----python

- any operation with float and integer will always return float values.
- **boolean values** such as true or false should always start with capital letters
- string can be executed in single , double or triple quotes(',",'")
- strings are immutable which means that any changes made in strings are not possible
- In python we don't use && || or ! instead we use and or not in words
- Lists are same as arrays but contains heterogeneous arrays
- string H E L L O
  0 1 2 3 4
  -5-4-3-2-1
- strings are collections of characters which have two types of indexing: normal indexing and reverse indexing.
- normal indexing starts from 0 from the left side of the string whereas the reverse indexing starts from the rights side of the string.
- When we are using sort function we need homogeneous function

id(variable_name) is used to get the memory address of the normal variable
Range[start,stop,steps] for loop range {steps is optional}

TASK:

- Input : knowledge. 'egdelwonk'
  Output : edgewnok.
  s = "knowledge"
  S[::-1]
  ---
  i=input("enter the string :")
  j=len(i)//2 # integer division //
  print('length ',j)
  a=i[:j] # Slicing the input
  print('Slicing ',a) # contains starting letters
  b="" # Empty varible

  for y in a: # y is the range

```
    b=y+b
    print("for loop",b)
print("value after the loop",b)
i=i[j:]+b
print(i)
```

- Hello to yello replace h with y

```
 s='hello'
id(s)
 s='y'+s[1:]
 print(s)
```

- Pandas

Pandas:

Pandas is a python library which gives us perfect set of tools for data analysis
If we are using data in python we are going to need pandas
Data analysis
Data Science
Machine Learning

With pandas we can load ,prepare ,manipulate model and analyze data
We can join data
We can merge data
We can reshape data
Support different formats of file formats csv , excel
We can take different databases and put it together and analyze the data
High performance data analysis tool

- Series { represented in 1 dimensional array } --- list
- Data frame { represented in 2 dimensional array } ----- list or dictionary or other dataframe
- Panel { represented in multidimensional array }-------

Data , major access , minor access
Rows & columns

Which all roll around called a Data frame.

.describe()
.drop( ) delete the data

.value_counts() counts of the data
.mean() cal the mean
.groupby() grouping the col
.head()
.plot()
.index() date & time
.Series()
.DataFrame() most efficient in data structure to do the analysis on the data sets
.panel(data , major access, minor access, dtype)

Questions

- **What is the difference between lower method and casefold method?**
  The casefold() method removes all case distinctions present in a string. It is used for caseless matching, i.e. ignores cases when comparing. For example, the German lowercase letter ß is equivalent to ss . However, since ß is already lowercase, the lower() method does nothing to it.
- **What is the logic to be used to ignore the case of the string?**
  equalsIgnorecase()

- **Which is the method used to convert the string as a list?**
  split()
- **What is the use of tkinter lib?**
  GUI INTERFACE TO INTERACT WITH THE user
- **What is the precedence of the arithmetic operations?**
  PEMDAS
- **What is the use of pass ,break and continue with code examples?(Search an object & use break)**
  Pass. Let's start with something relatively easy (but not straightforward). Nothing happens when a pass is executed. ...

  Break. The **break** statement allows you to leave a for or while loop prematurely. ...

  Continue. The **continue** statement ignores the rest of the statements inside a loop, and continues with the next iteration.

Jothisharma - Linked-Sin

Python interprets any non zero values as True
None and 0 is interpreted as false

Strings

Strings are used to record the text information such as name. In Python, Strings act as “Sequence” which means Python tracks every element in the String as a sequence. This is one of the important features of the Python language.

- We can use a : to perform *slicing* which grabs everything up to a designated point. For example:
- In Python, Objects have built-in methods which means these methods are functions present inside the object (we will learn about these in much more depth later) that can perform actions or commands on the object itself.
- Methods can be called with a period followed by the method name. Methods are in the form: object.method(parameters)
- **In python everything is an object**

Questions

- **Learn the memory module for normal variable & list?**

- **What is the precedence of the arithmetic operations?**
  (20+3)+12+8/4*3

(Search an object & use break)

List
Flag
Logic- have to use break
If flag is true{ if the position of the element is present}
Else{did not present}

```
lst=[]                                          #empty string to take variables

num=int(input("enter the size of the list :"))  #to take the size of the array

for n in range(num):                            #assigning range
   number = int(input("Enter the elements :"))  #input of the array
   lst.append(number)                           #making the list

x = int(input("Enter the search element :"))    # search element

i = 0                                           # i as empty
flag = False                                    #initial value of flag is false
while i < len(lst):                             #searching through the index from the list
   if lst[i]==x:                                #comparing
      flag=True                                 #if the element exists in the list
      Break                                     #breaks
   i = i + 1                                    #or else moves to the next index
if flag ==1:                                    # if flags is True we have found the element
   print('was found at index',i)
else:
   print('was not found')
```

