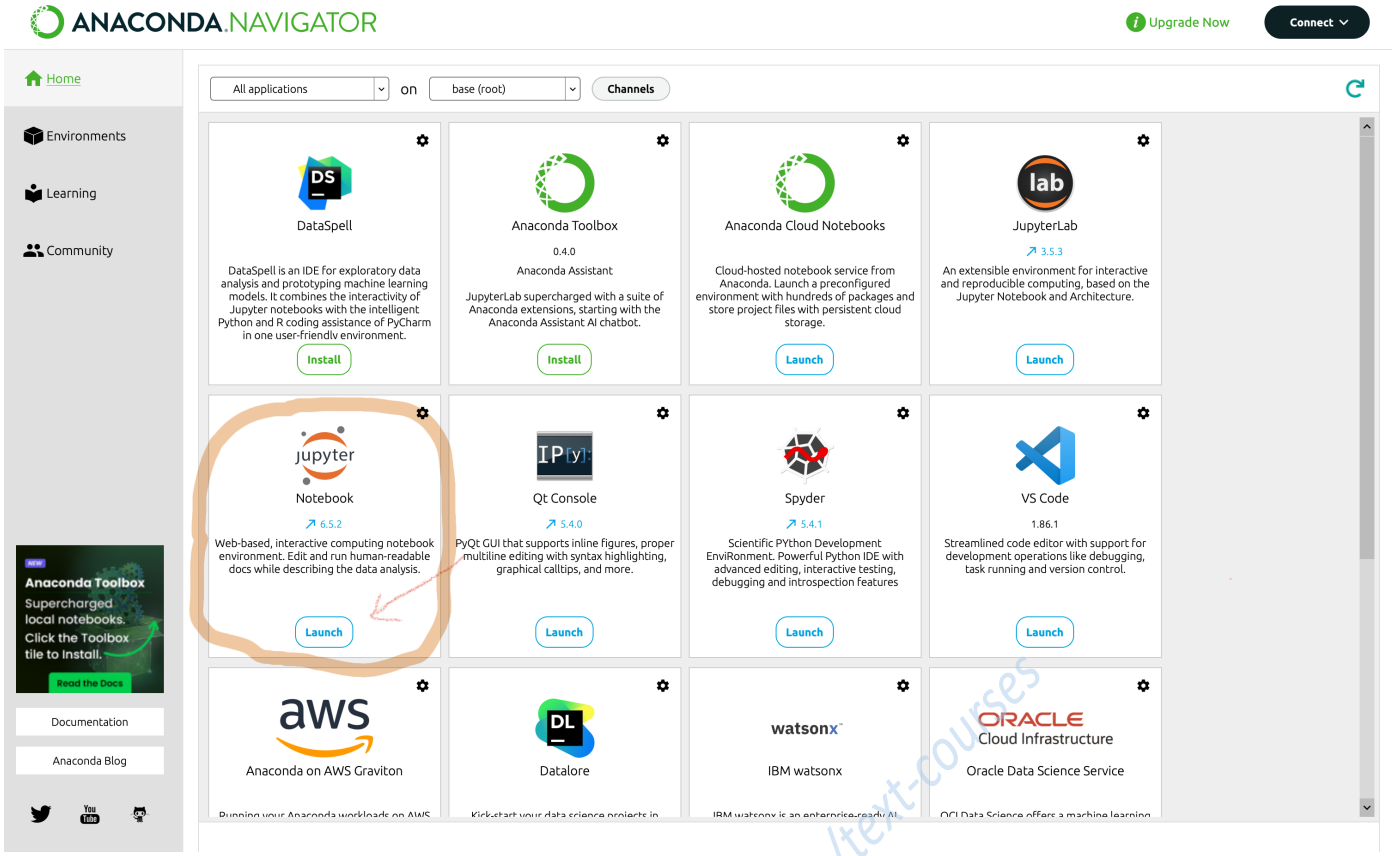
Setting up your machine

we will start our discussion from scratch , right at installing the setup required to run python programs on your computer. If you have got that ready on your machine already , you can skip the first section and move to next small discussion on markdown text in jupyter notebooks.

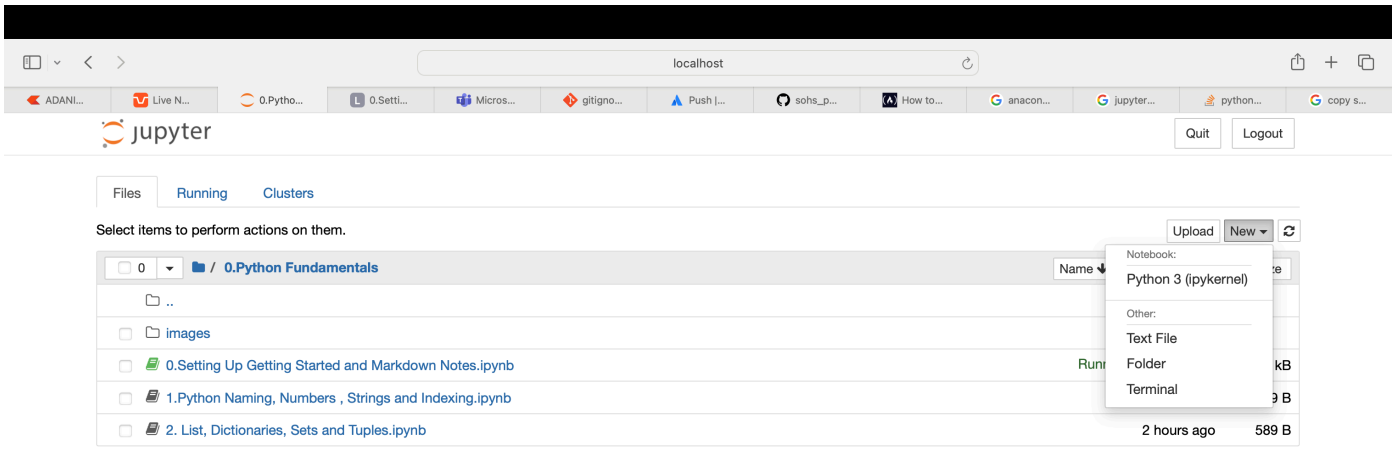
Download Anaconda , Install and launch

- Google "download anaconda", and go to the first link that you get and download and install for the OS that you are are running on your laptop/PC
- Anaconda is a python distribution which comes bundled with lots of essentials for ML and we don't have to worry about package version clashes .
- After installation , search for "anaconda navigator" and start it , this is what the logo looks like.

- once you launch this , you will get a navigator with multiple applications in them. What we want to run is "jupyter notebook" . Don't worry about other applications shown here in the image . click on launch for jupyter notebook. The first one in the second row application here in the image . The location of this in the navigator on your machine can be different, but the logos will match.



- This will launch a browser window [it might also ask you to select your preferred browser for the same, also you don't need to be connected to internet for this] which is folder browser. Browse to the location where you want to create python files for this session . Files and Folder shown here, they exist physically on your machine . So you can create those folders and then browse to those location here . Once you have reached to that location click on the right top option called new and select python 3, this will create a python notebook, where we will write our code and comments and notes .



- By default the notebook will be named "Untitled", click on that name [on top, left] and give it a meaningful name. this will have a cell [appears like a one line bar with empty square brackets on the left side]. This cell is actually where you will write your code/programs . Click on it and you will see your cursor appear in the box , you can start typing there. here are some useful keyboard shortcuts
    - Execute program in a cell -> shift + enter
    - Add cell above or below the selected cell -> escape + a/b
    - delete a cell -> escape + d + d
    - undo deletion -> escape + z
    - convert a code to cell to markdown cell -> escape + m
    - convert a markdown cell to code cell -> escape + y
- to open existing notebook, just click on them and they will open in the browser [will have extension .ipynb]
- whats a markdown cell ? -> a markdown cell is where you can write formatted text using some fixed symbols which make the text become a heading or bold or italic or bullet points and so on. you can make use of this to make better notes

markdown symbols

`#` : h1 heading . write # and then after a space write your heading

`##` : h2 heading . as you increasing number of hashes , the heading will become smaller and smaller .

starting a line with `*` or `-` , makes that a bulleted line . Hit a tab to make nested bullets.

surrounding a text with `**` will make it bold . `**some text**` -> **some text**

surrounding a text with `*` will make it italic . `*italic text*` -> *italic text*

if you use latex for math symbols. that also works. `$y_i=\beta_0+\beta_1*x_1 \cdots$` -> $y_i = \beta_0 + \beta_1 * x_1 \cdots$. If you are not aware of latex, don't worry about it for now, has nothing to do with our journey of learning python and ML

you can edit your markdown notes , just like the code cell [double click on the formatted text that you see and it will reveal the markdown cell] and to see resulting formatted text , just do : shift+enter

```
# you can also write free english comments in the code cell itself by starting those lines with symbol hash [#]
# python will not try to execute those lines as code
# shortcut for converting code lines to comments : select lines + ctrl + /
```

Lets Get Started

to start working with python, you will need to store your information in python objects . . you chose a name for an object and then using `=` symbol assign it what ever value you want it to contain . it doesn't really matter whether there is space between the `=` symbol and value or the name. with spaces though , things look neater. There are some simple rules to follow when naming your objects .

