
[4, 1, 1]
(rows → T2B)

- Practical example:

```
→ from IPython.display import Image, display
# Specify path to image                    → image path
image_path = "numpy-images/panda.png"
# Display the image
display(Image(image_path))
```

displays image

```
#Turn image into numpy array
→ from matplotlib.image import imread
panda = imread("numpy-images/panda.png")
print(type(panda))
```

<class 'numpy.ndarray'>

→ panda.size, panda.shape, panda.ndim # if we zoom in, we see pixels (so our array stores these color values)

(24465000, (2330, 3500, 3), 3)

→ np.linspace(1, 100, 23) → Creates an array with 23 evenly spaced nos. between 1 to 100

Start, Stop, total evenly spaced nos

Matplotlib : Plotting And Data Visualization

In this section, we will learn:
- → Matplotlib workflow
- → Importing matplotlib & the 2 ways of plotting
- → Plotting data from Numpy arrays
- → Plotting data from pandas Dataframes
- → Customizing plots
- → Saving and sharing plots

```
%matplotlib inline
import matplotlib.pyplot as plt
```

A matplotlib workflow: → Create Data → Create plot (figure) → Plot data (axes on figure) → Customize plot → Save/Share plot

- Creating empty graph:

→ plt.plot()

→ plt.plot();

→ plt.plot()
plt.show()

creates figure

→ fig = plt.figure()
ax = fig.add_subplot()
plt.show()

adds some axes

```
→ x = [1,2,3,4]
  y = [11,22,33,44]
  plt.plot(x,y);
```

```
→ fig = plt.figure()
  ax = fig.add_axes([1,1,1,1])
  ax.plot(x,y)
  plt.show()
```

(dimensions of figure)

- Recommended method to create a figure:

```
→ fig, ax = plt.subplots()          (we can also replace this with a list)
  ax.plot(x,y) → adding data
  type(fig), type(ax)
```

(matplotlib.figure.Figure, matplotlib.axes._subplots.AxesSubplot)

- Example Workflow

→ Step 0: Import matplotlib & get it ready for plotting in Jupyter

```
%matplotlib inline
import matplotlib.pyplot as plt
```

Step 1: Prepare data

```
x = [1,2,3,4]
y = [11,22,33,44]
```

Step 2: Setup plot

```
fig, ax = plt.subplots(figsize=(5,5))
```

(width, height) # figure

Step 3: Plot data

```
ax.plot(x,y)
```

→ we are adding data to our axes, above we just created them!

Step 4: Customize plot

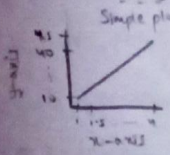
```
ax.set(title="Simple Plot", xlabel="x-axis", ylabel="y-axis")
```

Step 5: Save & show

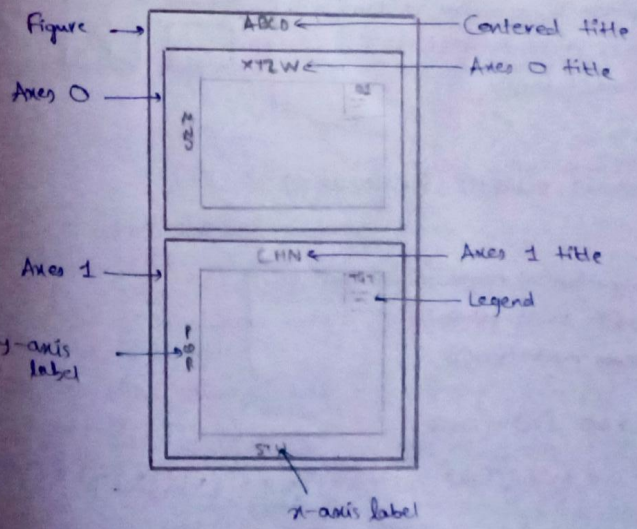
```
fig.savefig("File Images/sample-plot.png")
```

file path, image name

#



- Anatomy of a Matplotlib plot



- Making figures with Numpy arrays ; a) Line plot b) Scatter plot c) Bar Plot d) Histogram e) Subplots

Creating some data

```
x = np.linspace(0, 10, 100)
fig, ax = plt.subplots()
ax.plot(x, x**2);
```

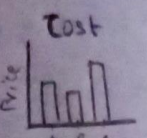
(Line plot by default) — x^2

```
fig, ax = plt.subplots()
ax.scatter(x, np.sin(x));
```

Making plot from dictionary:

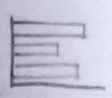
```
prices = {"A": 10, "B": 8, "C": 12}
fig, ax = plt.subplots()
ax.bar(prices.keys(), prices.values());
ax.set(title="Cost", ylabel="Price");
```

(not necessary)



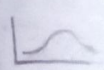
If we want horizontal bar graph, we have to pass "dict" as list!

```
fig, ax = plt.subplots()
ax.barh(list(prices.keys()), list(prices.values()));
```



```
x = np.random.randn(1000)
fig, ax = plt.subplots()
ax.hist(x);
```

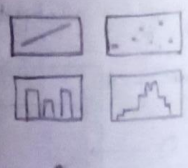
→ picks no. that lies in standard normal distribution curve



Now, there are 2 ways to create subplots;

Subplot option 1

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, figsize=(10,5))
# Plot to each diff. axis
ax1.plot(x, x/2);
ax2.scatter(np.random.random(10), np.random.random(10));
ax3.bar(prices.keys(), prices.values());
ax4.hist(np.random.randn(1000));
```



Subplot option 2

```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10,5))
# Plot to each diff. axis
ax[0,0].plot(x, x/2);
ax[0,1].scatter(np.random.random(10), np.random.random(10));
ax[1,0].bar(prices.keys(), prices.values());
ax[1,1].hist(np.random.randn(1000));
```

- Plotting from pandas Dataframe:

