Python Fundamentals

Our assumption of starting with this module is that you have already installed Anaconda distribution for python using links given on LMS, although a better option is to simply google `Download Anaconda` and follow the links [As link on LMS might be an old one , not updated as per the latest release]. Key things before we start with python programming

1. Make sure you go through the videos and have chosen a python editor which you like . (We use Jupyter notebooks in the course , scripts provided will be notebooks. You can use spyder as well, there will be no difference in syntax.)
2. Make sure that you code along , with videos as well as this book. There is no shortcut to learn programming except writing/modifying code on your own, executing and trouble shooting.

Lets begin

Creating Basic Objects with single values

Choose any name and equate to the value which you wish the object to contain. There are few restrictions however on chosing object names which we will discuss few steps later

```
x=4
y=5
x+y
```

```
9
```

For checking what value a particular object holds , you can simply write the object name in the cell and execute ; object value will be displayed in the output .

```
x
```

```
4
```

Just like R, everything in python is also case sensitive , if we now try to execute `X` [X in caps]

```
X
```

```
NameError                    Traceback (most recent call last)

 in
----> 1 X
NameError: name 'X' is not defined
```

You can see that python doesnt recognise X in caps as an object name because we have not created one. We named our object a lowercase x . As an important side note , Error messages in python are printed with complete traceback [Most of which is genereally not very helpful] , best place to start debugging is from the bottom.

Few basic rules to follow when chosing object names :

1. Name should not start with a number
2. Name should not have any space in between
3. No special character is allowed except underscore `_`
4. These rerserved key words should not be used as object names

`['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']`

Object Types and Typecasting

Although we dont need to declare type of objects at the time of creation explicitly , python does assign type to variables on its own. Anything between quotes (there is no difference between single or double quotes) is considered to be of string/character type.

```
x=25
y='lalit'
type(x)
```

> int

```
type(y)
```

> str

`int` here means integer and `str` means string. This assigned type is important as operations are allowed/not allowed depending on type of variables , irrespective of what values are stored in them .

This type assignment to objects affects/dictates what kind of operations are allowed on/with the object. For example adding a number to a string is going to throw and error. Lets try doing that anyway and see what happens .

```
x='34' # python will consider this to be a string because the value is within quotes
x+3
```

```
TypeError                          Traceback (most recent call last)
 in
1 x='34' # python will consider this to be a string because the value is within quotes
----> 2 x+3
TypeError: must be str, not int
```

However we can change the type of object using typecasting function and then do the operations .

```
int(x)+3
```

```
37
```

Just applying this function on x , doesnt change its type permanently , it outputs an integer on which the addition operation could be done . Type of x is still `str`

```
type(x)
```

```
str
```

If we do want to change the type of object , we'd need to equate the output of typecasting function to the original object itself

```
x=int(x)
type(x)
```

```
int
```

Last thing about typecasting , unlike R , if the data contained in a object is such that, it can not be converted to a certain type ; then you'll get an error instead of `missing value` as output

```
int('king')
```

```
ValueError                         Traceback (most recent call last)

 in
----> 1 int('king')
ValueError: invalid literal for int() with base 10: 'king'
```

Numeric Operations

Usual Arithematic operations can be used for numeric type objects (int , float , bool) by using appropriate universal symbols, lets look at some examples

```
x=2
y=19
x+y
```

> 21

```
x-y
```

> -17

```
x*y
```

> 38

```
x/y
```

> 0.10526315789473684

For exponents/power , python exclusively uses double astericks . ^ operator which can be used for exponents in R , should not be used in python for exponents (it does bitwise OR operation)

```
x**y
```

> 524288

You can ofcourse right complex calcualtions using parenthesis . In order to save the results of these computations , you need to simply equate them to an object name. Notice , when you do that, an explicit output will not be printed as we saw earlier

```
z=(x/y)**(x*y+3)
```

If you want to see the outcome , you can simply type the object name where you stored to result

```
z
```

> 8.190910549494282e-41

for using mathematical function you'll need to use package math

```
import math
math.log(34)
```

> 3.5263605246161616

```
math.exp(3.5263605246161616)
```

> 34.00000000000001

Boolean Objects

There are two values which boolean objects can take `True` and `False`. They need to be spelled as is for them to be considered boolean values

```python
x=True
y=False
type(x),type(y)
```