Python Naming Rules For Objects

- No special characters are allowed in the names except underscore `_`
- Names should not start with numbers , numbers can appear anywhere else in the name of an object
- You can not have spaces in the name of an object

if you violate these rules , python will simply throw an error and will not create those objects . some examples below . run those cells and see the errors for yourself [shift+enter]. try to make changes in the object name as per the rules mentioned above and see the error go away .

```
1obj = 4  # if you have to have that number in the name , how about you move it to the end of the name
```

```
  Cell In[1], line 1
    1obj = 4  # if you have to have that number in the name , how about you move it to the end of the name
    ^
SyntaxError: invalid decimal literal
```

```
$obj = 4 # remove the symbol $
```

```
  Cell In[2], line 1
    $obj = 4 # remove the symbol $
    ^
SyntaxError: invalid syntax
```

```
this is my obj=4 # if you do want your object name to be verbose/descriptive use _ where you have spaces
```

```
  Cell In[3], line 1
    this is my obj=4 # if you do want your object name to be verbose/descriptive use _ where you have spaces
               ^
SyntaxError: invalid syntax
```

Now that we have figured out how to store values in some python object , let's see how to operate on those values starting with numeric operations .

Numbers and Numeric Operations

```
x = 4
y = 3
```

you can see what is stored in your object , by writing just the object name and executing that cell

```
x
```

```
4
```

```
y
```

```
3
```

addition ,subtraction, multiplication , division

all the usual symbols work for these operations

```
x+y
```

```
7
```

```
x-y
```

```
1
```

```
x*y
```

```
12
```

```
x/y
```

```
1.3333333333333333
```

if you want to store these results , you can simply assign these expressions themselves directly to another object . Notice that when you assign the results of an operation to an object as we are doing below , the outcome is not displayed as earlier . If you want to see the outcome you can always type the name of the object that you assigned your result to and execute that to see the value of the outcome stored in that object.

```
a = (x+y)/(x-y)
```

```
a
```

```
7.0
```

Exponentiation

for calculating a^b kind of expression , you need to use `**` for powers or exponents . since in our case here `x=4` and `y=3` . `x**y` would give result for 4^3

```
x**y
```

```
64
```

Special Math Functions

if you need to use special math operation using mathematical functions like sin, cos , log , exp etc; you can make use of package math . here is how to do it

- import package math , assign it a `nick-name` [or `alias` to it]
- use that alias to access the functions it contains
- inputs to a function are passed using `( )` , inputs go inside these `( )`, separated by comma if multiple inputs are required.

```
import math as m
```

in the above cell we assign alias `m` to package `math`. So here onwards , we can use `m` instead of writing the whole word `math` to refer to the package math. Thats how aliases work .

```
m.exp(4)
```

```
54.598150033144236
```

```
m.log(m.exp(4))
```

```
4.0
```

```
m.log10(10**35)
```

```
35.0
```

```
m.sin(2*m.pi) # you will get a number in scentific notation, very close to zero
```

```
-2.4492935982947064e-16
```

Strings

in programming language words or text data is referred to as strings [possibly originating from referring to words as strings of characters] . to pass strings as information to python objects , we pass them in single, double or triple quotes [python is indifferent to any variant and considers all of them as strings]

```
mystring = 'PytHON MaChIne'
```

everything inside the quotes is considered a character. everything ; a white space, a number , a special character, an alphabet, everything which appears in the quote is treated as a character and counted as one . lets look at some string functions .

length of string [number of characters]

```
len(mystring)
```

```
14
```

```
Note: Although we are using function len here to get number of characters in a string,
you will later come to know that len is a generic function which returns number of
elements in an iterable or iterable component of an object.In case of strings,
characters in the string are its elements and len returns number of those elements
amounting to number of characters in the string
```

some more functions

```
mystring.lower()
```

```
'python machine'
```

```
mystring.upper()
```

```
  'PYTHON MACHINE'
```

```
mystring.capitalize()
```

```
'Python Machine'
```

```
mystring
```

```
'PytHON MaChIne'
```

note that , none of the above functions have changed the original string object. they simply gave you an outcome of the change which those functions do. You can make those changes stick by simply assigning the outcomes back to the original object .

```
mystring=mystring.upper()
```

```
mystring
```

```
'PYTHON MACHINE'
```


Note that now the object value has been altered when you assigned the outcome of the function back to the same object, thus overwriting the original value.

Indexing

Iterable is an object which has elements in it that we can iterate over.

in python almost every Iterable is indexed [with few exception, but for now we don't need to worry about them] . Iterable is something which has multiple elements and we can access them individually or as a subgroup. String is our first encounter with an Iterable .The multiple elements of a string are its individual characters .

indexing is done internally in python like this.

- from left to right; that is default and starts with 0
- from right to left , starts with -1

```
mystring='HEALTH SCIENCES'
```

we are going to use slightly longer string for this section. don't worry about the for loop and other syntax here , this simply tells you index of each character in the string

```
for index,char in enumerate(mystring):
    print(char, index)
```

```
H 0
E 1
A 2
L 3
T 4
H 5
  6
S 7
C 8
I 9
E 10
N 11
C 12
E 13
S 14
```

```
for index,char in enumerate(mystring):
    print(char, -len(mystring)+index)
```

```
H -15
E -14
A -13
L -12
T -11
H -10
  -9
S -8
C -7
I -6
E -5
N -4
C -3
E -2
S -1
```

we can use these indices to access individual characters in the string. python uses `[ ]` for passing positions, conditions, indices etc to an iterable. execute these and try to guess before hand what the outcome will be. if you pass an index which exceeds the length of the string, you will get an error

```
mystring[-10]
```

```
'H'
```

```
mystring[0]
```

```
'H'
```

```
mystring[2]
```

```
'A'
```

Below code will throw error, because we are trying to pass index which does not exist in the string because number of characters in the string are simply not as much as the index being passed here.

```
mystring[20]
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

Cell In[33], line 1
----> 1 mystring[20]
```

```
IndexError: string index out of range
```


Accessing a group , passing indices for starting , stopping and step size

you can pass something on the lines of mystring[start:stop:step] . python's behaviour is little peculiar here , the last index or stop index value is not included . if you want to read up on discussion on possible rationale for this , go here : https://stackoverflow.com/questions/11364533/why-are-slice-and-range-upper-bound-exclusive

but thats just to satiate your curiosity. you can continue here without visiting that link and making peace with the fact that the last index is not included in the outcome .

For following examples it will help if you write the string 'HEALTH SCIENCES' on a paper with index of words on top of them.

in the example below outcome will start at index 1 value , whichs E and go till index 4 only

```
mystring[1:5]
```

```
'EALT'
```

```
mystring[1:10]
```

```
'EALTH SCI'
```

by default : the step size is 1 and string traversal happens from left to right . you dont have to explicitly mention that. But lets say you want to change the step size to 2, meaning traversal will happen in a fashion that next character will be 2 step away instead of the immediate next one

```
mystring[1:10:2]
```

```
  'ELHSI'
```

you can reverse the traversal direction by setting the step size as -1

```
mystring[1:10:-1]
```

```
''
```

the outcome that you see is empty because you cant go from index 1 to 10 when traversing from right to left, hence the empty string. you will have to pass appropriate starting and ending posisitions as well

```
mystring[5:2:-1]
```

```
'HTL'
```


if you leave the starting position empty , python assumes that starting position is at the very beggining of the traversal and if you leave the ending position empty, python assumes that stopping positions is at the very end of the traversal . here are some examples . try to guess what the outcome would be before you execute and if the outcome is different from what you expected ,

```
mystring[::-1]
```

```
'SECNEICS HTLAEH'
```

Couple more examples for you to practice and align your understanding.

```
mystring[:-14:-2]
```

```
'SCEC TA'
```

```
mystring[2::2]
```

```
'AT CECS'
```

the same idea of indexing will apply to other ordered iterables also, like lists , arrays, series etc that we will encounter as we progress in the book.

Now we will move on to object types in python which can store more than one value and come with a lot of functionalities enabling us to do much more with python.

List, Dictionary, Set and Tuples

These ideas in python enable us to store multiple values in a single object. Many other complex data structures that we will encounter later in python derive from these . Most fundamental here is a list .