```
car_sales["Price"] = car_sales["Price"].str.replace('[\$\,\.]', '')
car_sales
```

(or `car_sales["Price"].str.replace('[\$\,]', '')`)

$4000.00 → 400000
$5000.00 → 500000

replaces '$', ',', '.' with '' ← empty string

```
car_sales["Price"] = car_sales["Price"].str[:-2]
car_sales
```

removes last 2 digits

400000 → 4000
500000 → 5000

```
car_sales["Sale Date"] = pd.date_range("1/1/2023", periods=len(car_sales))
car_sales
```

(new col.)

	Sale Date
0	2023-01-01
1	2023-01-02

```
car_sales["Total Sales"] = car_sales["Price"].astype(int).cumsum()
car_sales
```

(new col)

we convert to int. else it will concatenate string

cumulative sales

	Total Sales
0	4000
1	9000

4000+5000

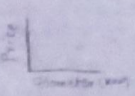
it adds current value to prev. value

```
type(car_sales["Price"][0])   : str
```

(ctrl+/ = comment)

Scatter plot

```
car_sales["Price"] = car_sales["Price"].astype(int)
car_sales.plot(x="Odometer (KM)", y="Price", kind="scatter");
```



Line plot

```
car_sales.plot(x="Sale Date", y="Total Sales");
```

Bar graph

```
x = np.random.rand(10, 4)  # 10 rows, 4 columns
df = pd.DataFrame(x, columns=['a', 'b', 'c', 'd'])
df.plot.bar();   # or: df.plot(kind="bar");
```

```
car_sales.plot(x="Make", y="Odometer (KM)", kind="bar");
```

Hist

```
car_sales["Odometer (KM)"].plot.hist(bins=20);  # plot(kind="hist");
```

default = 10 (need not mention) — basically the no. of columns our histogram is divided into

- Now trying for another dataset (heart_disease → has 14 columns)

```
heart_disease.plot.hist(figsize=(10,30), subplots=True);
```

→ 14 histograms

- Which one should we use? → when plotting something quickly, okay to use pyplot method (.plot())

↓

When plotting something more advanced, use the OO method (object oriented)

Creating a sub frame

```
over_50 = heart_disease[heart_disease["age"] > 50]
```

Pyplot method

```
over_50.plot(kind="scatter", x="age", y="chol", c="target");
```

→ scatterplot

gives us the "scale" Kinda to the scatterplot

```
# Pyplot + OO
fig, ax = plt.subplots(figsize=(10,6))
over_50.plot(kind="scatter", x="age", y="chol", c="target", ax=ax);
ax.set_xlim([45, 100]);   # changes 'x' range to 100
                  low  high
```

If we don't write → ax.set_xlim([50, 75])
If we write → ax.set_xlim([50, 100])

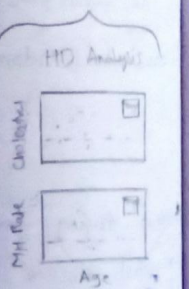
```
# OO from scratch
# Subplot of chol, age, thalach
fig, (ax0, ax1) = plt.subplots(nrows=2, ncols=1, figsize=(10,10), sharex=True)
# add data to ax0
scatter = ax0.scatter(x=over_50["age"], y=over_50["chol"], c=over_50["target"])
# customize ax0
ax0.set(title="Heart Disease Level", ylabel="Cholesterol");
# add a legend to ax0
ax0.legend(*scatter.legend_elements(), title="Target")
# add a meanline
ax0.axhline(y=over_50["chol"].mean(), linestyle='--')
```

(if we didn't write sharex=True, we need to write 'xlabel' of ax0)

(axhline = horizontal line) (y = 'y' value)

```
# add data to ax1
scatter = ax1.scatter(x=over_50["age"], y=over_50["thalach"], c=over_50["target"])
# customize ax1
ax1.set(title="HR and MHR Rate", xlabel="Age", ylabel="MHR Rate");
# add a legend to ax1
ax1.legend(*scatter.legend_elements(), title="Target")
# add a meanline
ax1.axhline(y=over_50["thalach"].mean(), linestyle='--')
# add title to figure
fig.suptitle("HD Analysis", fontsize=16, fontweight="bold")
```

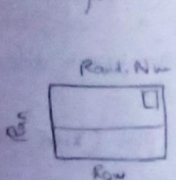
~ Anatomy of matplotlib figure




```
# if we don't need to subplot
fig, ax = plt.subplots(figsize=(10,6))
# rest as same, replace ax0 -> ax
```

Customizing Matplotlib plots & getting stylish:

```
plt.style.available   # shows different styles available
plt.style.use('dark_background')   # updates the plot internally
car_sales["Price"].plot();   # graph plotted with 'dark_background' style
x = np.random.randn(10, 4)   # create some data
df = pd.DataFrame(x, columns=['a', 'b', 'c', 'd'])
ax = df.plot(kind="bar")
ax.set(title="Rand. Num", xlabel="Row", ylabel="Rand")
ax.legend().set_visible(True)   # make legends visible
```



Customizing our full-fledged heart-disease plot!

```
plt.style.use('dark_background')
```

Now a small change: if we want to change 'plotted line/dot' colour as it's

```
scatter = ax.scatter(x=over_50["age"], y=over_50["chol"], c=over_50["target"], cmap="winter")
```

(colour map) (winter = colour name)

Now to save image/graph, we have 2 options:

1. Right click → 'Save image as'
2. In code:

```
fig.savefig("heart-disease-analysis-plot.png")
```

→ gets saved where your jupyter notebook is saved!

Resetting the figure:

```
fig, ax = plt.subplots();
```

Scikit-Learn: Creating Machine Learning Models

In this section, we will learn: → An end-to-end Scikit-Learn workflow

- → Getting data ready (to be used with ML models)
- → Choosing a ML model
- → Fitting a model to the data (learning patterns)
- → Making predictions with a model (using patterns)
- → Evaluating model predictions
- → Improving model predictions
- → Saving & loading models

0. An end-to-end Scikit-Learn workflow:

```
import pandas as pd
heart_disease = pd.read_csv("heart-disease.csv")
x = heart_disease.drop("target", axis=1)   # creating 'x' & 'y'
y = heart_disease["target"]
```

(axis=1 means column) (get data ready)

	age	sex	cp	chol	fbs	...	thal	target
0	63	1	3	233	1		1	1
1	54	0	0	239	1		2	0

```
from sklearn.ensemble import RandomForestClassifier   # choose right model & hyperparameters
clf = RandomForestClassifier()   # clf = classifier
clf.get_params()   # shows its parameters
```

```
from sklearn.model_selection import train_test_split   # splitting model
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
clf.fit(x_train, y_train)
```

splits x, y to training & test data with size 0.2 means 80% - 20%

→ says classification model, random forest, find patterns in training data

y_preds = clf.predict(x_test) → makes a prediction

clf.score(x_train, y_train) → evaluating model → 1's model found patterns in training data

→ returns mean accuracy on given test data & labels

clf.score(x_test, y_test) → shows accuracy on test dataset

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

print(classification_report(y_test, y_preds)) → shows classification metrics

compare test labels to predictions

accuracy_score(y_test, y_preds)

np.random.seed(23)

for i in range(10, 140, 10): → runs 13 times with diff. n_est. parameter & giving diff. accuracy each time

clf = RandomForestClassifier(n_estimators=i).fit(x_train, y_train)

print(f"Accuracy with {i} estimators: {clf.score(x_test, y_test)*100:.2f}%") → upto 2 decimal places in python shorthand

import pickle → save model & load it

pickle.dump(clf, open("rfm_1.pkl", "wb")) → writing binary

saved file name

loaded_model = pickle.load(open("rfm_1.pkl", "rb")) → pickle → read binary

loaded_model.score(x_test, y_test) → scores as per last n_estimator value i.e. =130

1. Getting our data ready to be used with machine learning:

3 main things: → Split data into features (x) & labels (y)

→ Filling (aka imputing) or disregarding missing values

→ Converting non-numerical → numerical (aka feature encoding) (☆)

x = heart_disease.drop("target", axis=1) → removes "target" column

car_sales →

	Make	Colour	Odometer (KM)	Doors	Price
0	Toyota	Red	150000	4	4000
1	Honda	Blue	3240	3	5000

x = car_sales.drop("Price", axis=1)

y = car_sales["Price"]

Now we have to change them into numerical

Turn categories into numbers

from sklearn.preprocessing import OneHotEncoder

from sklearn.compose import ColumnTransformer → Keep it in "cat_fea" as car can have 3,4,5 doors! so just enhancing!

categorical_features = ["Make", "Colour", "Doors"]

one_hot = OneHotEncoder() → instantiating it, a process used to turn categories → numbers

transformer = ColumnTransformer([("one_hot", one_hot, categorical_features)], remainder="passthrough") → accepts tuples with a name

transformed_x = transformer.fit_transform(x) → applies transformation in 'col' the to dataframe 'x'

takes one hot features & applies on categorical features & rest all i.e. remainder = passthrough (don't do anything)

(A)

pd.DataFrame(transformed_x) →

	0	1	2	...	12
0	0.0	1.0	0.0	...	150000.0
1	0.0	0.0	1.0	...	323400.0

Another way to do:

dummies = pd.get_dummies(car_sales[["Make", "Colour", "Doors"]])

dummies →

	Doors	Make_Toyota	Make_Honda	...	Colour_Blue	Colour_Red
0	4	1	0		0	0
1	3	0	1		1	0

from sklearn.ensemble import RandomForestRegressor → regressor: can predict numbers

model = RandomForestRegressor()

np.random.seed(26)

x_train, x_test, y_train, y_test = train_test_split(transformed_x, y, test_size=0.2)

model.fit(x_train, y_train)

model.score(x_test, y_test) → returns co-eff. of determination of the prediction (aka R^2: statistical measure of how well model predicts)

(B)

If missing data is there → imputation / disregarding (☆)

car_sales_missing = pd.read_csv("car-sales-missing-data.csv")

car_sales_missing.isna().sum() → shows how many missing rows there in each col. (is nan)

car_sales_missing["Doors"].value_counts() → checking how many doors of each type

• Fill missing data with Pandas:

car_sales_missing["Make"].fillna("missing", inplace=True) → fills missing values

" ["Colour"]. " "

" ["Odometer (KM)"]. " (car_sales_missing["Odometer"].mean(), inplace=True)

" ["Doors"]. " (4, inplace=True)

x = car_sales_missing.dropna(inplace=True) → removes empty rows (here only rows with "Price" empty will be removed)

car_sales_missing.isna().sum() → shows no. of rows empty for each column

(A)

• Fill missing values with Scikit-Learn

from sklearn.impute import SimpleImputer & NUM. as "mean" & fill categorical values as "missing"

from sklearn.compose import ColumnTransformer

cat_imputer = SimpleImputer(strategy="constant", fill_value="missing")

door_imputer = SimpleImputer(" " " fill_value=4)

num_imputer = SimpleImputer(strategy="mean")

cat_features = ["Make", "Colour"] } defining columns

door_features = ["Doors"]

num_features = ["Odometer"]

imputer = ColumnTransformer([("cat_imputer", cat_imputer, cat_features), ("door_imputer", door_imputer, door_features), ("num_imputer", num_imputer, num_features)]) → takes list of multiple diff. transformers

(creating an imputer) (something that fills the missing data)

filled_x = imputer.fit_transform(x) → transform the data
↳ o/p: [["Toyota", "White", 4.0, 150045.0], [ ], [

car_sales_filled = pd.DataFrame(filled_x, columns = ["Make", "Colour", "Doors", "Odometer"])
car_sales_filled.isna().sum() → shows if any missing value there (here no = 0)

(A)

transformed_x = transformer.fit_transform(car_sales_filled)

np.random.seed(3)
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

(B)

2. Choosing the right estimator/algorithm for your problem:

Within estimators, we have classification models & then within estimators we also have regresses, which is a regression model.

→ Sklearn refers to machine learning models, algorithms as 'estimators'
→ Classification problem - predicting a category (heart disease or not).
Sometimes you'll see 'clf' (short for classifier) used as a classification estimator.
→ Regression problem - predicting a number (selling price of a car)

• Picking a machine learning model for a regression problem:

from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
housing_df = pd.DataFrame(housing["data"], columns = housing["feature_names"])
from sklearn.linear_model import Ridge

x = housing_df.drop("target", axis = 1)
y = housing_df["target"]
} create data
model = Ridge() → instantiating the model

(B)-(x) → (D)

Now what if 'Ridge' didn't work (or) the score didn't fit our needs!
Then trying another model!

→ from sklearn.ensemble import RandomForestRegressor
x = # same as above
y = "
model = RandomForestRegressor()
(based on decision tree algorithm)

(E)-(x)

SciKit-Learn Algo. Cheat-Sheet: "scikit-learn.org/stable/tutorial/machine_learning_map/index.html"

• Picking a machine learning model for a classification problem:

from sklearn.svm import LinearSVC
x = heart_disease.drop("target", axis = 1)
y = heart_disease["target"]
} make data
(F)

clf = LinearSVC() # instantiate LinearSVC
np.random.seed(23) → setup random seed, so their values/output doesn't change
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2) → split data
clf.fit(x_train, y_train)
clf.score(x_test, y_test) → evaluating
↳ returns mean accuracy

(C)

Also, trying from other model

from sklearn.ensemble import RandomForestClassifier → importing this class (RFC)
clf = RandomForestClassifier()
clf.fit(--) → fit model to data (training the ML model)
clf.score(--) → evaluate the RFC (use the patterns that model has learnt)
# rest all same as above

* If you have structured data → use ensemble methods
(ex. table, etc)

* If you have unstructured data, use deep/transfer learning
(ex. images, audio, etc)

3. Fitting algorithms on our data and use it to make predictions

x → features, features variables, data
y → labels, target, target variables

// importing RFC
// making heart-disease data
(C) → clf.fit(--) → fit model to data, training the ML model

(can also be used for regression problems)

Now, there are 2 ways to make predictions: → predict()
→ predict_proba()

• predict()

clf.predict(x_test)
y_preds = clf.predict(x_test) → compare prediction to truth labels to evaluate the model
np.mean(y_test == y_preds) } finding the accuracy (same answer) → from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_preds)

- predict_proba() → returns possibilities of a classification model

clf.predict_proba(x_test[:5]) → o/p: array([[0.89, 0.11], [0.18, 0.82]])
(predicting on the same data) — prob. of '0' — prob. of '1'

clf.predict(x_test[:5]) → o/p: [0,1,1,0,1]

from sklearn.ensemble import RandomForestRegressor

np.random.seed(42)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
model = RandomForestRegressor() creating model instance
model.fit(x_train, y_train)
y_preds = model.predict(x_test) → getting o/p similar to y_test
from sklearn.metrics import mean_absolute_error → comparing predictions to the truth
mean_absolute_error(y_test, y_preds)
o/p: 0.333636
On avg. each one of our models prediction is 0.33 diff. to the actual test value

4. Evaluating a machine learning model

3 ways to evaluate Scikit-learn models/estimators → estimator's built in score() method
- problem specific metric functions
- the scoring parameter

- score() method:

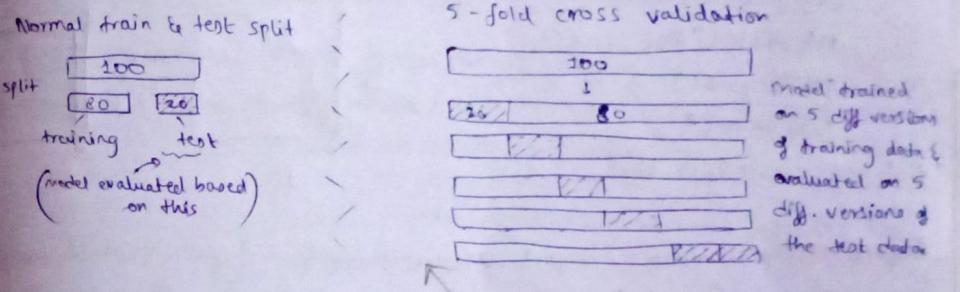
(E) → heart_disease
clf = RandomForestClassifier()
(C)
clf.score(x_train, y_train) → max value = 1.0 & min = 0.0
clf.score(x_test, y_test)
→ returns co-eff. of determination → portion of variation in the dependent variable that is predictable from the independent variable
→ in simple words, returns mean accuracy

(E)
model = RandomForestRegressor()
(C)
model.score(x_test, y_test) → returns R^2 value

- from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
(F)
(E)
clf = RandomForestClassifier()
(C)
cross_val_score(clf, x, y, cv=5) → 5 splits → shows array of 5 values (diff. mean acc.)
scoring parameter of classifier by default - mean acc.
by default scoring = None
If we want, we can import our own scoring methods!

clf_single_score = clf.score(x_test, y_test)
clf_cross_val_score = np.mean(cross_val_score(clf, x, y, cv=5)) → take mean of 5 fold cross validation score
clf_single_score, clf_cross_val_score → comparing both
o/p: 0.80..., 0.82...

- Cross-validation: (cv = (we can write any no.))



- Classification models evaluation metrics:
  - Accuracy
  - Area under ROC curve
  - Confusion Matrix
  - Classification Report

(F)
cross_val_score = cross_val_score(clf, x, y, cv=5)

print(f"Heart_d cross-val. acc: {np.mean(cross_val_score)*100:.2f}%") → upto 2 decimals

- Area under ROC curve:

ROC curves are comparison of a model's true positive rate (tpr) versus model's false positive rate (fpr)
- True (+) = model predicts 1 when truth is 1
- False (+) = model " 1 " truth " 0
- True (-) = model " 0 " truth " 0
- False (-) = model " 0 " truth " 1

from sklearn.metrics import roc_curve
clf.fit(x_train, y_train)
y_probs = clf.predict_proba(x_test) → make predictions with probabilities
y_probs[:5]
o/p: [0.45, 0.55] → prob. of 0, prob. of 1

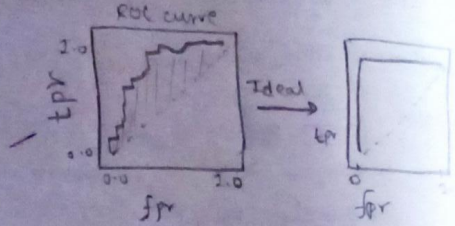
y_probs_positive = y_probs[:, 1] → slicing of column 1

fpr, tpr, thresholds = roc_curve(y_test, y_probs_positive)
↳ takes (y_true, y_

fpr → shows fpr

Creating a func. to plot roc curves