String Indexing:

S[0] - gives the 1st element of the string.
S[1]- gives 2nd letter of the string
S[2]- gives third letter of the string
S[1:]- from the location 1 everything.
S[:3]- everything till index position 3
S[:] - everything
S[-1]- last element of the string
S[:-1]- everything but the last element

Index and slice notation is used to grab elements of a sequenec by a specified step size (where in 1 is the default size). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

S[::1]-it shows everything
S[::2]-it skips 2 elements from the string
S[::-1]- this will print the string reverse

String properties:

S = hello world

S+ concatenate me'
s= S + 'concatenate me
print(s)
Hello world concatenate me

Multiple elements z*10
Zzzzzzzzzzzzzzzzzzz 10 times

Basic built in stings

S.upper()
S.lower()
S.split()
S.split('o') splits where the element "o" is
S.find() it finds the element in the string posts ti position
S.count() counts the elements which we declare
S.center(40,z**)** this command will put the string in the middle covering wil z
expandtabs() will expand tab notations \t into spaces. Let's see an example to understand the concept.
'hello\thi'.expandtabs() where \t is will be the place where you'll get the tab space
S ='hello'
S.isalnum() checks that the string has alphanumeric values
S.isalpha() checks that the string has alphabet values
S.islower() check that the string is all in lowercase.
S.isspace() checks that the string has space
S.istitle() checks that the string is Title
S.isuper() checks the string is uppercase.
S.endswith() checks the element is equal to the end of the string are both equal

Regular expression

S.split('e') Splits the string wherever the element "e" is in the string

Lists

Print Formatting:

“Insert Another String with curly brackets:{}”,format({‘ The Inserted String’})
The curly brackets will be replaced by the Formatted text

o

lists

Earlier, while discussing the introduction to strings we introduced the concept of a sequence in Python. In Python, Lists can be considered as the most general version of a "sequence". Unlike strings, they are mutable which means the elements inside a list can be changed!
Lists are constructed with brackets [] and commas separating every element in the list.
Let's go ahead and see how we can construct lists!

---

my_list = [1,2,3]
We just created a list of integers, but lists can actually hold different object types. For example:
my_list = ['A srting',1,2,3]

my_list

....
len() **#length of the list**
 # Grab index 1 and everything past it
My_list[1:]
#We can also use "+" to concatenate lists, just like we did for Strings.
my_list + [6]
 # Make the list double
my_list *2

Basic List Methods

If you are familiar with another programming language, start to draw parallels between lists in Python and arrays in other languages. There are two reasons which tell us why the lists in Python are more flexible than arrays in other programming languages: a. They have no fixed size (which means we need not to specify how big the list will be) b. They have no fixed type constraint Let's go ahead and explore some more special methods for lists:

If **l** is a list
 # Append
l.append('append me!!')

 # Pop off the 0 indexed item
l.pop(0)
 # Use reverse to reverse order (this is permanent!)
new_list.reverse()
 # Use sort to sort the list (in this case alphabetical order, but for numbers it will go ascending)
new_list.sort()

Nesting Lists

Nesting Lists is one of the great features in Python data structures. Nesting Lists means we can have data structures within data structures.For example: A list inside a list.Let's see how Nesting lists works!
 # Let's make three lists
a=[1,2,3,4]
b=[6,7,8,9]
c=[10,11,12,13]
 # Make a list of lists to form a matrix
matrix = [a,b,c]
 # Grab first item in matrix object
matrix[0]
 # Grab first item of the first item in the matrix object
matrix[0][0]

List Comprehensions

Python has an advanced feature called list comprehensions which allows for quick construction of lists.Before we try to understand list comprehensions completely we need to understand "for" loops.So don't worry if you don't completely understand this section, and feel free to just skip it since we will return to this topic later.Here are few of our examples which helps you to understand list comprehensions.
 # Build a list comprehension by deconstructing a for loop within a []
first_col = [row[3] for row in matrix]

Advanced Lists

In this series of lectures, we will be diving a little deeper into all the available methods in a list object. These are just methods that should be encountered without some additional exploring. Its pretty likely that you've already encountered some of these yourself! Lets begin!

#append:Definitely, You have used this method by now, which merely appends an element to the end of a list:
l.append(4)

#count:We discussed this during the methods lectures, but here it is again. count() takes in an element and returns the number of times it occurs in your list:

```
l.count(2)    #number of elements present in the list
```

#extend¶Many times people find the difference between extend and append to be unclear. So note that,

append: Appends object at end

```
x = [1,2,3]
x.append([4,5])
print(x)
```

#extend: extends list by appending elements from the iterable

```
x=[1,2,3]
x.extend([4,5])
print(x)
```

#Note how extend, append each element in that iterable. That is the key difference.