List

A list can contain anything as element, including another list itself . Its created by passing elements in a `[ ]` separated by `,` . There is no restriction on what can be an element of a list , they dont all have to be of the same type either .

```
x=[2,4,99,"python","a",0,-99]
```

indexing of lists work just like string . Just that now an index is assigned to an element one by one , instead of characters of the string. try to guess the outcomes of the following .

```
a=x[-4]
```

```
a
```

```
'python'
```

since this happens to be a string we can further subset it the way we were doing earlier . [Note that further subsetting depends on if the element is subsettable, for example , this would not make sense if this was an integer instead]

```
a[2]
```

```
't'
```

We can do this subsetting in one go as well, instead of assigning `x[4]` to an intermediate object first.

```
x[-4][2]
```

```
't'
```

Lets look at some more ways in which indexing is same as strings for lists otherwise

```
x[2]
```

```
99
```

```
x[:4]
```

```
[2, 4, 99, 'python']
```

```
x[:2:-1]
```

```
[-99, 0, 'a', 'python']
```


adding elements to a list

We have seen how to create a list , in following few sections we will see how to add more elements to an existing list.

adding another list

```
x=x+[3,4]
x
```

```
[2, 4, 99, 'python', 'a', 0, -99, 3, 4]
```

there are more efficient ways to do it instead of overwriting objects like this, lets look at functions specifically built for adding elements to a lisy in python

append and extend for single elements

there is no difference in the outcome , if you consider single elements for both append and extend . but these functions make in-place changes instead of explicitly showing the results . Meaning you will not see an explict outcome when you run them, but the object itself would have changed internally.

```
x.append(3)
```

```
x
```

```
[2, 4, 99, 'python', 'a', 0, -99, 3]
```

since `extend` only works with iterables, we will need to make single number `33` an iterable by making it a list with a single element in it by writing it as `[33]`, you can try without the square brackets and note the error that you get here in the accompanying notebook.

```
x.extend([33])
x
```

```
[2, 4, 99, 'python', 'a', 0, -99, 3, 33]
```

append and extend for iterables

when it comes to iterables like another list or string for example; append adds that to the list as is, where as extend extracts each element of the iterable being added and adds elements individually

```
x.append([78,99,0])
x
```

```
[2, 4, 99, 'python', 'a', 0, -99, 3, 33, [78, 99, 0]]
```

```
x[-1]
```

```
[78, 99, 0]
```



```
x.extend([78,99,0])
x
```

```
[2, 4, 99, 'python', 'a', 0, -99, 3, 33, [78, 99, 0], 78, 99, 0]
```

Replacing items in a list

```
x
```

```
[2, 4, 99, 'python', 'a', 0, -99, 3, 33, [78, 99, 0], 78, 99, 0]
```

Code below will replace the index `3` element `python` with the value `-100` .

```
x[3]=-100
x
```

```
[2, 4, 99, -100, 'a', 0, -99, 3, 33, [78, 99, 0], 78, 99, 0]
```

Removing elements from the lists

There are multiple functions to do so with their own quirks and specifications . we will start with function remove.

remove

Like other few functions that we have seen for lists, this one also makes inplace changes . It starts removing one by one , by value , starting at the first occurence of the value. Meaning, if there are multiple occurences of a value in the list, in one run, it will only remove the first occurence that i finds and will leave the rest.

function will throw an error if the value that you are trying to remove from the list, does not exist in the list.

```
x
```

```
[2, 4, 99, -100, 'a', 0, -99, 3, 33, [78, 99, 0], 78, 99, 0]
```

```
x.remove(99)
x
```

```
[2, 4, -100, 'a', 0, -99, 3, 33, [78, 99, 0], 78, 99, 0]
```


pop

by default it removes the last value and displays the removed value as outcome. You can also pass a particular positions

```
x.pop()
```

```
0
```

```
x
```

```
[2, 4, -100, 'a', 0, -99, 3, 33, [78, 99, 0], 78, 99]
```

```
x.pop(2)
```

```
-100
```

```
x
```

```
[2, 4, 'a', 0, -99, 3, 33, [78, 99, 0], 78, 99]
```

Sorting a list

keep in mind that a list is sortable only if it contains elements which can all be meaningfully compared. so a list containing mix of say numbers and string can not be sorted . Lets create a list with all numeric elements. Note that sort also makes inplace changes and does not display any output.

```
x=[4,89,0,-10,4,22]
```

```
x.sort()
x
```

```
[-10, 0, 4, 4, 22, 89]
```

```
x.reverse()
x
```

```
[89, 22, 4, 4, 0, -10]
```

try creating a list with mixed kind elements and then try to sort, see what the error say

Creating an empty list

This can be usefule when you start a program which will generate some values that you want to store. you can do all the operations like adding , extending etc on this empty list.

```
x=[]
```

Dictionaries

dictionaries are a kind of data structures where you can access items with a key instead of their positions. this is much quicker than searching through a list. In general this serves many other purposes too. in order to create a dictionary we use `{ }` where `key:value` pairs are separated by `,` .

- anything can be a key , except lists and dictionaries [more precise statement is that any mutable object can not be a key , but thats a technical detail we can skip for now]
- keys need to be unique

- there is no restriction on values

```
mydict= {'city':'delhi',2:'pin', 'num':[3,4,5,6],'pin':2}
```

accessing elements in dictionry

you access values stored in the dictionary by passing the keys associated with them. If the key doesnt exist, python throws `KeyError` as seen below.

```
mydict[56]
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

Cell In[35], line 1
----> 1 mydict[56]
```

```
KeyError: 56
```

```
mydict[2]
```

```
  'pin'
```

deleting an element

you can use the keyword `del` to delete a key:value pair from the dictionary as shown below

```
mydict
```

```
{'city': 'delhi', 2: 'pin', 'num': [3, 4, 5, 6], 'pin': 2}
```

```
del mydict['city']
```

```
mydict
```

```
{2: 'pin', 'num': [3, 4, 5, 6], 'pin': 2}
```

adding an element

since dictionary is not ordered or indexed , there is no sense of end or beginning in a dictionary. There dont exist functions like append,extend,remvoe etc for dictionaries.

adding an element to a dictionary is as simple as choosing a key and assigning it a value in the dictionary object. note that you need to use `[ ]`

```
mydict['abc']=42
mydict
```

```
{2: 'pin', 'num': [3, 4, 5, 6], 'pin': 2, 'abc': 42}
```

you can also add another dictionary to an existing one using function update

```
mydict.update({'qwe':[45,56,'test'],2:999})
mydict
```

```
{2: 999, 'num': [3, 4, 5, 6], 'pin': 2, 'abc': 42, 'qwe': [45, 56, 'test']}
```

note that the key which already existed is simply replaced with new key:value pair instead of creating a duplicate key

extract keys , values and key value pairs

this behaves like a list

```
mydict.keys()
```

```
dict_keys([2, 'num', 'pin', 'abc', 'qwe'])
```

This also behaves like a list

```
mydict.values()
```

```
dict_values([999, [3, 4, 5, 6], 2, 42, [45, 56, 'test']])
```

This is like a list of lists [tuples to be technically correct]. Each list inside is pair of key and value from the dictionary.

```
mydict.items()
```

```
dict_items([(2, 999), ('num', [3, 4, 5, 6]), ('pin', 2), ('abc', 42), ('qwe', [45, 56, 'test'])])
```

you can also create dictionary with list of tuples , where the first member of tuples become key and the second a value

```
new_dict=dict([(1,2),('name','john'),('country','uk')])
new_dict
```

```
{1: 2, 'name': 'john', 'country': 'uk'}
```

creating an empty dictionary

you can do all the dictionary operations on this [accept deleting elements because none exist in an empty dictionary]

```
x={}
```

Sets

sets are collection of individual strictly unique items [no duplicates] of immutable objects [so you cant have lists and dictionaries as elements of set]. They are good for keeping track of unique occurences and support all mathematical set operations . we create a set by putting values in `{ }`, separated by `,`

code below will throw an error because we are trying to put a list which is mutable as one of the elements. Simply put sets can not have lists , dictionaries , sets.

```
myset={2,[3,4,5],7,7,8,9,9,2}
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

Cell In[50], line 1
----> 1 myset={2,[3,4,5],7,7,8,9,9,2}
```

```
TypeError: unhashable type: 'list'
```

```
myset={2,3,3,4,5,7,7,8,9,9,2}
myset
```

```
{2, 3, 4, 5, 7, 8, 9}
```

note that duplicates have been automatically removed

add, remove elements

```
myset.add(10)
myset
```

```
{2, 3, 4, 5, 7, 8, 9, 10}
```

```
myset.remove(2)
myset
```

```
{3, 4, 5, 7, 8, 9, 10}
```


accessing elements and conditional check

sets are unordered , implying that you dont have usual indexing with which you can access the elements [set object is not subscriptable] . you can write a for loop over them , but you cant be sure that the elements will appear in the same order in which you passed them while creating the set

code below will throw an error indicating that sets dont support indexing. If you have to do something like this, you can always go back to lists.

```
myset[2]
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

Cell In[60], line 1
----> 1 myset[2]
```

```
TypeError: 'set' object is not subscriptable
```

```
myset={45,6,7,11,34,9,10,23,11,}
for elem in myset:
    print(elem)
```

```
34
6
7
9
10
11
45
23
```