> (bool, bool)

you can do usual boolean operations using both words and symbols

```python
x and y , x & y
```

> (False, False)

```python
x or y , x | y
```

> (True, True)

```python
not x , not y
```

> (False, True)

Note that for negation , `!` is not being used as we did in R. In python you simply use the keyword `not` for reversing boolean values

Writing Conditions

In practice , it doesnt happen too often that we create objects with boolean values explictitly . Boolean values are usually results of conditions that we apply on other objects containing data. Here are some examples

Equality Condition

```python
x=34
y=12
x==y
```

> False

Since x is not equal to y, outcome of the condition is `False`. Note catefully that for writing equality condition , we used two equal to sign , one simple equal to sign is reserved for assignment

In-Equality Condition

```python
x!=y
```

> True

Greater than , Less than

```
x>y
```

> True

```
x<y
```

> False

Greater than or equal to , Less than or equal to

```
x>= y
```

> True

```
x<=y
```

> False

These conditions can be written for character data also . in case of equality , strings should match exactly including lower/upper case of characters as well. In case of less than , greater than kind of comparison , result [True/False] depends on dictionary order of the strings [Not their length]

Membership condition for iterables

you can use operators `in` and `not in` to check if some string/element is prsenent in a string/list . The examples here do not show this in context of lists , because we havent yet introduced them

```
x='python'
'py' in x
```

> True

```
'TH' in x # all string comparisons are case sensitive
```

> False

Compound Conditions

you can use parenthesis to write a compound conditions which is essentially is combination of multiple individual conditions here is an example

```
x=45
y=67
(x > 20 and y<10) or (x ==4 or y > 15 )
```

> True

Notice that , eventual result of the condition is a single boolean value

String Operations

```
x='Python'
y="Data"
```

All string data will need to be within quotes . Note that it doesnt matter whether you are using double quotes or single quotes

length of a string (iterable)

```
len(x),len(y)
```

> (6, 4)

same function `len` will also work in context of lists

Duplicating strings

When a string is multiplied by an integer (not decimals/float) , it results in duplication of the base string

```
'python'*3
```

> 'pythonpythonpython'

Concatenating strings

Addition operator `+` when used between strings does concatenation

```
x+y
```

> 'PythonData'

```
x+' and '+y
```

> 'Python and Data'

```
z=x+' and '+y
z
```

> 'Python and Data'

Converting to lower case

```
z.lower()
```

> 'python and data'

```
z
```

> 'Python and Data'

Note that these functions are giving explicit output [not making inplace changes in the object itself]. If you want to change z itself , you'll need to equate the function call to object it self

```
z=z.lower()
z
```

> 'python and data'

Converting to upper case

```
z.upper()
```

> 'PYTHON AND DATA'

Proper case [first letter capital]

```
z.capitalize()
```

> 'Python and data'

Adding white spaces to string

```
z.rjust(20)
```

> '   python and data'

we just added some white spaces [to left side] to our string z, note that the number passed to function `rjust` represents length of the string after adding white spaces. It does not represent number of white spaces being added to the string. If this input is smaller than the string in question , then the result is same as the original string with not changes whatsoever

```
len(z.rjust(20))
```

> 20

```
z.rjust(3)
```

> 'python and data'

try for yourself and see what these functions do : `ljust` , `center`

Removing white spaces from a string

lets first add leading and trailing spaces to our string and then we'll learn about how to remove those spaces using availabel string functions in python

```
z=z.center(30)
z
```

> '   python and data   '

```
z.strip() # removes both leading and trailing white spaces
```

> 'python and data'

```
z.rstrip() # removes only trailing white spaces
```

> '   python and data'

```
z.lstrip() # removes only leading white spaces
```

> 'python and data   '

Replacing a substring with another

```
z.replace('a',"@#!")
```

> '   python @#!nd d@#!t@#!   '

- first argument represents substring to be replaced
- second argument represents the substring which will replace the substring mentioned in first argument
- By default , all occurences are replaced , however you can use 3rd argument to the function to control that , here is an example