#index:index returns the element placed as an argument. Make a note that if the element is not in the list then it returns an error.

```
l.index(2)
```

#insert:Two arguments can be placed in insert method.Syntax: insert(index,object)This method places the object at the index supplied. For example:

```
# Place a letter at the index 2
l.insert(2,'two')
```

#pop:You most likely have already seen pop(), which allows us to "pop" off the last element of a list.

```
ele = l.pop()
```

#remove:The remove() method removes the first occurrence of a value. For example:

```
l.remove(2) #it removes the specific element from the table.
```

#reverse:As the name suggests, reverse() helps you to reverse a list. Note this occurs in place! Meaning it affects your list permanently.

```
l.reverse()       #it reverse the elements of the list
```

#sort:sort will sort your list in place:

```
l.sort()          #it sort the elements in ascending order
```

Tuples

In Python, tuples are similar to lists but they are immutable i.e. they cannot be changed. You would use the tuples to present data that shouldn't be changed, such as days of week or dates on a calendar.

In this section, we will get a brief overview of the following key topics:
1.) Constructing Tuples
2.) Basic Tuple Methods
3.) Immutability

4.) When to Use Tuples

You'll have an intuition of how to use tuples based on what you've learned about lists. But, Tuples work very similar to lists but the major difference is tuples are immutable.

Constructing Tuples

The construction of tuples use () with elements separated by commas where in the arguments will be passed within brackets. For example:

 # Can create a tuple with mixed types

```
t = (1,2,3,4)
```

 # Check len just like a list

```
len(t)
```

 # Can also mix object types

```
t = ('one',2)
```

 # Show

```
t
```

 # Use indexing just like we did in lists

```
t[1]
```

Basic Tuple Methods

Tuples have built-in methods, but not as many as lists do. Let's see two samples of tuple built-in methods:

 # Use .index to enter a value and return the index

```
t.index('one')
```

 # Use .count to count the number of times a value appears

```
t.count('one')
```

Immutability

As tuples are immutable, it can't be stressed enough and add more into it. To drive that point home:

Because tuples are immutable they can't grow. Once a tuple is made we can not add to it.

```
t.append([3])
```

When to use Tuples

You may be wondering, "Why bother using tuples when they have a few available methods?"

Tuples are not used often as lists in programming but are used when immutability is necessary. While you are passing around an object and if you need to make sure that it does not get changed then tuple becomes your solution. It provides a convenient source of data integrity.

You should now be able to create and use tuples in your programming as well as have a complete understanding of their immutability.

Sets

Sets are an unordered collection of *unique* elements which can be constructed using the set() function.

Let's go ahead and create a set to see how it works.

#create a set

```
x = set()
```

 # We add to sets with the add() method

```
x.add(1)
```

#Show

```
x
```

Note that the curly brackets do not indicate a dictionary! Using only keys, you can draw analogies as a set being a dictionary.

We know that a set has an only unique entry. Now, let us see what happens when we try to add something more that is already present in a set?

In [ ]:

 # Add a different element

```
x.add(2)
```

#Show

```
x
```

 # Try to add the same element

```
x.add(1)
```

#Show

```
x
```

Notice, how it won't place another 1 there as a set is only concerned with unique elements! However, We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

 # Create a list with repeats

```
l=[1,1,2,2,3,4,5,6,1,1]
```

 # Cast as set to get unique values

```
set(l)
```

Dictionaries

We have learned about "Sequences" {strings} in the previous session. Now, let's switch the gears and learn about "mappings" in Python. These dictionaries are nothing but hash tables in other programming languages.

In this section, we will learn briefly about an introduction to dictionaries and what it consists of:

1.) Constructing a Dictionary

2.) Accessing objects from a Dictionary

3.) Nesting Dictionaries

4.) Basic Dictionary Methods

Before we dive deep into this concept, let's understand what are Mappings?

Mappings are a collection of objects that are stored by a "key". Unlike a sequence, mapping store objects by their relative position. This is an important distinction since mappings won't retain the order since they have objects defined by a key.

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

 # Make a dictionary with {} and : to signify a key and a value

```
my_dict = {'key1':'value1','key2':'value2'}
```

 # Call values by their key

```
my_dict['key2']
```

Note that dictionaries are very flexible in the data types they can hold. For example:

```
my_dict={'key1':123,'key2':[12.23,33],'key3':['item1','item2','item3']}
```

#Let's call items from the dictionary

```
my_dict['key3']
```

 # Can call an index on that value

```
my_dict['key3'][0]
```

#Can then even call methods on that value

```
my_dict['key3'][0].upper()
```

We can effect the values of a key as well. For instance:

```
My_dict['key1']
```

 # Subtract 123 from the value

```
my_dict['key1'] = my_dict['key1']-123
```

#Check

```
my_dict['key1']
```

Note, Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could also use += or -= for the above statement. For example:

 # Set the object equal to itself minus 123

```
my_dict['key1'] -=123
```

```
my_dict['key1']
```

We can also create keys by assignment. For instance if we started off with an empty dictionary, we could continually add to it:

 # Create a new dictionary

```
d = {}
```

 # Create a new key through assignment

```
d['animals']= 'Dog'
```

 # Can do this with any object

```
d['answer'] = 42
```

Nesting with Dictionaries

Let's understand how flexible Python is with nesting objects and calling methods on them. let's have a look at the dictionary nested inside a dictionary:

 # Dictionary nested inside a dictionary nested inside a dictionary

```
d = {'key1':{'nestkey':{'subnestkey':'values'}}}
```

That's the inception of dictionaries. Now, Let's see how we can grab that value:

 # Keep calling the keys

```
d['key1']['nestkey']['subnestkey']
```

```
d['key1']#['nestkey']['subnestkey']
```

```
d['key1']['nestkey']#['subnestkey']
```

A few Dictionary Methods

There are a few methods we can call on a dictionary. Let's get a quick introduction to a few methods:

 # Create a typical dictionary

```
d = {'key1':1,'key2':2,'key3':3}
```

 # Method to return a list of all keys

```
d.keys()
```

 # Method to grab all values

```
d.values()
```

 # Method to return tuples of all items (we'll learn about tuples soon)

```
d.items()
```

Type *Markdown* and LaTeX:

α

2

α2

Dictionary Comprehensions

Just like List Comprehensions, Dictionary Data Types also support their own version of comprehension for quick creation. It is not as commonly used as List Comprehensions, but the syntax is:

```
print(list(range(10)))
```

```
{x:x**2 for x in range(10)} #single star multiply ,,double star means power
```

One of the reasons is the difficulty in structuring the key names that are not based on the values.

Function:

https://docs.python.org/2/library/functions.html

Introduction to Functions

What is a function in Python and how to create a function?
Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.
So what is a function?
A function groups a set of statements together to run the statements more than once. It allows us to specify parameters that can serve as inputs to the functions.
Functions allow us to reuse the code instead of writing the code again and again. If you recall strings and lists, remember that len() function is used to find the length of a string. Since checking the length of a sequence is a common task, you would want to write a function that can do this repeatedly at command.
Function is one of the most basic levels of reusing code in Python, and it will also allow us to start thinking of program design.

def Statements

Now, let us learn how to build a function and what is the syntax in Python.

The syntax for def statements will be in the following form:

```
def namee_of_the_function(arg1,arg2):



    # Do stuff here

    #return desired result
```

We begin with def then a space followed by the name of the function. Try to keep names relevant and simple as possible, for example, len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (such as len).

Next, comes the number of arguments separated by a comma within a pair of parentheses which acts as input to the defined function, reference them and the function definition with a colon.

Here comes the important step to indent to begin the code inside the defined functions properly. Also remember, Python makes use of *whitespace* to organize code and a lot of other programming languages do not do this.

Next, you'll see the doc-string where you write the basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these doc-strings by pressing Shift+Tab after a function name. It is not mandatory to include docstrings with simple functions, but it is a good practice to put them as this will help the programmers to easily understand the code you write.

After all this, you can begin writing the code you wish to execute.

The best way to learn functions is by going through examples. So let's try to analyze and understand examples that relate back to the various objects and data structures we learned.

Example 1: A simple print 'hello' function¶

```
def say_hello():
    print('hello')
#Call the function
say_hello()
```

Example 2: A simple greeting function¶

Let's write a function that greets people with their name.

```
def greeting(name):

    print(' Hello %s'%name)

    print('How you doing %s' %name)

----

greeting('bob')
```

Using return

Let's see some examples that use a return statement. Return allows a function to "return" a result that can then be stored as a variable, or used in whatever manner a user wants.

Example 3: Addition function¶

```
def add_num(num1,num2):
    return num1 + num2
#calling of the function
add_num(2,6)

# Can also save as variable due to return
result = add_num(4,5)
#Assigning the value of the two functions

result
```

What happens if we input two strings?

```
# adding two strings will be concat between those two strings
print('bobby\t', 'markus')
```

In Python we don't declare variable types, this function could be used to add numbers or sequences together! Going forward, We'll learn about adding in checks to make sure a user puts in the correct arguments into a function.

Let's also start using *break*,*continue*, and *pass* statements in our code. We introduced these during the whole lecture.
Now, let's see a complete example of creating a function to check if a number is prime (a common interview exercise).

We know a number is said to be prime if that number is only divisible by 1 and itself. Let's write our first version of the function to check all the numbers from 1 to N and perform modulo checks.