you can check whether a value exists in the set using `in` operator. Outcome is the boolean value `True` or `False`, depending on whether the value is in the set or not.

```
6 in myset
```

```
True
```

```
7 not in myset
```

```
False
```

set operations

Lets create two sets to see set operations in action.

```
set1={2,4,6,8,10,12,14,16,18,20,22,24,26,28,30}
set2={3,6,9,12,15,18,21,24,27,30}
```

union

its a symmetric operation , meaning `set1.union(set2)` and `set2.union(set1)` will give the same outcome. union returns all elements combined from `set1` and `set2` [no duplicates]

```
set1.union(set2)
```

```
{2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 26, 27, 28, 30}
```

intersection

returns a set with elements present in both the sets, this is also a symmetric operation.

```
set1.intersection(set2)
```

```
{6, 12, 18, 24, 30}
```

set difference

code below will return only those elements from `set1` which are not there in `set2`. This naturally will not be a symmteric operation. Results of `set1-set2` and `set2-set1` are going to be difference

```
set1-set2
```

```
{2, 4, 8, 10, 14, 16, 20, 22, 26, 28}
```

symmetric difference

This function returns all the elements from both sets which are not common or in other words, are exclusive to the sets.

```
set1.symmetric_difference(set2)
```

```
{2, 3, 4, 8, 9, 10, 14, 15, 16, 20, 21, 22, 26, 27, 28}
```

this is equivalent to `set1.union(set2)-set1.intersection(set2)`

creating an empty set

since `{}` is reserved for creating an empty dictionary we will need to use function `set` for creating an empty set.

```
x=set()
```

Tuples

Tuples are ordered but immutable and can store fixed number of items post their creation. They are good for maintaining constrained information. In practice as beginners we will not find many direct use cases , but they are used by many data structures internally .

You can create tuples by passing values to `( )` separated by comma Or simply writing values after `=` separated by commas.

```
mytuple1=(3,4,4,5,6,8,9,9,9)
mytuple1
```

```
(3, 4, 4, 5, 6, 8, 9, 9, 9)
```

```
mytuple2=4,5,5,5,5,5,8,9
mytuple2
```



```
(4, 5, 5, 5, 5, 5, 8, 9)
```

Tuples are ordered just like lists and indexing for them will work just like lists.

```
mytuple1[-1]
```

```
9
```

However they are immutable, meaning you can not change their elements. They dont have any append,extend,remove etc like functions for the same reason.

```
mytuple1[2]=3
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

Cell In[77], line 1
----> 1 mytuple1[2]=3
```

```
TypeError: 'tuple' object does not support item assignment
```

as mentioned earlier tuples dont have any addition or removal functions; you can however add two tuples to create a new one but that doesnt alter the original ones as intended.

If I have to order these data structures lists, dictionaries, sets, and tuples that we just learnt about in terms of how often you will be using them. I will start with keeping lists on top, you will encounter and use various forms of lists very frequently in your programming. Dictionaries will come second. Sets and tuples will not be used very frequently in our case.

Next we are going to learn to write conditions and flow control with if-else blocks.

Conditions and Flow Control

you can write conditions with usual symbols found in other tools you might have used in the past . Here is a list of symbols which python makes use of for conditions

- Equality => `==`
- Inequality => `!=`
- Greater than => `>`
- Greather than or equal to => `>=`
- Less than => `<`
- Less than or equal to => `<=`
- Negation => `not` [add not in front of a condition or boolean and it flips it]

below given are some examples . Outcome of a conditional statement is boolean `True` or `False` depending on whether the condition is true or false. you can write complex conditions by combinining multiple conditions using `and` or `or` keywords with the usage of parenthesis.

Use parenthesis `( )` for better readability and consistent evaluation of the conditions . Outcome of a condition [simple or compound] will be single `True` or `False`.

Also note that, by using `not` you can flip the boolean value of the outcome.

```
x=4
y=34
x==y , x!=y , x>y, x<y, x>=y, x<=y , not True, not False, not x==y
```

```
(False, True, False, True, False, True, False, True, True)
```

you can similarly write conditions on strings as well. for equality/inequaluty string comparison is case sensitive. For other comparisons it follows dictionary order. [My advice here is that avoid using less than greater than comparisons for strings]

Compound Conditions

try to guess what the outcome will be and revisit if you are wrong. make changes and experiment. For refreshing your memory:

- combining two booleans with `and` will only be True if both of them are True
- combining two boolean with `or` will only be False if both of them are False

```
(not x>45 and y==34 ) or ('python'=='Python' or 'python'=='python')
```

```
  True
```


Flow Control

So far we have been writing programs where each line gets executed in sequence. As you start writing more mature programs, you will realise that many times you need to devide your program into multiple difference blocks of code and execute them conditionally [if-else] or iteratively [for and while loops] or repeatedly [functions and classes].

code blocks in python are defined by indentation level. some rules for indentation

- indentation needs to be consistent within a code block. avoid mixing tabs and spaces
- code blocks start with symbol `:` which appears when you start a syntactical code block. this could be if-else block, or a loop, function, class etc . once you have symbol `:` , python expects indentation, without which it will throw error. It will also throw error if you forget the symbol `:` and start with a code block.

we will explore that with `if-else` statements

for the code written below: `%` is used as modulo operator in python. it gives remainder on divison. for example if you divide 100 by 32, remainder will be 4. This operator is generally used to check divisibility [if `a` is divisible by `b` then `a%b==0` .]

```
100%32
```

```
4
```

```
x=10

if x%2==0:
    print('this is an even number')
```

```
this is an even number
```

try chagning x to an odd number you will notice the code inside `if` block is not executed because the condition isnt true anymore. One more thing to notice here is that `if` block does not need an accompanying `else` block by default. However a stand alone `else` block doesnt make sense

```
x=10

if x%2==0:
    print('this is an even number')

else :
    print('this is an odd number')
```

```
this is an even number
```

note that for starting `else` block you need to break the indentation and again start at one level below at the same level as `if` block because of two reasons

- `else` is a separate code block
- and it accompanies/compliments `if`

you can write as many code lines in a code block as you want. it need not be limited to just one line as shown above

selective sequential check vs all condition check [using `elif` short for `else if` vs using just `if`]

run the following codes and try to figure out how the two behave differently . Feel free to make changes in the initial inputs and conditions and see if you can guess correctly what the outcome of the codes will be

```
x=[3,4,5,6,7,8]
```

```
if 3 in x:
    print('3 is in the list')
if 10 in x:
    print('10 is in the list')
if 4 in x:
    print('4 is in the list')
else:
    print('end of checks')

```

```
3 is in the list
4 is in the list
```

```
if 3 in x:
    print('3 is in the list')
elif 10 in x:
    print('10 is in the list')
elif 40 in x:
    print('4 is in the list')
else:
    print('end of checks')
```

```
3 is in the list
```


Iterative Operations : For and While Loop

In any programming language you will find work arounds whenever you need to run some code lines which are exactly the same barring few inputs which are coming from some data collection. Essentially the task which you are doing in the code lines is iterating over the data collection and doing same thing for all elements of that data collection . These work arounds which support iterative operations are called loops. there are two kinds :

- for loops : they iterate over a data collection [iterable] [e.g. `list` , `sets` , `arrays` , `series` etc], execution of code stops when there is an error or the data collection elements are finished
- while loops: they iteratively execute the code block contained in them until a condition is true

consider this hypothetical task of going through all the elements of the list and retaining only those elements which are divisible by both 3 and 5. in the first example , instead of retaining them in another list, we will simply print them

```
x=list(range(3,40,3))
x
```

```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39]
```

general structure of the for loop will be as following

```
for name_for_the_index_or_element in data_collection_or_index_range:
    do something here
```

- name for the index or element : you can directly iterate over the elements or get them using an index . the name that you chose keeps on getting reassigned the value at each next iteration of the loop
- data collection or index range : if you are directly iterating over the elements then you put data collection in the for statement or if you are using an index then you put range of values which the index is going to take

lets do the task mentioned above . I wil show you both using index as well as directly iterating over the data elements .