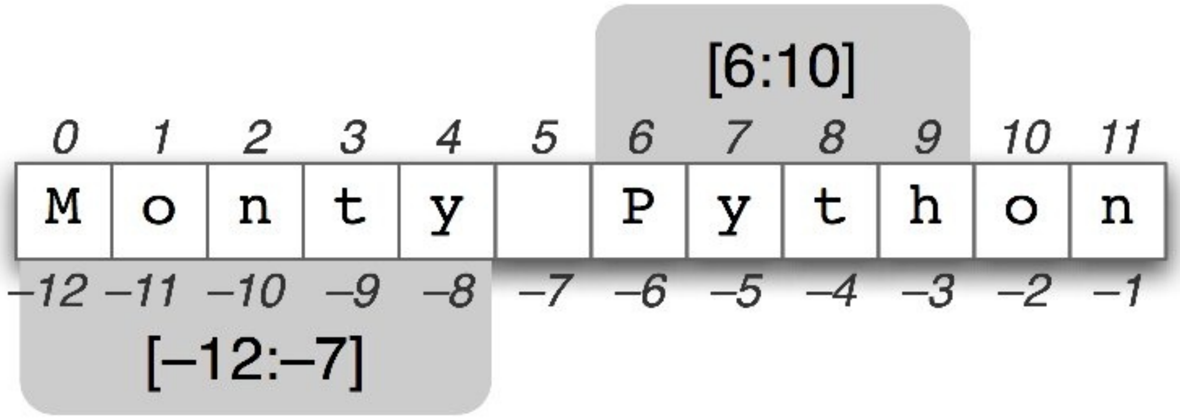
```
z.replace('a',"@#!",2)
```

> '   python @#!nd d@#!ta   '

this only replaces first 2 occurences as specified in the third argument

Position indices and access for iterables [Strings , Lists]

```
x='Monty Python'
```



In Figure above you can see that each individual character of the string (including white spaces) is assigned an index. From left to right index start with 0 . From right to left , index start with -1.

You can use these indices to extract parts of substring.

```
x[6]
```

> 'P'

```
x[-9]
```

> 't'

multiple characters can be extracted as well [contiguous ranges only] by passing range of indices

```
x[6:10]
```

> 'Pyth'

```
x[-12:-7]
```

> 'Monty'

Note that x[a:b] will give you part of string starting with index `a` and ending with index `b-1` [last value is not included in the output]. By default this assumes step size 1; from left to right. If starting position happens to be occuring after the ending position , result is simpy an empty string [instead of an error]

```
x[-7:-12]
```

> ''

You can change step size by making use of third input here

```python
x[1:9]
```

> 'onty Pyt'

```python
x[1:9:2]
```

> 'ot y'

step size 2 means that , starting from beginning , the next element will be at step size 2

you can also have step size negative , in that case , direction changes to :: right to left

```python
x[-7:-12:-1]
```

> ' ytno'

Just like step size (which we dont always specify), we dont need to specify starting and ending points either. In absence of starting position , first value of the strings becomes the default starting point . In absence of ending position, last value of the strings becomes the default end point.

```python
x[:4]
```

> 'Mont'

```python
x[5:]
```

> ' Python'

Note : Same way of indexing will be used for lists as well as mentioned in the title. However in case of lists , index will assigned to each individual element. Rest of the behaviour will be same as strings , as we discussed here

Lists

Lists are our first encounter with data structures in python. Objects which can hold more than one values. Many other data strcutures that we are going to come across in python are derived from these.

Lists are collection of multiple objects (of any kind), you can create them by simply putting the values within square brackets `[]` separated by commas

```python
x=[20,92,43,83,"john","a","c",45]
len(x)
```

> 8

In general you are going to find out, that when you apply len function on a data structure object , outcome is going to be how many elements they contain.