 # Find the prime number:

```
def is_prime(num):
    '''
    Naive method of checking for prime
    '''
    for n in range(2,num):
        if num % n == 0:
            print('not prime')
            break
    else:  # If never mod zero, then prime
            print(' its a prime number')
```

 # calling of the function

```
is_prime(70)
```

Note: That is how we break the code after the print statement! We can actually improve this by only checking the square root of the target number, also we can disregard all even numbers after checking for 2. We'll also switch to returning a boolean value to get an example of using return statements:

#by using the library function math execute the program for the prime number

```
import math
def is_prime(num):
    '''
    A Better method of checking for prime
    '''
    if num % 2 ==0 and num > 2 :
        return False
    for i in range(3, int(math.sqrt(num)) + 1 , 2):
        if num % i == 0:
            print(i)
            return False
    return True
```

 # calling the function

```
is_prime(77)
```

Iterators and Generators¶

In this section, you will be learning the differences between iterations and generation in Python and also how to construct our own generators with the "yield" statement. Generators allow us to generate as we go along instead of storing everything in the memory.

We have learned how to create functions with "def" and the "return" statement. In Python, the Generator function allows us to write a function that can send back a value and then later resume to pick up where it was left. It also allows us to generate a sequence of values over time. The main difference in syntax will be the use of a **yield** statement.

In most aspects, a generator function will appear very similar to a normal function. The main difference is when a generator function is called and compiled they become an object that supports an iteration protocol. That means when they are called they don't actually return a value and then exit, the generator functions will automatically suspend and resume their execution and state around the last point of value generation.

The main advantage here is "state suspension" which means, instead of computing an entire series of values upfront, the generator functions can be suspended. To understand this concept better let's go ahead and learn how to create some generator functions.

```
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num**3

# calling the function
for x in gencubes(10):
    print (x)
```

Great! Since we have a generator function we don't have to keep track of every single cube we created.

Generators are the best for calculating large sets of results (particularly in calculations that involve loops themselves) when we don't want to allocate memory for all of the results at the same time.

Let's create another sample generator which calculates fibonacci numbers:

```
def genfibon(n):
    '''
    Generate  a fibonacci number sequence up to n
    '''
    a=1
    b=1
    for i in range(n):
        yield a
        a,b = b,a+b
```

 # Calling the function of fibonacci

```
for num in genfibon(10):
    print(num)
```

What if this was a normal function, what would it look like?

```
def fibon(n):
    a =1
    b =1
    output = []

    for i in range(n):
        output.append(a)
        a,b=b,a+b
    return output
```

 # calling the function

```
fibon(10)
```

Note, if we call some huge value of "n", the second function will have to keep track of every single result. In our case, we only care about the previous result to generate the next one.

next() and iter() built-in functions

A key to fully understand generators is the next() and the iter() function.

The next function allows us to access the next element in a sequence. Let's check how it works.

```
def simple_gen():
    for x in range(3):
        yield x
```

 # Assign simple_gen

```
g= simple_gen()
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

After yielding all the values next() caused a StopIteration error. This error informs us that all the values have been yielded.

You might be wondering why we don't get this error while using a for loop? The "for loop" automatically catches this error and stops calling next.

Let's go ahead and check out how to use iter(). You remember that strings are iterable:

```
s = 'hello'
```

#Iterate over string

```
for l in s:
```

```
    print(l)
```

But that doesn't mean the string itself is an *iterator*! We can check this with the next() function:

```
next(s)
```

This means that a string object supports iteration, but we can not directly iterate over it as we could with a generator function. The iter() function allows us to do just that!

```
s_iter =iter(s)
next(s_iter)
next(s_iter)
next(s_iter)
next(s_iter)
```

map()¶

The map() is a function that takes in two arguments:

1. A function
2. A sequence iterable.

In the form: map(function, sequence)

The first argument is the name of a function and the second a sequence (e.g. a list). map() applies the function to all the elements of the sequence. It returns a new list with the elements changed by the function.

When we went over list comprehension we created a small expression to convert Fahrenheit to Celsius. Let's do the same here but use map.

We'll start with two functions:

```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)
```

```
temp = [0, 22.5, 40, 100]
```

Now let's see map() in action:

```
F_temps = list(map(fahrenheit, temp))
```

#Show

```
F_temps
```

 # Convert back

```
list(map(celsius, F_temps))
```

In the example above, we haven't used a lambda expression. By using lambda, it is not necessary to define and name fahrenheit() and celsius() functions.