Note that function `range(n)` creates a list $[0, 1, 2, \cdots, n-1]$.

```
list(range(len(x)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
for i in range(len(x)):
    if x[i]%3==0 and x[i]%5==0:
        print(x[i])
```

```
15
30
```

```
for elem in x:
    if elem%3==0 and elem%5==0:
        print(elem)
```

```
15
30
```

if we want to retain this result in a list, we can start with an empty list and whenever the condition is satisfied , append the element to the list.

```
result=[]

for elem in x:
    if elem%3==0 and elem%5==0:
        result.append(elem)
```

```
result
```

```
[15, 30]
```

lets do the same task with a while loop. Every time we write a while loop we will have to think how we can come up with a condition which will suffice for what we are trying to do . In this case we can create a placeholder element and everytime we go inside the while loop, we will increase its count. The condition we will use : when that count surpasses number of elements; we stop.

```
i=0
result=[]
while i<len(x):
    if x[i]%3==0 and x[i]%5==0:
        result.append(x[i])
    i=i+1
```

```
result
```

```
[15, 30]
```

In this particular example , using while loop might seem like an overkill, and that is right. But in many situation iterating until a condition is met is more intuitive instead of iterating over all the elements of some data collection. Although in this case for loop looks like a more natural choice, but both ideas have their own place in the programming eco system

List Comprehension

Python has an interesting functionality of containing the for loop within a list . The need for the idea arises from : many times we need to retain results of iterative code in another list , we can do that using this functionality [known as list comprehension] where we write the for loop directly in the list .

lets first write a for loop and then see its equivalent list comprehension

```
x=list(range(10))
x
```

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

```
squares_forloop=[]

for elem in x:
    squares_forloop.append(elem**2)

squares_forloop
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
squares_lc=[elem**2 for elem in x ]
squares_lc
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```


the first calculation [`elem**2` in this case] essentially becomes element of the new list and that calculation is done `for elem in x` iterating over members of x, and the results keep on getting added to the list as new memebrs separated by comma automatically.

we can use if-else conditions also in list comprehension but in my experience that code becomes difficult to read for a lot of practitioners and in such a scenario i will advise you to write explicit for loops instead of using list comprehension .Below here is the earlier task that we did with divisibility of 3 and 5, done with using list comprehension along with the for loop that we wrote; for you to draw parrallels.

```
x=list(range(3,40,3))
x
```

```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39]
```

```
result_fl=[]

for elem in x:
    if elem%3==0 and elem%5==0:
        result_fl.append(elem)
result_fl
```

```
[15, 30]
```

```
result_lc=[elem for elem in x if elem%3==0 and elem%5==0]
result_lc
```

```
[15, 30]
```

Exercise

create a list containing random letters . [i will do this part for you]

```
import numpy as np
```

```
x=np.random.choice(list('abcdefghijkl'),100)
x
```

```
array(['c', 'b', 'l', 'l', 'k', 'g', 'f', 'k', 'g', 'f', 'a', 'k', 'b',
       'h', 'a', 'b', 'h', 'l', 'e', 'l', 'c', 'e', 'k', 'g', 'a', 'c',
       'i', 'j', 'k', 'b', 'j', 'j', 'c', 'j', 'i', 'l', 'g', 'a', 'c',
       'h', 'b', 'd', 'd', 'l', 'i', 'a', 'k', 'l', 'a', 'i', 'f', 'g',
       'k', 'f', 'c', 'd', 'k', 'h', 'j', 'g', 'k', 'l', 'h', 'l', 'e',
       'd', 'g', 'a', 'f', 'g', 'h', 'i', 'h', 'g', 'a', 'f', 'l', 'j',
       'g', 'i', 'd', 'f', 'h', 'l', 'h', 'f', 'b', 'k', 'a', 'j', 'k',
       'l', 'j', 'h', 'e', 'c', 'f', 'd', 'e', 'f'], dtype='<U1')
```

since this is random, results are going to be different on different machines and even on the same machines between different runs

1. create a dictionary which has the unique letters as keys and their counts in the list as values . Follow these steps

```
# create an empty dictionary . hint dict={}
# write a for loop iterating over the list/array created above .
# inside the for loop for each element check if the element is already in the keys of the dictionary
# [hint if letter in dict.keys()]
# if it is there then increase the count [hint: dict[letter]=dict[letter]+1]
# if it is not there then intialise it [hint: dict[letter]=1]
```

2. find which letter has maximum frequency [this can be a little challenging as i will ask you to search/find things which we have not discussed in the class]

```
# convert the dictionary to list of tuples where each tuple is key value pair [hint : dict.items()]

# search "how to sort a list of tuples with a particular position element".
# go to stackoverflow or elsewhere and see if you can make sense of any of the suggested solutions
# ignore the ones with lambda functions
```

Functions

Functions in python in another language exist so that you dont have to write same piece of code again and again to do similar tasks. Instead of that you simply call that entire piece of code with the function name with required inputs for the task and it internally executes that block of code of any time you call the function and returns with the results . We use keyword `def` to define a function code block . Here is a simple example of a function which does first part of the exercise given in earlier section .


```
import numpy as np
x=np.random.choice(list('abcdefghijkl'),40)
```

```
x
```

```
array(['k', 'g', 'e', 'l', 'h', 'b', 'g', 'b', 'f', 'a', 'f', 'c', 'b',
       'e', 'g', 'k', 'b', 'e', 'l', 'g', 'a', 'l', 'd', 'h', 'j', 'h',
       'h', 'l', 'h', 'g', 'i', 'j', 'c', 'k', 'j', 'f', 'h', 'h', 'e',
       'b'], dtype='<U1')
```

```
def freq_dict(some_list):

    result_dict={}

    for elem in some_list:

        if elem in result_dict.keys():
            result_dict[elem]+=1
        else:
            result_dict[elem]=1

    return result_dict
```

```
freq_dict(x)
```

```
{'k': 3,
 'g': 5,
 'e': 4,
 'l': 4,
 'h': 7,
 'b': 5,
 'f': 3,
 'a': 2,
 'c': 2,
 'd': 1,
 'j': 3,
 'i': 1}
```



```
mylist=list('slkdjhfdfsanalkdpowmqdposdnqnweqjnwq')
```

```
freq_dict(mylist)
```

```
{'s': 3,
 'l': 2,
 'k': 2,
 'd': 5,
 'j': 2,
 'h': 1,
 'f': 2,
 'a': 2,
 'n': 4,
 'p': 2,
```

```
 'o': 2,
 'w': 3,
 'm': 1,
 'q': 4,
 'e': 1}
```

couple things to note here :

- input to function [here that is `some_list`] are popularly known as `arguments` to the function or just `args`
- function can take any number of inputs
- object names used inside the function are local to function in scope. they will not be available globally. this allows people to reuse object names inside functions without worrying about that choice messing up things outside the function
- processing inside a function stops once python encounters a return statement . Anything that you right after that in the function will not be executed unless there is way to circumvent the return statement. An example given below uses multiple return statements and also demonstrates that the functions can be called inside themselves. This is called recursive functions, all though its not a good programming practice .


```
def factorial(n):

    if n==1:
        return 1
    else:
        return n*factorial(n-1)
```

```
factorial(5)
```

```
  120
```

Default arguments

when you define args for a function, at that moment you can give them some default value; which will be used only if user doesnt give any input for those arguments . If user does provide some input for those arguments , then default value provided in the function definition is overwritten

In the example below we have given default value of `1` to `a` by simply writing `a=1` while defining the arguments to the function. [similar way was used for other arguments as well]

```
def mysum(a=1,b=10,c=100):

    return a+10*b+100*c
```

```
mysum()
```

```
10101
```

experiment with passing just 1 arugment or 2 and so on, and see what default values are used. Remember that one you pass an argument, it overrides the default values.

When you are calling your function by passing argument values only without the argument name, they are called `positional` argument. When you do that, the values that you pass are assigned to arguments in the function in the same sequence as they were in while being defined in the function code. so in the example given above, if you pass a single value as argument, it will go to `a`. If you pass two values, first one will go to `a` and second will go to `b`.

when you are calling the function, if you name your arguments [named arugments or keyword arguments], the sequence in which you pass the arguments does not matter

```
mysum(c=30,a=2,b=1)
```

```
3012
```


but if you dont name your arguments , values are assigned sequentially and if there are not enough inputs available , for the remaining arguments , default values are used

```
mysum(3,4)
```

```
10043
```

in general you should avoid mixing named [called `keyword args`] and unnamed arguments [called `positional args`] together. By rule, once you start naming arguments in your function call, an unnamed arugment can not be passed . But a named argument can follow after a positional argument

```
mysum(a=3,4,5)
```

```
  Cell In[12], line 1
    mysum(a=3,4,5)
                ^
SyntaxError: positional argument follows keyword argument
```

```
mysum(3,4,c=10)
```

```
1043
```

however you need to keep in mind , that once you called a function with unnamed arguments, it started assigning the values internally in the function, so `a=3` and `b=4` happens and then it encounters named argument `c`. this is ok, however if you tried something like this

```
mysum(3,4,a=10)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

Cell In[14], line 1
----> 1 mysum(3,4,a=10)
```

```
TypeError: mysum() got multiple values for argument 'a'
```

you get an error because `a` has already been assigned

*args and **kwargs

you can techincally write functions with variable number of arguments or arguments with names decided at the time of execution, but that is something which is out of scope for us here . We will look at `*args` and `**keyword` from the perspective of calling a function where inputs can be in a list or dictionary bunched up together instead of conveniently available separately.

consider that input to our function written above, arive from some upstream process in a list like given below.

```
myargs=[4,5,6]
```

Now if we pass them as is , it throws error.[can you think why?]