Indexing in context of lists works just like strings as mentioned earlier.Only difference being that, **for lists each individual element is assigned an index.** Here are some examples of using indices with lists. Notice the similarity with strings

```
x[4]
```

> 'john'

```
x[-2]
```

> 'c'

```
x[:4]
```

> [20, 92, 43, 83]

```
x[2:]
```

> [43, 83, 'john', 'a', 'c', 45]

```
x[1:9:2]
```

> [92, 83, 'a', 45]

```
x[-12:-5]
```

> [20, 92, 43]

```
x[-5:-12]
```

> []

```
x[-5:-12:-1]
```

> [83, 43, 92, 20]

Reassigning existing values in a list

The way we access elements of lists , we can re-assign them as well.

```
x
```

> [20, 92, 43, 83, 'john', 'a', 'c', 45]

```python
x[3]
```

```
83
```

```python
x[3]='python'
x
```

```
[20, 92, 43, 'python', 'john', 'a', 'c', 45]
```

Its not necessary that assigned value has the same type as the original one . We can re-assign multiple values also, keeping in mind that there, the reassignment should be done with list of values [its not necessary that number of values should be as many as the original ones]

```python
x
```

```
[20, 92, 43, 'python', 'john', 'a', 'c', 45]
```

```python
x[2:6]
```

```
[43, 'python', 'john', 'a']
```

```python
x[2:6]=[-10,'doe',-20]
x
```

```
[20, 92, -10, 'doe', -20, 'c', 45]
```

Adding new values to a list

There are 4 ways in which we can add new elements to a list

- Simply sum two lists
- Use function append , it adds elements to the end of the list. if you try to append a list to a list, entire list gets added as a single element [we'll see examples]
- Use function extend , this also adds elements to the end of the list. if you try to use extend to add a list to a list, elements become individual elements of the new list [we'll see examples]
- Use function insert to insert values at a specific position in the list

```python
x=[20,92,43,83]
x=x+[-10,12]
x
```

```
[20, 92, 43, 83, -10, 12]
```

```python
x.append(100) # note that, this makes in-place changes to x
x
```

```
[20, 92, 43, 83, -10, 12, 100]
```

Notice what happens when try to add list to a list using append

```python
len(x)
```

```
7
```

```python
x.append(['a','b','c'])
x
```

```
[20, 92, 43, 83, -10, 12, 100, ['a', 'b', 'c']]
```

```python
len(x)
```

```
8
```

```python
x[7]
```

```
['a', 'b', 'c']
```

lets try to do similar thing with function extend instead

```python
x.extend([3,5,7])
x
```

```
[20, 92, 43, 83, -10, 12, 100, ['a', 'b', 'c'], 3, 5, 7]
```

```python
len(x)
```

```
11
```

this time all elements of the list , get added as individual elements

first 3 methods here , however dont enable us to add elements in any desired position in the list. For that we need to use function `insert`

```python
x
```

```
[20, 92, 43, 83, -10, 12, 100, ['a', 'b', 'c'], 3, 5, 7]
```

```python
x.insert(4,'python')
x
```

```
[20, 92, 43, 83, 'python', -10, 12, 100, ['a', 'b', 'c'], 3, 5, 7]
```

There is now an additional element in the list at index 4

Removing elements from a list

function `pop` removes values from specified location. If any location is not specified , it simply removes the last element from the list

```
x
```

```
[20, 92, 43, 83, 'python', -10, 12, 100, ['a', 'b', 'c'], 3, 5, 7]
```

```
x.pop()
```

```
7
```

```
x
```

```
[20, 92, 43, 83, 'python', -10, 12, 100, ['a', 'b', 'c'], 3, 5]
```

If you specify a position, as shown in the next example, element in that index gets removed from the list

```
x.pop(4)
```

```
'python'
```

```
x
```

```
[20, 92, 43, 83, -10, 12, 100, ['a', 'b', 'c'], 3, 5]
```

This however might be a big hassle if we dont have prior information on where in the list value resides which we want to remove . function `remove` comes to your rescue .

```
x.remove(83)
x
```

```
[20, 92, 43, -10, 12, 100, ['a', 'b', 'c'], 3, 5]
```

This will throw an error if the value doesnt exist in the list

```
x.remove(83)
```

```
ValueError                    Traceback (most recent call last)

 in
----> 1 x.remove(83)
ValueError: list.remove(x): x not in list
```

Another thing to note is that, if there are multiple occurences of the same value in the list then at a time only first occurence will be removed. You might need to run remove function in a loop (which we will learn about in some time)