```
import matplotlib.pyplot as plt
def plot_roc_curve(fpr, tpr):
    """
    Plots a ROC curve given fpr & tpr of a model.
    """
    plt.plot(fpr, tpr, color="orange", label="ROC")
    plt.plot([0,1], [0,1], color="darkblue", linestyle="--", label="Guessing")
                ↳ represents x, y co-ord of 2 points on diagonal line
    plt.xlabel("FPR")
    plt.ylabel("TPR")
    plt.title("ROC Curve")
    plt.legend()
    plt.show()
plot_roc_curve(fpr, tpr)
```




```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test, y_probs_positive) → takes test labels & prob.
```
↳ 0.93 → gives area under curve wrt ideal fig!

roc_auc_score(y_test, y_test) → 1 > perfect auc score

ROC & AUC curves are evaluation metrics for binary classification models (a model which predicts one thing or another, such as heart disease or not)

ROC curves compares tpr vs fpr at diff. classification thresholds

The AUC metric tells you how well your model is at choosing between classes.

- Confusion Matrix: quick way to compare the labels a model predicts & the actual labels it was supposed to predict.

In essence, giving you an idea where the model is getting confused.

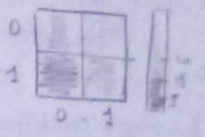
```
from sklearn.metrics import confusion_matrix
y_preds = clf.predict(x_test)
confusion_matrix(y_test, y_preds)
```
(Visualizing) → [[177, 8], [4, 32]] → understanding more clearly


```
pd.crosstab(y_test, y_preds, rownames=["Actual"], colnames=["Predicted"])
```
→ Making our matrix more visual with seaborn's heatmap() → plots rectangular data as a color-encoded matrix

```
import seaborn as sns
sns.set(font_scale=1.5) → setting font scale
# create a confusion matrix
conf_mat = confusion_matrix(y_test, y_preds)
# plotting it
sns.heatmap(conf_mat);
```

Predicted / Actual	0	1
0	177	8
1	4	32

ideal split



Now creating matrix using Scikit-learn → 2 methods!

1)
```
from sklearn.metrics import ConfusionMatrixDisplay
ConfusionMatrixDisplay.from_estimator(estimator=clf, X=x, y=y)
```
Plots confusion matrix given an estimator & data

(estimators make predictions for me without passing it)

2) ConfusionMatrixDisplay.from_predictions(y_true=y_test, y_pred=y_preds)

↳ plots conf. matrix given true & predicted labels

- Classification Report:

It's like report bank of a no. of different parameters evaluating our classification models

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_preds))
```

	precision	recall	f1-score	support
0	0.81	0.68	0.74	25
1	--	--	0.94	36
accuracy			0.8	61
macro avg	--	--	--	--
weighted avg	--	--	--	--

Now precision & recall becomes important when there aren't much balanced classes

(no. of test cases 0-0 & 1-1 → here 25-36 so balanced)
- not like 1 in 60! we need to identify which one patient is '1'!

★ Classification Model Evaluation Metrics/Techniques:

- Accuracy: good measure to start if classes are balanced. Perfect accuracy is 1.0.
- Recall: indicates the proportion of actual positives which were correctly classified. Model which produces no false negatives has a recall of 1.0
- Precision: indicates the proportion of +ve identifications (model predicted class 1) which were actually correct. Model with no false positives has precision of 1.
- F1 score: combination of precision & recall. (max value = 1.0)
- Support: no. of samples each metric was calculated on
- Macro avg: the avg. precision, recall & f1-score between classes. If you have class imbalances, pay attention to this metric.
- Weighted avg: the weighted avg precision, recall & f1 score between classes. Weighted means each metric is calculated wrt how many samples there are in each class.
- Confusion Matrix: compares predicted values with true values in a tabular way. If 100% correct, all values in the matrix will be top left to bottom right (diagonal line)
- Cross-Validation: splits your dataset into multiple parts & train & tests your model on each part then evaluates performance as an avg.
- Classification Report: returns some of the main classification metrics such as precision, recall & f1-score.
- ROC curve: aka "receiver operating characteristic" is a plot of true positive rate vs false-positive rate
- Area Under Curve (AUC) score: The area underneath the ROC curve. A perfect model achieves an AUC score of 1.0.

★ What classification metric should you use?

- Accuracy is a good measure to start if all classes are balanced.
- Precision & recall become more imp. when classes are imbalanced.
- If false-positive predictions are worse than false-negatives, aim for high precision.
- If false-negative predictions are worse than false-positives, aim for higher recall.
- F1 score is a combination of precision & recall
- A confusion matrix is always a good way to visualize how a classification model is going.