```
list(map(lambda x: (5.0/9)*(x - 32), F_temps))
```

Map is more commonly used with lambda expressions since the entire purpose of a map() is to save effort on creating manuals for loops.
map() can be applied to more than one iterable. The iterables must have the same length.

For instance, if we are working with two lists-map() will apply its lambda function to the elements of the argument lists, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the nth index is reached.

For example, let's map a lambda expression to two lists:

```
a = [1,2,3,4]

b = [5,6,7,8]

c = [9,10,11,12]

list(map(lambda x,y:x+y, a,b))

# Now all three lists

list(map(lambda x,y,z:x+y+z, a,b,c))
```

In the above example, the parameter 'x' gets its values from the list 'a', while 'y' gets its values from 'b' and 'z' from list 'c'. Go ahead and create your own example to make sure that you completely understand mapping more than one iterable.

reduce()

The function reduce(function, sequence) continually applies the function to the sequence. It then returns a single value.

If seq = [s1, s2, s3, ... , sn], calling reduce(function, sequence) works like this:

- At first the first two elements of sequence will be applied to function, i.e. func(s1,s2)
- The list on which reduce() works looks like this: [function(s1, s2), s3, ... , sn]
- In the next step the function will be applied on the previous result and the third element of the list, i.e. function(function(s1, s2),s3)
- The list looks like: [function(function(s1, s2),s3), ... , sn]
- It continues like this until just one element is left and return this element as the result of reduce()

Let's see an example:

```
from functools import reduce

lst = [47,11,42,58]
```

```
reduce(lambda x,y:x+y,lst)
```

Let's look at a diagram to get a better understanding of what is going on here:

Note how we keep reducing the sequence until a single final value is obtained. Let's see another example:

#Find the maximum of a sequence (This already exists as max())

```
max_find = lambda a,b: a if (a>b)else b
```

```
print(lst)
```

#Find max

```
reduce(lambda a,b: a if (a>b)else b,lst)
```

filter

The function filter(function, list) offers a convenient way to filter out all the elements of an iterable, for which the function returns "True".

The function filter(function(),l) needs a function as its first argument. The function needs to return a Boolean value (either True or False). This function will be applied to every element of the iterable. Only if the function returns "True" will the element of the iterable be included in the result.

Let's see some examples:

#First let's make a function

```
def even_check(num):
    if num%2 == 0:
        return True
```

Now let's filter a list of numbers. Note that putting the function into a filter without any parenthesis might feel strange, but keep in mind that functions are objects as well.

```
lst = range(20)
```

```
print(list(lst))
```

```
list(filter(even_check,lst))
```

filter() is more commonly used with lambda functions, this is because we usually use filter for a quick job where we don't want to write an entire function. Let's repeat the example above using a lambda expression:

```
list(filter(lambda x: x%2==0,lst))
```

Object Oriented Programming and File I/O

Object Oriented Programming (OOP) is a programming paradigm that allows abstraction through the concept of interacting entities. This programming works contradictory to conventional model and is procedural, in which programs are organized as a sequence of commands or statements to perform.

We can think of an object as an entity that resides in memory, has a state and it's able to perform some actions.

More formally objects are entities that represent **instances** of a general abstract concept called **class**. In Python, "attributes" are the variables defining an object state and the possible actions are called "methods".

In Python, everything is an object also classes and functions.

1 How to define classes

1.1 Creating a class

Suppose we want to create a class, named Person, as a prototype, a sort of template for any number of 'Person' objects (instances).
The following python syntax defines a class:

```
class ClassName(base_classes):

        statements
```

Class names should always be uppercase (it's a naming convention).
Say we need to model a Person as:

- Name
- Surname
- Age

```
class Person:
   pass
jhon_doe = Person()
jhon_doe.name = "Alex"
jhon_doe.surname = "Baldwin"
jhon_doe.year_of_birth = 1985


print(jhon_doe)

print("%s %s was born in %d,"%(jhon_doe.name,jhon_doe.surname,jhon_doe.year_of_birth))
```

The following example defines an empty class (i.e. the class doesn't have a state) called *Person* then creates a *Person* instance called *john_doe* and adds three attributes to *john_doe*. We see that we can access objects attributes using the "dot" operator.

This isn't a recommended style because classes should describe homogeneous entities. A way to do so is the following

```
class Person:
    def __init__(self, name, surname, year_of_birth):
        self.name = name
        self.surname = surname
        self.year_of_birth = year_of_birth
```

Is a special *Python* method that is automatically called after an object construction. Its purpose is to initialize every object state. The first argument (by convention) **self** is automatically passed either and refers to the object itself.

In the preceding example, `__init__` adds three attributes to every object that is instantiated. So the class is actually describing each object's state.

We cannot directly manipulate any class rather we need to create an instance of the class

```
alec = Person("Alec","Baldwin",1958)
print(alec)
print("%s %s was born on %d," %(alec.name, alec.surname, alec.year_of_birth))
```

We have just created an instance of the Person class, bound to the variable `alec`.

1.2 Methods

```
class Person:
def __init__(self, name, surname, year_of_birth): #__init__ is an initialization of class
self.name = name
        self.surname = surname
        self.year_of_birth = year_of_birth
    def age(self, current_year):
        return current_year - self.year_of_birth
  def __str__(self):
        return "%s %s was born in %d ,"%(self.name, self.surname,self.year_of_birth)

alec = Person("Alec","Baldwin",1958)
#alec = Person("bob","mady",1969)
print(alec)
print(alec.age(2021))
```