Rearranging values of a list

```
x=[2,3,40,2,14,2,3,11,71,26]
x.sort()
x
```

> [2, 2, 2, 3, 3, 11, 14, 26, 40, 71]

default order of sorting is ascending , we can use option `reverse` and set it to `True` if sorting in descending manner is required

```
x=[2,3,40,2,14,2,3,11,71,26]
x.sort(reverse=True)
x
```

> [71, 40, 26, 14, 11, 3, 3, 2, 2, 2]

Many at times we might need to simply flip [reverse order] the values of a list without really doing any sorting. function `reverse` can be used for that .

```
x=[2,3,40,2,14,2,3,11,71,26]
x.reverse()
x
```

> [26, 71, 11, 3, 2, 14, 2, 40, 3, 2]

Flow control in python

so far we have been looking at object creation and then modifying them in various ways and doing operations on them. We as such havent done anything which follows couple of logical decisions in the process. In this section we'll be learning few new tools in python which enables to include decision making in the programming.

We'll also be learning to write repetetive codes in a more concise manner with for and while loops.

we'll start with if-else code blocks

if-else

```
x=12
if x%2==0:
    print(x,' is even')
else :
    print(x,'x is odd')
```

> 12 is even

couple of important things to learn here , both in context of if-else block and in general about python.

- a condition follows after the keyword if
- colon indicates , end of condition and start of the code block if
- if the condition is true , program written inside the `if` block will be executed
- if the condition is not true , program written in `else` block will be executed
- Notice the **indentation** , instead of curly braces to define code blocks, in python levels of indentations are used , this makes the code easy to read

Lets look at one more example to understand functionality of if-else block and importance of indentation. Lets say , given 3 numbers we are tryin to find maximum value among them

```
a,b,c=30,10,-10 # you can assign values at once like this
# this doesnt work : a=3,b=10,c=-10
```

```
if a>b :
    if a>c:
        mymax=a
    else :
        mymax=c
else :
    if b>c:
        mymax=b
    else:
        mymax=c
```

```
mymax
```

> 30

You can experiment with passing different numbers. Couple of lessons to learn from this

- you can have code blocks inside code blocks
- level of indentation increases as we add more code blocks inside already existing one

```
x=[5,40,12,-10,0,32,4,3,6,72]
```

For loop

Lets say i wanted to do odd/even exercise for all the numbers in this list. Technically we can do this , by value of x in the code that we had written 10 times. or writing 10 if-else blocks. turns out, given the for loop, we dont need to do that . lets see

```python
for element in x:
    if element%2==0:
        print(element,' is even')
    else:
        print(element,' is odd')
```

```
5  is odd
40  is even
12  is even
-10  is even
0  is even
32  is even
4  is even
3  is odd
6  is even
72  is even
```

- `element` here is known as index . it can be given any name like generic objects in python , `element` is not a fixed name
- x/value_list can be any list or ingeneral iterable
- body of the for loop is executed as many times as there are values in the value_list

here is another example

```python
cities=['Mumbai','London','Bangalore','Pune','Hyderabad']
for i in  cities:
    num_chars=len(i)
    print(i+ ':'+ str(num_chars))
```

```
Mumbai:6
London:6
Bangalore:9
Pune:4
Hyderabad:9
```

for loops can have multiple indices as well if the value list elements themselves are lists

```python
x=[[1,2],['a','b'],[34,67]]
for i,j in x:
    print(i,j)
```

```
1 2
a b
34 67
```

how ever for multiple indices to work , all the list element within the larger list need to same number of elements

While Loop

We have seen so far that for loops work with indices iterating over a value list. Many at times , we need to do this iterative operation basis a condition instead of a value list . We can do this using a while loop . Lets see an example

Lets say we want to remove all the occurences of a value from a lits. Remove function that we learned about , only removes first occurence . We can manually keep on running call to remove untill the all the occurences are removed or we can put this inside a while loop with the condition .

```
a=[3,3,4,4,43,3,3,3,2,2,45,67,89,3,3]
while 3 in a :
    a.remove(3)
```

```
a
```

> [4, 4, 43, 2, 2, 45, 67, 89]

List Comprehension

when we need to construct another list using an existing one doing some operation, usual way of doing that is to start with an empty list; iterate over the existing list, do some operation on the elements and append the result to empty list . like this :

```
x=[3,89,7,-90,10,0,9,1]
y=[]

for elem in x:
    sq=elem**2
    y.append(sq)
```

```
y
```