```
mysum(myargs)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

Cell In[16], line 1
----> 1 mysum(myargs)
```

```
Cell In[8], line 3, in mysum(a, b, c)
      1 def mysum(a=1,b=10,c=100):
----> 3     return a+10*b+100*c
```

```
TypeError: can only concatenate list (not "int") to list
```

above call throws an error because python considers the list to be a single object, however if we call the function with `*` adjacent to the list , then python would consider each element of the list a separate input to the function. This `*` before a list in a function call is called an unpacking operator.

```
mysum(*myargs)
```

```
654
```

this was similar to writing `mysum(mylist[0],mylist[1],mylist[2])` . this might seem doable for now. But imagine doing this manually instead of using `*mylist` when the list has 15-20 elements.

similarly sometimes you might have your inputs saved in a dictionary with the argument name as keys . In that case you can use `**` adjacent to dictionary to unpack them as named arguments when calling that function

```
mydict={'a':3,'b':4,'c':5}
```

```
mysum(**mydict)
```

```
543
```

A general advice , instead of directly starting to write a function for the process; write simple code first for that process. Then figure out what are the inputs to the process and then wrap the whole thing into a function. It will be easier to debug step by step and update instead of writing the whole function and debug through multiple runs

Classes

By now , you might have seen classes written in python a couple times . It could have been in our examples or during your own exploration across the interwebs or elsewhere . The reason you see them everywhere is because they make things organised , legible and are considered **the** way to write better code . We havent however covered them formally in a structured manner, we will do just that in this unit .

Before we dive into python specific syntax for the topic , i want to you to rethink about these ideas pertaining to programming as a practice .

- Programming is a way to lay down generic abstraction of logic for problem solving . All programming languages approach this in many different ways, the difference arising from mostly the first domain of usage at the very birth of the language. Overtime however these approaches turn into a style , not necessarily right or wrong, but a means to have sustained consistency across the codebases of different programmers.
- When you are writing code as a mature programmer (problem solver), your vision should be covering many more possible scenarios than just the one that you immediately see right in front of you.
- A very key idea behind writing programs in industry is that :
    - its going to be **reused** in the future
    - and **reused by others** in the future to solve similar problem
- This calls for your program to have some basic aesthetics :
    - Your program should be easy to read and understand . A program broken into smaller components is much easier to understand bit by bit than a 1000 line behemoth at once.
    - Your program should be easy to edit and efficient in terms of affecting that change . For example a component which is being used across your program should be written in such a way that a single edit can make the change flow through all the instances . On the other hand a component which is not supposed to be shared across instances, should have a barrier protecting it in some way from getting accidentally altered in the course of program execution for other instances
    - Your program is going to grow , into having more functionalities eventually . This evolution should be possible without altering the existing implementation. In other words , if someone is trying to solve a similar problem , they need not be starting from scratch; instead , they should be able to build on top of your code without hurting any other current applications using your code.

you will also realise as we progress , how these ideas are the driving influence of many implemented functionalities in python classes. Ok, keep those ideas in mind and as you become more mature as a programmer, feel free to add some of your own thoughts to the same .

Why Classes ?

A very intuitive answer to this question is another question . That is, why rooms and walls in a house? They exist to make your house more organised . Contain similar things (data) , and activities (methods) together in one place and provide a way to keep those separate from activities and things which are different from them. Technically in a house; kitchen and toilet can exist side by side without walls , but

would you want to live in that house?

Lets dive into an example which we will keep on modifying through out this discussion in order to add details as we discuss them .

Consider following bit of program :

```
day = 12
month = 3
year = 2022
```

taken together these three , make a date 12th march 2022. But there is nothing which is tying them together . And things can get messy and confusing if we had another date to handle along with this one. We will not only have to create many more objects with plausibly similar sounding names , but we will have to be careful about them not getting mixed up with each other .

at a very basic level , classes will provide you a way to do this [keep in mind that they can do much more than that, but we can start our discussion with this]

```
class Date:
  day=12
  month=3
  year=2022

```

I can create an object of this class as follows :

```
a=Date()
```

`day` , `month` and `year` are attributes of this class. we can access this using our `.` dot notation .

```
a.day, a.month,a.year
```

outcome :

```
12,3,2022
```

formally `a` here is an instance of the class `Date` .

In its current form class Date is only serving the purpose of tying up things together into a single object . However using this i can not really create *another date* with different values for the attributes `day` , `month` and `year` . we are stuck with 12th march 2022.

So we want to have another date but with different initial values for its attributes . we can achieve that by making use of a special fixed method `__init__` [has **two** underscores before and after the word init] .

```
class Date :
  def __init__(self,day,month,year):
    self.day=day
    self.month=month
    self.year=year
```

```
a=Date(12,3,2022)
b=Date(1,1,2023)
```

```
a.day,b.day
```

outcome : `12` , `1`

you can access other attributes also in the same way.

with these two examples we brought couple new concepts here . lets discuss those details one by one before we move forward.

constructor , class attributes and instance attributes

When you write `a=Date()` , python internally calls an otherwise hidden method `__new__` to create an object of the class `Date` . you can write your own version of `__new__` within a class definition if you want to do something more than just creating an object of the class . This `__new__` is same as what is known as constructor in few other programming languages like `JAVA` or `C` . All that it does in its current form is to create an object of the class.

The attributes that we created outside the method `__init__` are called class attribute . They are shared by all objects of the class . And thats why we couldn't create two different dates when `day` , `month` and `year` were class attributes . If you change a class attribute after creating an object, that change happens for all the instances of the class. lets understand that better by doing.

Lets also add a class attribute to our current class code so that we have both instance and class attributes and understand the difference better .

```
class Date :
  welcome_message='hello!'
  def __init__(self,day,month,year):
    self.day=day
    self.month=month
    self.year=year
```

lets create two objects of the class Date

```
a=Date(12,3,2022)
b=Date(10,11,2022)
```

if we look at attributes of these instances of class Date, we will find them having 3 attributes each

```
a.__dict__
```

outcome = `{'day': 12, 'month': 3, 'year': 2022}`

```
b.__dict__
```

outcome = `{'day': 10, 'month': 11, 'year': 2022}`

All these instances however have access to class attributes by default .

```
a.welcome_message,b.welcome_message
```

outcome : `('hello!', 'hello!')`

and all the instances have same attribute value which we assigned to the class attributes . Key thing to understand here is that if the instance attribute with same name doesn't exist , then only it defaults to class attribute of the same name .

we can make changes to an instance attribute by giving reference of the instance . for example :

```
a.day=25
```

now if we look at attributes of a

```
a.__dict__
```

outcome = `{'day': 25, 'month': 3, 'year': 2022}` . value for attribute `day` has changed.

we can create additional attributes as well

```
a.hour=12
a.__dict__
```

Outcome: `{'day': 25, 'month': 3, 'year': 2022,'hour':12}` . another attribute with the name `hour` has been added to the instance.

You need to notice that this change in one instance , whether changing value of existing attribute or addition of new attributes has no effect on any other instance .

```
b.__dict__
```

outcome : `{'day': 10, 'month': 11, 'year': 2022}`

if we want to make changes to a class attribute , we will have to take reference of the class [not the instance]

```
Date.welcome_message='hi'
```

This change now will affect all the instances of the class at once . lets see

```
a.welcome_message,b.welcome_message
```

outcome : `('hi!', 'hi!')` . change has appeared in both `a` and `b` without having to explicitly make those changes in individually.

if we try to make this change happen with reference of instance , instead of changing class attribute; it will create another instance attribute which will override the class attribute . lets understand with an example what i mean here .

```
a.welcome_message='hola'
```

Outcome: `{'day': 25, 'month': 3, 'year': 2022,'hour':12,'welcome_message='hola'}` you can see the new instance attribute appearing here .

Now onwards if we access `a.welcome_message` , it will look for this instance attribute instead of the class attribute .

```
a.welcome_message,b.welcome_message
```

outcome : `('hola', 'hi')`

Even if we make changes to class attribute now , it will not make changes to instance attribute `welcome_message` of `a` . The change will appear only for those instances which do not still have an instance attribute with the same name .