- Regression model evaluation metrics:
  - R^2/Co-eff. of determination
  - Mean Absolute error (MAE)
  - Mean Squared error (MSE)

- R^2: it compares my models prediction to the mean of the targets. Values can range from -ve ∞ (a very poor model) to 1.
  For ex: if all my model does is predict the mean of the targets, it's r^2 value = 0. If my model predicts a range of nos, r^2 = 1

```
from sklearn.metrics import r2_score
r2_score(y_true = y_test, y_pred = y_test)  ↳ op: 1.0
```

- Mean Absolute Error (MAE): it's avg of the absolute differences between predictions and actual values.
  It gives you an idea of how wrong your model predictions are!

```
from sklearn.metrics import mean_absolute_error
y_preds = model.predict(x_test)
mae = mean_absolute_error(y_test, y_preds)
mae  ↳ 0.33
```

MAE turns -ve to +ve

```
df = pd.DataFrame(data={"actual values": y_test, "predicted values": y_preds})
df["differences"] = df["predicted values"] - df["actual values"]
df.head(5)
```

	actual values	predicted values	differences
4924	4.700	4.312	-0.34 ...
1369	0.81	0.95	0.13

```
df["differences"].mean()  ↳ -0.02
np.abs(df["differences"]).mean()  ↳ 0.33
```

getting diff from mae as -ve didn't change to +ve

- Mean Squared Error (MSE): MSE is the mean of the square of the errors btw actual & predicted values

```
from sklearn.metrics import mean_squared_error
y_preds = Model.predict(x_test)
mse = mean_absolute_error(y_test, y_preds)
mse → 0.33
df["squared differences"] = np.square(df["differences"])
df.head()
```
↳ now here diff. is in decimals but it actually represents real world values so you square large no. → large error!

★ Regression Model Evaluation Metrics/Techniques:

- R^2/co-eff. of determination
- MAE
- MSE

★ Which regression metric should you use?

- R^2 is similar to accuracy. It gives you a quick indication of how well your model might be doing. Generally the closer your R^2 value is to 1.0, the better the model.
  But it doesn't really tell exactly how wrong your model is in terms of how far off each prediction is.
- MAE gives a better indication of how far off each of your model's prediction are on average.
- MSE amplifies larger differences.

• Scoring () parameter:

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
np.random.seed(32)
```
(E)
```
clf = RandomForestClassifier()
cv_acc = cross_val_score(clf, x, y, cv=5, scoring=None)
cv_acc, np.mean(cv_acc)*100:.2f
```
↳ returns array with 5 values

↳ (cross-validated accuracy)

↳ mean we are using estimator's default scoring evaluation metric i.e. "accuracy" for classification models

```
cv_acc = cross_val_score(clf, x, y, cv=5, scoring="accuracy")
cv_acc, np.mean(cv_acc)*100:.2f
```
↳ returns 5 values ↳ cross-validated accuracy } op of above & this one is same!

```
cv_precision = cross_val_score(clf, x, y, cv=5, scoring="precision")
cv_precision, np.mean(cv_precision)*100:.2f
```
↳ returns 5 values ↳ cross-validated precision

```
cv_recall = cross_val_score(clf, x, y, cv=5, scoring="recall")
cv_recall, np.mean(cv_recall)*100:2f
```
↳ returns 5 values ↳ cross-validated recall

Now using it for regression problem

```
from — import —
```
(D)
```
model = RandomForestRegressor()
cv_r2 = cross_val_score(model, x, y, cv=5, scoring=None)
np.mean(cv_r2) → op: 0.653
```
Mean squared error
```
cv_mse = cross_val_score(model, x, y, cv=5, scoring="neg_mean_squared_error")
np.mean(cv_mse) → op: -0.430
```
mean absolute error
```
cv_mae = cross_val_score(model, x, y, cv=5, scoring="neg_mean_absolute_error")
np.mean(cv_mae) → op: -0.468
```

• Diff. evaluation metrics as Scikit-learn functions:

```
- from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
  " " .ensemble " RandomForestClassifier
  " " .model_selection " train_test_split
```
(F)
```
clf.fit(x_train, y_train)
y_preds = clf.predict(x_test)
```
making predictions

evaluating model using evaluate function
```
print("Classifier metrics on test set")
print(f"Accuracy : {accuracy_score(y_test, y_preds)}") → 0.895
print(f"Precision : {precision_score(y_test, y_preds)}") → 0.842
print(f"Recall : {recall_score(y_test, y_preds)}") → 0.969
print(f"F1 : {f1_score(y_test, y_preds)}") → 0.901
```

- from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
  " " .ensemble " RandomForestRegressor
  " " .model_selection " train_test_split

np.random.seed(42)

(1)

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

model = RandomForestRegressor()

model.fit(x_train, y_train)

y_preds = model.predict(x_test)

print("Regressor metrics on the test set")

print(f"R2 score: {r2_score(y_test, y_preds)}") → 0.822

print(f"MAE: {mean_absolute_error(y_test, y_preds)}") → 0.321

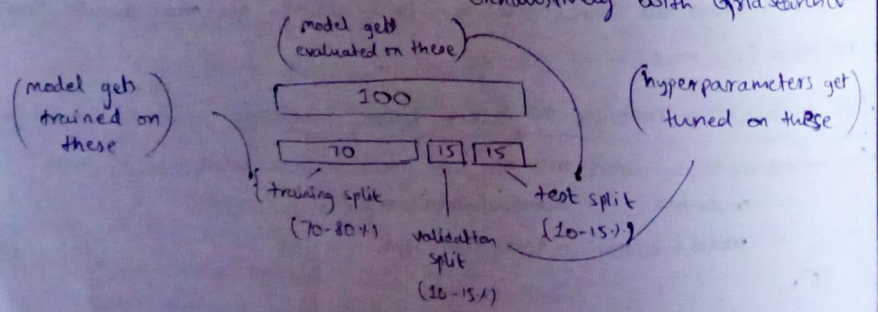
print(f"MSE: {mean_squared_error(y_test, y_preds)}") → 0.236

5.0 Improving a model

Hyperparameters vs Parameters
↳ model find these pattern in data

↳ Settings on a model you can adjust to (potentially) improve its ability to find patterns

3 ways to adjust hyperparameters → by hand
→ randomly with RandomSearchCV
↳ exhaustively with GridSearchCV



from ____ import RandomForestClassifier

clf = Ra_f_Cl_()

clf.get_params() → gets parameters for this estimator

will try & adjust → max_depth, max_features, min_samples_leaf, min_samples_split, n_estimators

Tuning hyperparameters by hand