> [9, 7921, 49, 8100, 100, 0, 81, 1]

y now contains squares of all numbers in x. There is another way of achieving this where we write the for loop (shorter version of it) inside the empty list directly

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

```
y
```

> [9, 7921, 49, 8100, 100, 0, 81, 1]

Here the operation on each elem in x is `elem**2` this becomes element of the list automatically , without having to write an explicit for loop. This is called list comprehension . We can include conditional statement also in list comprehension . Lets first look at explicit for loop for the same .

```python
x= [ 4,-5,67,98,11,-20,7]
import math
y=[]
for elem in x:
    if elem>0:
        y.append(math.log(elem))

print(y)
```

> [1.3862943611198906, 4.204692619390966, 4.584967478670572, 2.3978952727983707, 1.9459101490553132]

```python
y=[math.log(elem) for elem in x if elem>0]
print(y)
```

> [1.3862943611198906, 4.204692619390966, 4.584967478670572, 2.3978952727983707, 1.9459101490553132]

Note: usage of `print` here is just to display this horizontally in comparison to vertically the way it naturally gets displayed .

you can include else statement as well

```python
logs=[math.log(num) if num>0 else 'out of domain' for num in x]
print(logs)
```

> [1.3862943611198906, 'out of domain', 4.204692619390966, 4.584967478670572, 2.3978952727983707, 'out of domain', 1.9459101490553132]

One caution here , dont always push for writing a list comprehension instead of a for loop. Purpose of list comprehension is to shorten the code at the same time, not compromising on the readability of a code. They provide no performance improvement on run time. Hence if list comprehension starts to become too complex , its better to go with an explicit for loop

Dictionaries , Set and Tuples

In this section we'll learn about other data structres in python, different from lists. They are not as widely used but do come with special properties which might be useful in special cases . We'll start our discussion with dictionaries . Dictionaries , unlike lists are unordered; meaning their elements do not have any index attached to them and can not be accessed like list element by passing element index/position. Elements of a dictionary is made up of key:value pair . Here is an example

```
my_dict=
{'name':'lalit','city':'hyderabad','locality':'gachibowli','num_vehicles':2
,3:78,4:[3,4,5,6]}
```

```
len(my_dict)
```

```
6
```

this dictionary has 6 elements (key:value pairs)

```
type(my_dict)
```

```
dict
```

special functions associated with dictionaries let you extract keys and values of dictionaries

```
my_dict.keys()
```

```
dict_keys(['name', 'city', 'locality', 'num_vehicles', 3, 4])
```

```
my_dict.values()
```

```
dict_values(['lalit', 'hyderabad', 'gachibowli', 2, 78, [3, 4, 5, 6]])
```

As you can see, numbers and strings alike can be keys of dictionaries . Now that we can not really use indices to access elements of dictionaries , how do we access them ? Using keys , as follows

```
my_dict['name']
```

```
'lalit'
```

```
my_dict[3]
```

```
78
```

As you can see, output is; values associated with the keys . A dictionary does not support duplicate keys , values however have no such restriction .

For adding a new key value pair , you simply need to do this :

```
my_dict['city']='delhi'
print(my_dict)
```

```
{'name': 'lalit', 'city': 'delhi', 'locality': 'gachibowli', 'num_vehicles': 2, 3: 78, 4: [3, 4, 5, 6]}
```

you can see that dictionary now has one more key:value pair; `'city': 'delhi'`

for removing an element , you can use keyword del

```python
del my_dict[4]
print(my_dict)
```

> {'name': 'lalit', 'city': 'delhi', 'locality': 'gachibowli', 'num_vehicles': 2, 3: 78}

Although rarely used , but you can do soemthing similar to list comprehension for dictionaries also

```python
d = {elem:elem**2 for elem in x}
d
```

> {4: 16, -5: 25, 67: 4489, 98: 9604, 11: 121, -20: 400, 7: 49}

here elem has become the key of dictionary and elem**2, the asscoaietd value with it