```
Date.welcome_message='jump!'
a.welcome_message,b.welcome_message
```

outcome : `('hola', 'jump!')`

Key ideas out of this discussion are :

- Attributes created outside any method under class definition are class attributes
- We can change class attributes , using the reference to class . This change affects all the instances of the class at once
- If we use instance reference instead , it ends up creating an instance attribute with the same name
- Instance attributes can be changed using instance reference and these changes do not affect any other instance
- We should not mix names of class and instance attributes , it can lead to a lot of confusion down the line.

self ?

We very conspicuously avoided discussion on the first argument to our `__init__` function so far . Lets understand that now. Remember `__new__` ? It creates object of the invoked class. We also learnt that in order to make changes to an attribute of an instance or to create an attribute of an instance , we need to pass reference of that instance .

Inside the class definition, `self` does that job. If we want to access attributes of an instance inside the class definition anywhere , we can use `self.name_of_the_attribute` . This idea will become more clear in time when we discuss different kind of methods which can exist in class definition . [instance method, class method and static methods]

Until now we have seen example of only one method in the class definition `__init__` . It was an instance method . All instance methods in class definition will have `self` as their first argument . And it will be used as reference for accessing attributes associated with the class instance [object]

> technically you can have some other word also as the name of the first argument of an instance method , it will still have the same meaning and functionality. But i will advice to keep on using the word self . Its universally used convention, and unnecessarily using something else will make your code confusing to others .

Instance Methods , Class Methods and Static Methods

All methods in a class definition are instance methods unless explicitly declared to be class or static methods using specific decorators for the same .

lets start with an instance method first , we will get little more experience with and understanding about the `self` .

But before we dive into writing methods/functions inside our class definition , we need to address a curious question. Binding different data attributes together with a custom class makes sense. it makes them portable , easy to manage etc. Why do we now want to add functions also in the mix ?

- One thing a class definition does , is to shield whatever data [attribute] or activity [functions] from the outside world . You can access them only with reference to the class.
- This necessicity for reference makes the user aware of the context in which these are being used . Sitting for example could mean two different things depending on whether we are sitting in a toilet vs in our study room.
- This utility of context awareness which usage of classes brings , applies to both data as well as functions

Consider adding a method to our class `Date` for calculating which quarter it falls in . here is the updated code

```
class Date :
  welcome_message='hello!'
  def __init__(self,day,month,year):
    self.day=day
    self.month=month
    self.year=year
  def quarter(self):
    q=int(self.month/4)+1
    return q
```

lets create an object the class `Date` for this modified definition

```
a=Date(3,7,2022)
```

we can call this function in two ways .

```
a.quarter()
```

Outcome : `3`

Here we are calling function `quarter` with an instance reference , be default this reference gets passed as the first argument to the function which internally is referenced as `self`

another not so common way of calling this function is with class reference

```
Date.quarter(a)
```

Outcome : `3`

Now a class can have any number of instances , we need to explicitly pass reference to object in question that is `a` here in our example. This again internally gets used as `self`

this was example of an instance method. in case it needed more arguments depending on what it does , they will follow after `self` . In our example we did not need more argument, thats why we had just self there .

It seems like just instance methods should suffice why do we need these other kinds ? [class and static methods]. Lets consider some use cases in order to understand that need.

In its current form we can create a Date object only when we supply day, month and year separately as numbers . What if we also want to add functionality to create a Date object from a string like this : `21-4-2022` [21st April 2022] .

we can add a method which extracts day, month and year from this string and then create an object of class `Date` from those. But we can not have another `__init__` method which takes now a string as an argument. what we can do is to use reference to class which can call the earlier `__init__` method as usual internally.

class methods have this ability to refer to a class. lets see an example implementation to understand this better.

```
class Date :
  welcome_message='hello!'
  def __init__(self,day,month,year):
    self.day=day
    self.month=month
    self.year=year

  def quarter(self):
    q=int(self.month/4)+1
    return q

  @classmethod
  def from_string(cls,date_as_string):
    elems=date_as_string.split('-')
    day,month,year=[int(x) for x in elems]
    mydate=cls(day,month,year)
    # this is same as Date(day,month,year) given cls is a reference to class
    return mydate

```

the decorator `@classmethod` makes it so that the first argument to the function is considered reference to the class instead of any instance. we can now using function `from_string` , create an object of class `Date` from a string.

```
a = Date.from_string('21-04-2022')
a.__dict__
```

Outcome : `{'day': 21, 'month': 4, 'year': 2022}`

We wrote class `Date` to create a room for all things to do with date. There can be some functionalities related to date which don't really require us to working with an object of class Date. Meaning they don't require access to class or instance attributes or functions , there is no obligation towards having their first arguments as a reference to an instance or class. All that they do is something related to dates.

For example, lets say we want to check whether all components of a string date are numeric or not. we can add that as a static method to our class definition.

```
class Date :
    welcome_message='hello!'
    def __init__(self,day,month,year):
        self.day=day
        self.month=month
        self.year=year

    def quarter(self):
        q=int(self.month/4)+1
        return q

    @classmethod
    def from_string(cls,date_as_string):

```

```
        elems=date_as_string.split('-')
        day,month,year=[int(x) for x in elems]
        mydate=cls(day,month,year)
        return mydate

    @staticmethod
    def is_numeric_date(date_as_string):
        day,month,year=date_as_string.split('-')

        return day.isdigit() and month.isdigit() and year.isdigit()
```

we can call this static method with class reference without really having to create a date kind object first.

```
Date.is_numeric_date('21-Apr-2022')
```

outcome : `False`

property

sometimes we might want to have an attribute which can not be changed even with instance reference, but we would still want it to get updated if its being calculated using some other attributes and those attributes see some change.

For example , lets say we want to have an attribute for our date which is either `True` or `False` depending on whether the year is a leap year or not .

lets try to add that with ideas that we are currently aware of .

```
class Date :
    welcome_message='hello!'
    def __init__(self,day,month,year):
        self.day=day
        self.month=month
        self.year=year
        self.leapyear=year%4==0

    def quarter(self):
        q=int(self.month/4)+1
        return q

    @classmethod
    def from_string(cls,date_as_string):
        elems=date_as_string.split('-')
        day,month,year=[int(x) for x in elems]
        mydate=cls(day,month,year)
        return mydate

    @staticmethod
    def is_numeric_date(date_as_string):
        day,month,year=date_as_string.split('-')
```

```
23
24          return day.isdigit() and month.isdigit() and year.isdigit()
```

lets create an object now

```
1 a=Date(2,12,2022)
2 a.leapyear
```

Outcome : `False`

so far so good. But this attribute can be modified like this

```
1 a.leapyear=True
2 a.leapyear
```

outcome : True

In our context , this should not be allowed . leap year should only be true if year is divisible by 4 and should only be set internally .

second issue with this is that if we updated the year , this attribute will not be updated. this is what i mean.

```
1 b=Date(2,12,2022)
2 b.leapyear
```

Outcome : `False`

```
1 b.year=2020
2 b.leapyear
```

Outcome : `False`

See here , despite the year now being a leapyear , the attribute leaper is still false.

we can address both these issues with one decorator known as `@property` . lets see that in action in our own code.

```
1  class Date :
2      welcome_message='hello!'
3      def __init__(self,day,month,year):
4          self.day=day
5          self.month=month
6          self.year=year
7
8      @property
9      def leapyear(self):
10         return self.year%4==0
11
12     def quarter(self):
```

```
        q=int(self.month/4)+1
        return q

    @classmethod
    def from_string(cls,date_as_string):
        elems=date_as_string.split('-')
        elems=[int(x) for x in elems]
        mydate=cls(*elems)
        return mydate

    @staticmethod
    def is_numeric_date(date_as_string):
        elems=date_as_string.split('-')

        return all(map(str.isdigit,elems))
```

although `leapyear` here is written as function, it provides an attribute named leapyear due to the decorator `@property` .

```
a=Date(2,12,2022)
a.leapyear
```

Outcome: `False`

lets try to change the attribute to `True` now .

```
a.leapyear=True
```

this results in following exception:

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [66], in <cell line: 1>()
----> 1 a.leapyear=True

AttributeError: can't set attribute
```

Ok , our first wish is granted with this . leapyear can not be edited now.

lets see what happens if modify the value of year , will it modify leap year too? lets see

```
a.year=2020
a.leapyear
```

Outcome : `True`

this is the second thing that we wanted. If year gets changed , the derived attribute leapyear should also get updated. we are able to achieve both our goals with `@property` decorator.

to summarise what decorator `@propety` is doing

- it creates an attribute which can not be modified manually
- this attribute will be updated automatically if it is derived from an editable attribute upon its change

attribute created with function with `@property` decorator need not necessarily be derived from another property with some elaborate calculation. sometimes it can simply be used to create an non-editable attributes . here is a simple example.

```
class Square:
    def __init__(self, length):
        self._length = length

    @property
    def area(self):
        return self.length**2

    @property
    def length(self):
        return self._length
```

this way of adding an attribute which has its name starting with an `_` underscore tells python programmers , that this particular attribute is supposed to be internal to the class and should not be touched by users. However its a more of convention than an enforced idea.

the proper name attribute `length` without an `_` underscore , has been created with a function with `@property` decorator and that is impossible to edit.

dunder methods

special functions with double underscores and after are not limited just `__new__` and `__init__` . There are many which are present at the backend of every class by default , there are ones which you can override to do special things with your object.

for example function `__repr__` can be used to display a more readable version of your object.

for example , in current form of our code , when we just write the name of our object and execute we get something like this :

```
a
```

outcome : `<__main__.Date at 0x10ed5c550>`

this memory address tells us nothing about the object. lets change this behaviour. we are also going to write another function `__str__` and then discuss the difference in purpose for both `__str__` and `__repr__`.

```
class Date :
    welcome_message='hello!'
    def __init__(self,day,month,year):
        self.day=day
```

```
        self.month=month
        self.year=year

    @property
    def leapyear(self):
        return self.year%4==0

    def quarter(self):
        q=int(self.month/4)+1
        return q

    @classmethod
    def from_string(cls,date_as_string):
        elems=date_as_string.split('-')
        elems=[int(x) for x in elems]
        mydate=cls(*elems)
        return mydate

    @staticmethod
    def is_numeric_date(date_as_string):
        elems=date_as_string.split('-')
        return all(map(str.isdigit,elems))

    def __repr__(self):
      return(f'Date({self.day},{self.month},{self.year})')

    def __str__(self):
      return(f'Date is {self.day} day of month {self.month} in the year {self.year}')


```


now if we create an object of class `Date` and display it , the outcome will not be as meaning less.

```
a=Date(2,12,2022)
a
```

Outcome: `Date(2,12,2022)`

```
str(a)
```

Outcome : `'Date is 2 day of month 12 in the year 2022'` .