```
#Class method another example
class Person:                          # inside the class is methods
    def __init__(self, name, surname, year_of_birth):
        self.name = name
        self.surname = surname
        self.year_of_birth = year_of_birth
    def age(self, current_year):
        return current_year - self.year_of_birth

    def __str__(self):
        return "%s %s was born in %d ,"%(self.name, self.surname,self.year_of_birth)

alec = Person("Mark","Zuckerberg",1988)

print(alec)
print(alec.age(2021))
```

We defined two more methods: age and `__str__`. The latter is once again a special method that is called by Python when the object has to be represented as a string (e.g. when it has to be printed). If the `__str__` method isn't defined the **print** command shows the type of object and its address in memory. We can see that in order to call a method we use the same syntax for attributes (**instance_name.instance _method**).

2 Inheritance

Once a class is defined it models a concept. It is useful to extend a class behavior to model a less general concept. Say we need to model a Student, but we know that every student is also a Person so we shouldn't model the Person again but inherit from it instead.

```
class Student(Person):
    def __init__(self, student_id, *args, **kwargs):
        super(Student, self).__init__(*args, **kwargs)
        self._student_id = student_id

charlie = Student(1, 'Charlie', 'Brown',2006)
print(charlie)
print(type(charlie))
print(isinstance(charlie, Person))
print(isinstance(charlie, object))
```

Charlie now has the same behavior of a Person, but his state has also a student ID. A Person is one of the base classes of Student and Student is one of the sub classes of Person. Be aware that a subclass knows about its superclasses but the converse isn't true.

A sub class doesn't only inherit from its base classes, but from its base classes too, forming an inheritance tree that starts from an object (every class base class).

    super(Class, instance)

is a function that returns a proxy-object that delegates method calls to a parent or sibling class of type. So we used it to access Person's `__init__`.

2.1 Overriding methods

Inheritance allows to add new methods to a subclass but often is useful to change the behavior of a method defined in the superclass. To override a method just define it again.

```
class Student(Person):
    def __init__(self , student_id ,*args, **kwargs):
        super(Student, self).__init__(*args, **kwargs)
        self._student_id = student_id

    def __str__(self):
        return super(Student, self).__str__() + "And has ID:%d"%self._student_id

charlie = Student(1,"Charlie","Brown",1963)
print(charlie)
```

We defined `__str__` again overriding the one written in Person, but we wanted to extend it, so we used super to achieve our goal.

3 Encapsulation

Encapsulation is another powerful way to extend a class which consists of wrapping an object with a second one. There are two main reasons to use encapsulation:

- Composition
- Dynamic Extension

3.1 Composition

The abstraction process relies on creating a simplified model that remove useless details from a concept. In order to be simplified, a model should be described in terms of other simpler concepts. For example, we can say that a car is composed by:

- Tyres
- Engine
- Body

And break down each one of these elements in simpler parts until we reach primitive data.

4 Polymorphism and DuckTyping

`Python` uses dynamic typing which is also called duck typing. If an object implements a method you can use it, irrespective of the type. This is different from statically typed languages, where the type of a construct needs to be explicitly declared. Polymorphism is the ability to use the same syntax for objects of different types:

```
def summer(a,b):
    return a + b
print(summer(1,1))
print(summer(["a","b","c"],["d","e"]))
print(summer("abra","cadabra"))
```

5 How long should a class be?

There is an Object Oriented Programming (OOP) principle called Single Responsibility Principle (SRP) and it states: "A class should have one single responsibility" or "A class should have only one reason to change".

If you come across a class which doesn't follow the SRP principle, you should split it. You will be grateful to SRP during your software maintenance.

Files

Python uses file objects to interact with the external files on your computer. These file objects cab be of any file format on your computer i.e. can be an audio file, a text file, emails, Excel documents, etc. Note that You will probably need to install certain libraries or modules to interact with those various file types, but they are easily available. (We will cover downloading modules later on in the course).

Python has a built-in open function that allows us to open and play with basic file types. First we will need a file though. We're going to use some iPython magic to create a text file!

iPython Writing a File

```
%%writefile test.text
Hello, this is a quick supperman file
```

Python Opening a file

We can open a file with the open() function. This function also takes in arguments (also called parameters). Let's see how this is used:

In [10]:

```
# Open the text.txt we made earlier
my_file = open('test.text')
# We can now read the file
my_file.read()
# But what happens if we try to read it again?
my_file.read()
```

This happens because you can imagine the reading "cursor" is at the end of the file after having read it. So there is nothing left to read. We can reset the "cursor" like this:

```
# Seek to the start of file (index 0)
my_file.seek(0)
# Now read again
my_file.read()
```

In order to not have to reset every time, we can also use the readlines method. Use caution with large files, since everything will be held in memory. We will learn how to iterate over large files later in the course.