Next we'll look at sets. Sets are like mathematical sets. They are unordered collection of unique values . You can not have duplicate elements in a set , even if you forcefully try to . They are created just like lists , but using curly braces .

```python
x= {10, 2, 2, 4, 4, 4, 5, 60, 22, 76}
x
```

> {2, 4, 5, 10, 22, 60, 76}

to add and remove elements , there are special functions associated with sets

```python
x.add('python')
x
```

> {10, 2, 22, 4, 5, 60, 76, 'python'}

```python
x.remove(5)
```

As mentioned above , sets are unordered , meaning they do not have any indices associated with their elements and can not not be accessed the way we accessed lists .

```python
x[2]
```

```
TypeError                         Traceback (most recent call last)

 in
----> 1 x[2]
TypeError: 'set' object does not support indexing
```

How ever we can iterate through the elements

```python
for elem in x :
    print(elem)
```

```
2
4
10
76
python
22
60
```

Notice here that order of elements printed here is not the same as it was when we created the set . This can not be predetermined either . However will remain same once created .

They also have set operation function associated with them as we find with mathematical sets.

```
a= {12,  9, 13,  6,  4, 14, 17,  5,  1,  0}
b={2, 14, 16,  4, 19, 13, 12,  6}
```

```
a.union(b)
```

```
{0, 1, 2, 4, 5, 6, 9, 12, 13, 14, 16, 17, 19}
```

```
a.intersection(b)
```

```
{4, 6, 12, 13, 14}
```

```
a.difference(b)
```

```
{0, 1, 5, 9, 17}
```

```
b.difference(a)
```

```
{2, 16, 19}
```

```
a.symmetric_difference(b)
```

```
{0, 1, 2, 5, 9, 16, 17, 19}
```

- union : gives a new set containing elements from both a and b
- intersection : gives elements which are commong in both a and b
- difference : set1.difference(set2) will output those elements of set1 which are not there in set2
- symmetric_difference : gives elements from both a and b which are **not** common between them

Sets are used in practice when the usage doesnt require iterating over or indices . They are good for mainting collection of unique elements and checking presence of an element in sets is significantly faster in comparison to lists.

last data structure that we talk about is tuples . they are exactly same as list except one difference . You can not modify elements of tuples . They are created just like lists , except using small brackets .

```
x= ('a','b',45,98,0,-10,43)
```

tuples dont have any function like add/remove . In general there doesnt exist any way to modify a tuple once created (you can of course , change the type to list and then modify but thats beside the point)

```
x[4] = 99
```

```
TypeError                         Traceback (most recent call last)

 in
----> 1 x[4] = 99
TypeError: 'tuple' object does not support item assignment
```

Functions

A function is a block of code which only runs when it is called.You can pass data, known as parameters, into a function. A function can return data as a result (this is optional).Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes code reusable.

Lets say we are asked to come up with a program which, when provided a list, give us a dictionary containing unique elements of the list as keys and their counts as values

```
x= [2,2,3,4,4,4,4,2,2,2,2,2,4,5,5,5,6,6,1,1,1,3]
count_dict={}
for elem in x:
    if elem not in count_dict:
        count_dict[elem]=1
    count_dict[elem]=count_dict[elem]+1
```

```
count_dict
```

```
{2: 8, 3: 3, 4: 6, 5: 4, 6: 3, 1: 4}
```

Now if we need to do this multiple times in our project , we'll need to copy this entire bit of code whereever this is required. Instead of doing that we can make use of `functions` . We'll wrap this program in a function and when we need to use it, we'll have to write just one line, instead of writing the whole program.

function definitions start with keyword `def` , in the brackets which follow, we name the input which our program/function requires . In our case in discussion, there is a single input , a list . Functions can have multiple , even varying number of inputs . Lets convert , program written above into a function.

```python
def my_count(a):
    count_dict={}
    for elem in a:
        if elem not in count_dict:
            count_dict[elem]=1
        count_dict[elem]+=+1
    return(count_dict)
```

now i can call this function using a single line and it will return the output back .

```python
my_count(x)
```

> {2: 8, 3: 3, 4: 6, 5: 4, 6: 3, 1: 4}

```python
my_count([3,3,3,3,3,4,4,4,3,3,3,3,10,10,10,0,0,0,0,10,10,4,4,4])
```