In Summary :

- `__repr__` determines what string will be displayed when you try to display object as is . As such we can write `__repr__` function to do whatever we want, however convention is to have `__repr__` return a string which can be used as is to create said object of the class . in the result above I can use `Date(2,12,2022)` to create an object of class `Date` with mentioned attributes . If you apply method `eval` directly on that will also result in creating the object.

- `__str__` determines what will be the outcome if the object is converted into a string. Convention is to have output of this to be in human friendly language.
- `__repr__` is a fallback for `__str__` if `__str__` is missing
- Calling `print()` uses `__str__`

there are many more dunder methods that you can implement in your own cotenxt to achieve functionalities like algebraic operations. for example

- `__add__` for `+` operator
- `__eq__` for `==` operator

you can find a more exhaustive list here with more examples : https://analyticsindiamag.com/comprehensive-guide-to-python-dunder-methods/

Inheritance

Remeber at the beginning of this chapter , we discussed that the ability for someone else to be able to use your code in theirs is a must. Thats where inheritance comes . lets see it at work.

Consider that we want to now create an object which also has time in addition to date. Essentially a `DateTime` object. We dont want to modify code for `Date` in case its being used by someone else and modification to the original code breaks their program.

What we can do is to write another class , which inherits all functionality and attributes from the `Date` class and can add something of its own on top of all that.

```
class DateTime(Date):

    def __init__(self,day,month,year,hour,minute):

        Date.__init__(self,day,month,year)
        self.hour=hour
        self.minute=minute

    def is_night(self):

        return self.hour>19 or self.hour<5
```

Couple things going on here :

- All the methods including `__init__` are inherited . If we just want to add another method and are ok with current object attributes , we dont have to an `__init__` method here .
- But if we want to add more attributes to the object in the beginning , we will need to have our `__init__` method
- When you write a method with the same name in the new class , it overrides the one existing in the class from which it is inheritting everything.
- Child class has all the function which are their in the parent class , but the parent class doesnt get anything from the child class [as intended]

- We dont need to use `Date.__init__(self,day,month,year)` . We could have initialised all the attributes afresh in our own `__init__` . That line simply lets you avoid , rewriting whatever was already their in the parent class `__init__`.

```
a=Date(3,12,2022)
b=DateTime(3,12,2022,23,45)
```

```
a.is_night()
```

Outcome :

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [85], in <cell line: 1>()
----> 1 a.is_night()

AttributeError: 'Date' object has no attribute 'is_night'
```

```
b.is_night()
```

Outcome : `True`

```
b.quarter()
```

Outcome : `4`

Usually the dates with time as string looks like this : `3-12-2022 23:45`

because of this our original method `from_string` is going to fail . lets implement that method with our own modifications. [it will override the original `from_string` function in the context of `DateTime` objects]

```
class DateTime(Date):

    def __init__(self,day,month,year,hour,minute):

        Date.__init__(self,day,month,year)
        self.hour=hour
        self.minute=minute

    def is_night(self):
        return self.hour>19 or self.hour<5

    @classmethod
    def from_string(cls,date_as_string):
        date,time=date_as_string.split(' ')
        elems=date.split('-')
        elems.extend(time.split(':'))
        elems=[int(x) for x in elems]
```

```
        mydate=cls(*elems)
        return mydate


```

Now you can see that both are working

```
a=Date.from_string('3-12-2022')
b=DateTime.from_string('3-12-2022 23:45')
```

you can try accessing various attributes and functions and see how it goes. But we are not done fully. For example if you try to display object `b` or convert it to a string , it still displays like date without the time component.

```
b
```

Outcome : `Date(2,12,2022)`

This is happening because we have not overridden the functions `__repr__` and `__str__`. We can do two things

- write the original class `__repr__` and `__str__` in such a way that they work with varying number of attributes [its not going to be always possible to come up with a generic logic]
- in many cases it will be much simpler to simply override the original functions.

I am leaving this for you to modify . and any other functions that you might want to use, add or tinker with.

> Note: Any class can inherit for more than one class as well.

Some little more advanced ideas about python classes are listed here , you can treat them as optional if you feel a little overwhelmed right now.

ENUMS

Consider a scenario , in the context of dates again where you want to have a mapping in place for weekdays with numbers . You need following properties additionally :

- you can iterate through them
- mapping should not be editable , meaning monday should always map to say 1

this essentially is called an enumeration and we can define one of our own by writing a class which inherits from `ENUM` . Lets look at one

```
from enum import Enum

class Day(Enum):
    MONDAY = 1
    TUESDAY = 2
    WEDNESDAY = 3
    THURSDAY = 4
    FRIDAY = 5
    SATURDAY = 6
    SUNDAY = 7
```

You can access these with class name reference like this

```
Day.MONDAY,Day.TUESDAY
```

Outcome : `1` , `2`

you can iterate over them

```
list(Day)
```

Outcome :

```
[
    <Day.MONDAY: 1>,
    <Day.TUESDAY: 2>,
    <Day.WEDNESDAY: 3>,
    <Day.THURSDAY: 4>,
    <Day.FRIDAY: 5>,
    <Day.SATURDAY: 6>,
    <Day.SUNDAY: 7>
]
```


These enumerations can be created for any group of constants that you want to bind together with meaningful strings as their descriptions. They have several benefits . I am quoting some verbatim from this source [https://realpython.com/python-enum/]

- Enabling **direct iteration** over members, including their names and values
- Facilitating **code completion** within [IDEs and editors]
- Enabling **type** and **error checking** with static checkers
- Providing a hub of **searchable** names
- Mitigating **spelling mistakes** when using the members of an enumeration
- Ensuring **constant values** that can't be changed during the code's execution
- Guaranteeing **type safety** by differentiating the same value shared across several enums
- Improving **readability** and **maintainability** by using descriptive names

- Facilitating **debugging** by taking advantage of readable names instead of values with no explicit meaning

Dataclasses

This is a new addition to python and very much needed [it might not be present in your installation if its older, you can come back to this discussion whenever you update your installation]. Below given example comes directly from abstract of PEP557 [https://peps.python.org/pep-0557/#abstract] .

```
@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

this decorator `@dataclass` will internally create many many lines of code making the above code equivalent to :

```
class InventoryItem:
    def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0) -> None:
        self.name = name
        self.unit_price = unit_price
        self.quantity_on_hand = quantity_on_hand
    def __repr__(self):
        return f'InventoryItem(name={self.name!r}, unit_price={self.unit_price!r}, quantity_on_hand={self.quantity_on_hand!r})'
    def __eq__(self, other):
        if other.__class__ is self.__class__:
            return (self.name, self.unit_price, self.quantity_on_hand) == (other.name, other.unit_price, other.quantity_on_hand)
        return NotImplemented
    def __ne__(self, other):
        if other.__class__ is self.__class__:
            return (self.name, self.unit_price, self.quantity_on_hand) != (other.name, other.unit_price, other.quantity_on_hand)
        return NotImplemented
    def __lt__(self, other):
        if other.__class__ is self.__class__:
            return (self.name, self.unit_price, self.quantity_on_hand) < (other.name, other.unit_price, other.quantity_on_hand)
        return NotImplemented
    def __le__(self, other):
        if other.__class__ is self.__class__:
            return (self.name, self.unit_price, self.quantity_on_hand) <= (other.name, other.unit_price, other.quantity_on_hand)
```

```
        return NotImplemented
    def __gt__(self, other):
        if other.__class__ is self.__class__:
            return (self.name, self.unit_price, self.quantity_on_hand) > (other.name,
other.unit_price, other.quantity_on_hand)
        return NotImplemented
    def __ge__(self, other):
        if other.__class__ is self.__class__:
            return (self.name, self.unit_price, self.quantity_on_hand) >=
(other.name, other.unit_price, other.quantity_on_hand)
        return NotImplemented
```

this is hugely convenient , especially if your data object contains many more attributes. You can always write your own version of any of these methods and it will override the one created by `@dataclass` . That said, you are still totally free to use traditional classes to implement what works just as data storage with these added functionalities.

we will stop our discussion on classes as well as base python here. Starting from next chapter we will talk about packages in python which bring in a lot of new functionality to python; related to data handling and processing.