```
def evaluate_preds(y_true, y_preds):
    """
    Performs evaluation comparison on y_true labels vs y_preds labels on a classification.
    """
    accuracy = accuracy_score(y_true, y_preds)
    precision = precision_score(y_true, y_preds)
    recall = recall_score(y_true, y_preds)
    f1 = f1_score(y_true, y_preds)
    metric_dict = {"accuracy": round(accuracy, 2),
                   "precision": round(precision, 2),
                   "recall": round(recall, 2),
                   "f1": round(recall, 2)}
    print(f"Acc: {accuracy * 100:.2f}%")
    print(f"Precision: {precision:.2f}")
    print(f"Recall: {recall:.2f}")
    print(f"F1 score: {f1:.2f}")
    return metric_dict
```

```
from sklearn.ensemble import RandomForestClassifier
np.random.seed(33)
heart_disease = pd.read_csv("heart-disease.csv")
heart_disease_shuffled = heart_disease.sample(frac=1)   # shuffling data; means 100% of data
x = heart_disease_shuffled.drop("target", axis=1)
y = heart_disease_shuffled["target"]                    # split into 'x' & 'y'
train_split = round(0.7 * len(heart_disease_shuffled))  # 70% data
valid_split = round(train_split + 0.15 * len(heart_disease_shuffled))  # split into train, validation, test sets; 15% of data
x_train, y_train = x[:train_split], y[:train_split]
x_valid, y_valid = x[train_split:valid_split], y[train_split:valid_split]
x_test, y_test = x[valid_split:], y[valid_split:]
clf = RandomForestClassifier()   # instantiate classifier with baseline hyperparameters
clf.fit(x_train, y_train)
y_preds = clf.predict(x_valid)   # make baseline predictions
baseline_metrics = evaluate_preds(y_valid, y_preds)   # evaluate on validation sets
```

↳ op: Acc: 75.56%
Precision: 0.83
Recall: 0.74
F1 score: 0.78

Training it on diff dataset

```
clf_2 = RFC(max_depth=10)          # ran for 10 mins
clf_2.fit(x_train, y_train)
y_preds_2 = clf_2.predict(x_valid)
clf_2_metrics = evaluate_preds(y_valid, y_preds_2)
```

→ Acc: 82.22%
Precision: 0.89
Recall: 0.81
F1 score: 0.85

You notice that searching each parameter & writing values for it, is this method not preferable.

• Hyperparameter tuning with RandomizedSearchCV:

```
from sklearn.model_selection import RandomizedSearchCV
grid = {"n_estimators": [10, 100, 200, 500, 1000, 1200],
        "max_depth": [None, 5, 10, 20, 30],
        "max_features": ["auto", "sqrt"],
        "min_samples_split": [2, 4, 6],
        "min_samples_leaf": [1, 2, 4]}
```

in form of dictionary

```
np.random.seed(23)
```

(G)

(H)

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
clf = RandomForestClassifier(n_jobs=1)
```

→ dictate how much parallel processing to run during training & prediction

```
rs_clf = RandomizedSearchCV(estimator=clf,
                            param_distributions=grid,
                            n_iter=10,
                            cv=5,
                            verbose=2)
```

→ no. of models to try

RSCV → chooses any random value among above hyp-param & it does this 10 times

```
rs_clf.fit(x_train, y_train)
```

↳ Fitting 5 folds for each of 10 candidates, totalling 50 fits → shows o/p

```
rs_clf.best_params_
```

↳ checking best o/p among above 50

```
rs_y_preds = rs_clf.predict(x_test)
```

→ make predictions with best hyp-params

↳ last value called of rs_clf is chosen → (means calling hyp indirectly)

```
# evaluate the predictions
rs_metrics = evaluate_preds(y_test, rs_y_preds)
```

→ Acc: 83.61%
Precision: 0.89
Recall: 0.83
F1 score: 0.85

• Hyperparameter tuning with GridSearchCV

In this, we will go through each combination of custom hyp-param given

```
grid_2 = {"n_estimators": [100, 200, 500],
          "max_depth": [None], "max_features": ["auto", "sqrt"],
          "min_samples_split": [6], "min_samples_leaf": [1, 2]}
```

total combo it will go through = 3×1×2×1×1×2 = 12

```
from sklearn.model_selection import GridSearchCV
np.random.seed(23)
```

(G)

(H)

```
gs_clf = GridSearchCV(estimator=clf,
                      param_grid=grid_2,
                      cv=5,
                      verbose=2)
```

→ helps in running smoothly

```
gs_clf.fit(x_train, y_train)
```

→ Fitting 5 folds for each of 12 candidates, totalling 60 fits

```
gs_clf.best_params_
```

→ best hyp-param

```
gs_y_preds = gs_clf.predict(x_test)
```

→ making prediction

```
gs_metrics = evaluate_preds(y_test, gs_y_preds)
```

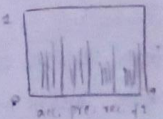
→ Acc: 83.61%
Precision: 0.86
Recall: 0.86
F1 score: 0.86

• Comparing different models:

Note: Data was split differently for diff. models, so the comparisons aren't fully correct.

```
compare_metrics = pd.DataFrame({"baseline": baseline_metrics,
                                "clf_2": clf_2_metrics,
                                "random search": rs_metrics,
                                "grid search": gs_metrics})
compare_metrics.plot.bar(figsize=(5, 4))
```