> {3: 10, 4: 7, 10: 6, 0: 5}

Default arguments/parameters to a function

Functions that we have been using (other than the one that we just wrote), can take variable number of inputs. If we miss passing some value they work with default value given to them . Lets see how to create a function with default values for arguments . We are going to look at a function which takes input 3 numbers and returns a weighted sum.

```python
def mysum(x,y,z):

    s=100*x+10*y+z
    return(s)
```

it works fine if we pass 3 arguments while calling it

```python
mysum(1,2,3)
```

> 123

but if we try to call it with less than 3 numbers , it starts to throw error

```python
mysum(1,2)
```

```
TypeError                         Traceback (most recent call last)
```

```
 in
----> 1 mysum(1,2)
TypeError: mysum() missing 1 required positional argument: 'z'
```

while creating the function itself, we could have given default values for it work with in order to avoid this . We can use any value as default value (only thing ensure is that , it should make sense in the context of what function does)

```python
def mysum(x=1,y=10,z=-1):

    s=100*x+10*y+z
    return(s)
```

```python
mysum(1,2,3)
```

```
123
```

it still works as before when we are explicitly passing all the values . However if we chose to pass lesser number of inputs , instead of throwing errors , it makes use of default values provided by us to each argument.

```python
mysum(1,2)
```

```
119
```

Last thing , about the functions; when we are passing the arguments while calling the function; they are assigned to various objects in the functions in the sequence in which they are passed. This can be changed if we name our arguments while we pass them. Sequence doesnt matter in that case.

```python
mysum(z=7,x=9)
```

```
1007
```

Classes

what is a class? Simply a logical grouping of data and functions . What do we mean by "logical grouping"? Well, a class can contain any data we'd like it to, and can have any functions (methods) attached to it that we please. Rather than just throwing random things together under the name "class", we try to create classes where there is a logical connection between things. Many times, classes are based on objects in the real world (like Customer or Product or Points).

Regardless, classes are a way of thinking about programs. When you think about and implement your system in this way, you're said to be performing Object-Oriented Programming. "Classes" and "objects" are words that are often used interchangeably.

Lets look at a use case which will convince you that when using a class to define a logical grouping makes your life easier

Lets say I want to keep track of customer records. Customers have lets say three attributes associated with them : name , account_balance and account_number. I will need to create three different objects for each customer and then ensure that I dont end up mixing those objects with another customer's details. I'll rather write a class for customers.

```python
class customer():

    # the first function is __init__ , with double underscore
    # this is used to set attributes of object of the class customer when it gets created
    # we can also put data here which can be used by any object of the class customer
    # can also be used by other methods/functions contained in the class

    # self here is a way to refer to object of the same class and its attribute

    def __init__(self,name,balance,ac_num):

        self.Name=name
        self.AC_balance=balance
        self.AcNum = ac_num
```

now i just need to create one object for customers with these attributes and can seamlessely avoid mixing up objects for these attributes across customers

```python
c1=customer('lalit',2000,'A3124')
```

```python
c1.Name
```

> 'lalit'

```python
c1.AC_balance
```

> 2000

```python
c1.AcNum
```

> 'A3124'

I can also attach methods/function to this class which will be available to object of this class only

```python
class customer():

    # the first function is __init__ , with double underscore
    # this is used to set attributes of object of the class customer when it gets created
    # we can also put data here which can be used by any object of the class customer
    # can also be used by other methods/functions contained in the class

    # self here is a way to refer to object of the same class and its attribute

    def __init__(self,name,balance,ac_num):

        self.Name=name
        self.AC_balance=balance
        self.AcNum = ac_num

    def withdraw(self,amount):

        self.AC_balance -= amount

    def get_account_number(self):

        print(self.AcNum)
```

```python
c1=customer('lalit',2000,'A3124')
```

```python
c1.AC_balance
```

```
2000
```

```python
c1.withdraw(243)
```

```python
c1.AC_balance
```

```
1757
```

```python
c1.get_account_number()
```

```
A3124
```

As a final note to this discussion; If idea of classes seemed too daunting, you can safely skip this. You will never need to write to your own classes until you start to work with some bigger projects which implement their own algorithm or complex data processing routine.