→ plots graph



6. Saving and loading a trained machine learning models

2 ways to save & load → Python's 'pickle' module / 'joblib' module

• import pickle

```
# save existing file to model
pickle.dump(gs_clf, open("gs_random_forest_model_1.pkl", "wb"))
```

saved filename; write binary; → pickle

```
loaded_pickle_model = pickle.load(open("gs_random_forest_model_1.pkl", "rb"))
# make some predictions
pickle_y_preds = loaded_pickle_model.predict(x_test)

evaluate_preds(y_test, pickle_y_preds)
```

read binary

- from joblib import dump, load
- # save model to file
dump(gs_clf, filename="gs_random_forest_model_1.joblib")
- # import a saved joblib file
loaded_joblib_model = load(filename="gs_random_forest_model_1.joblib")
- # make & evaluate joblib predictions
joblib_y_preds = loaded_joblib_model.predict(x_test)
evaluate_preds(y_test, joblib_y_preds)

If model is large, its better to use joblib to save & load.

7. Putting it all together

data = pd.read_csv("car-sales-extended-missing-data.csv") → has missing values too

	Make	Colour	Odometer (km)	Doors	Price
0					
1					

Things to remember
- All data should be numerical
- there should be no missing values
- Manipulate the test set the same as the training set
- Never test on data you've trained on
- Tune hyperparameters on validation set OR use cross-validation
- One best performance metric doesn't mean the best model.

- Getting data ready

```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
```

Modelling

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, GridSearchCV
```

Setup random seed

```
import numpy as np
np.random.seed(23)
```

Import data & drop rows with missing labels

```
data = pd.read_csv("car-sales-extended-missing-data.csv")
data.dropna(subset=["Price"], inplace=True) → drop rows with missing "Price" value
```

Define different features & transformer pipeline

```
categorical_features = ["Make", "Colour"]
categorical_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="constant", fill_value="missing")),
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])
```
→ replaces/imputes missing values in 'categorical features' with constant value 'missing' & then applies one-hot encoding

```
door_feature = ["Doors"]
door_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="constant", fill_value=4))
])
numeric_features = ["Odometer (km)"]
numeric_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="mean"))
])
```

Setup preprocessing steps (fill missing values, then convert to no.'s)

```
preprocessor = ColumnTransformer(transformers=[
    ("cat", categorical_transformer, categorical_features),
    ("door", door_transformer, door_feature),
    ("num", numeric_transformer, numeric_features)
])
```
Transformer for Door Feature ('door') - applies the door_transformer to column door_feature

Creating a preprocessing and modelling pipeline

```
model = Pipeline(steps=[("preprocessor", preprocessor),
                        ("model", RandomForestRegressor())])
```
→ applies 'preprocessor' to input data

Split data

```
x = data.drop("Price", axis=1)
y = data["Price"]
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
```

Fit & score our model

```
model.fit(x_train, x_test)
model.score(x_test, y_test)
```

```
#We can also use 'GridSearchCV' or 'RandomizedSearchCV' with our Pipeline
from sklearn.model_selection import GridSearchCV
pipe_grid = {
    "preprocessor__num__imputer__strategy": ["mean", "median"],
    "model__n_estimators": [100, 1000],
    "model__max_depth": [None, 5],
    "model__max_features": ["auto"],
    "model__min_samples_split": [2, 4]
}
```
means in 'preprocessor', go to 'num', go to 'imputer', go to 'strategy' and check 'mean' & 'median'

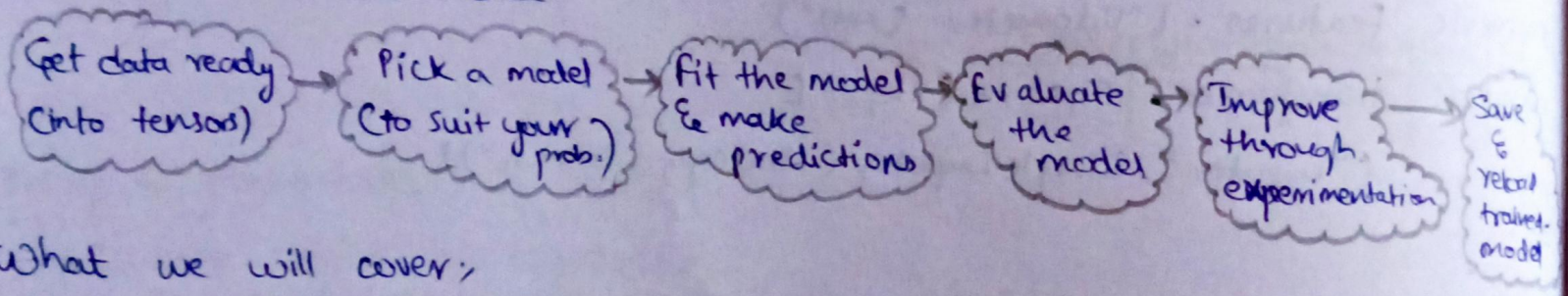
```
gs_model = GridSearchCV(model, pipe_grid, cv=5, verbose=2)
gs_model.fit(x_train, y_train)
gs_model.score(x_test, y_test)
```

Why use `Pipeline`? → reusability
→ consistent application
→ code organization
→ easy hyperparameter tuning.

Note: If Scikit-Learn version > 1.2 ; then "plot_roc_curve" will show error
so, use "RocCurveDisplay" syntax!

Neural Networks : Deep Learning, Transfer Learning & Tensorflow 2

A tensorflow workflow:



What we will cover:

→ An end-to-end multi-class classification workflow with Tensorflow
→ Preprocessing image data (getting it into Tensors)
→ Choosing a deep learning model → fitting a model to the data. (learning patterns)
→ Making predictions with a model (using patterns)
→ Evaluating model predictions → Saving & loading models
→ Using a trained model to make predictions on custom data