```
    # Seek to the start of file (index 0)

  my_file.seek(0)

# Readlines returns a list of the lines in the file.

my_file.readlines()
```

Writing to a File

By default, using the open() function will only allow us to read the file, we need to pass the argument 'w' to write over the file. For example:

```
# Add the second argument to the function, 'w' which stands for write
my_file =open('test.text','w+')
# Write to the file
my_file.write('This is a new text & line')
# Seek to the start of file (index 0)
my_file.seek(0)
# Read the file
my_file.read()
```

Iterating through a File¶

Let's get a quick preview of a for loop by iterating over a text file. First, let's make a new text file with some iPython Magic:

```
%%writefile mop.txt
First Line
Second Line
```

Now we can use a little bit of flow to tell the program to for through every line of the file and do something:

```
for line in open('mop.txt'):
    print(line)

# Pertaining to the first point above

for Preetham in open('test.txt'):

    print(Preetham)
```

StringIO

The StringIO module implements an in-memory filelike object. This object can then be used as input or output to most functions that would expect a standard file object.

The best way to show this is by example:

```
from io import StringIO
# Arbitrary String
message = 'This is just a normal string '
# Use StringIO method to set as file object
f = StringIO(message)
```

Now we have an object *f* that we will be able to treat just like a file. For example:

```
f.read()
```

We can also write to it

```
f.write('Second Line written to the file like object')
# Reset cursor just like you would a file
f.seek(0)
# Read again
f.read()
```

Modules and Packages

Modules in Python are simply Python files with the .py extension, which implement a set of functions. Modules are imported from other modules using the import command. Before you go ahead and import modules, check out the full list of built-in modules in the Python Standard library.

When a module is loaded into a running script for the first time, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

If we want to import module math, we simply import the module:

```
# import the library
import math

# use it (ceiling rounding)
print(math.ceil(2.4)) #Returns the smallest integer greater than or equal to x.
print(math.floor(2.4))          #floor is down #ceil is up so the answer will return 3
```

Exploring built-in modules

While exploring modules in Python, two important functions come in handy - the dir and help functions. dir functions show which functions are implemented in each module. Let us see the below example and understand better.

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

When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter:

In [11]:

```
help(math.ceil)
```

Help on built-in function ceil in module math: # output of the code above

ceil(x, /)
    Return the ceiling of x as an Integral.

    This is the smallest integer >= x.

Writing modules¶

Writing Python modules is very simple. To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

Writing packages

Packages are name-spaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

The twist is, each package in Python is a directory which MUST contain a special file called ***_init_*.py**. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the ***_init_*.py** file inside the foo directory.

To use the module bar, we can import it in two ways:

```
# Just an example, this won't work
import foo.bar
#it's a one way to access the package to access the module you need.
#first we need to create a file named after the module name ,
# the file should have an extension like file.py --file_name.py
#in this case it's foo
#and accessing bar module for the code

# OR could do it this way
from foo import bar
#it's an another way to access the package to access the module
#same as above which helps in accessing packages. foo is the file bar is the modul we need
```

In the first method, we must use the foo prefix whenever we access the module bar. In the second method, we don't, because we import the module to our module's name-space.

The ***_init_*.py** file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the ***_all_*** variable, like so:

__init__.py:

__all__= ["bar"]

#accessing the module from __init__ function

Errors and Exception Handling¶

In this section, we will learn about Errors and Exception Handling in Python. You've might have definitely encountered errors by this point in the course. For example:

print('Hello) <br class="Apple-interchange-newline">rint('Hello)

```
print('Hello)
```

```
  File "<ipython-input-14-db8c9988558c>", line 1
    print('Hello)
                 ^
SyntaxError: EOL while scanning string literal
```

Note how we get a SyntaxError, with the further description that it was an End of Line Error (EOL) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding of these various error types will help you debug your code much faster.

This type of error and description is known as an Exception. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal.

You can check out the full list of built-in exceptions [here](https://docs.python.org/2/library/exceptions.html). Now, let's learn how to handle errors and exceptions in our own code.

try and except¶

The basic terminology and syntax used to handle errors in Python is the **try** and **except** statements. The code which can cause an exception to occur is put in the *try* block and the handling of the exception are the implemented in the *except* block of code. The syntax form is:

try:

You do your operations here...

...

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

...

else:

If there is no exception then execute this block.

Using just except, we can check for any exception: To understand better let's check out a sample code that opens and writes a file:

```
# creates the file from the name given by the write command it writes what is given
try:
    f = open('testfile','w')
    f.write('Test write an this')
except IOError:
    #This will check for an IOError exception and then execute this print statement
    print('Error: Could not find file or read data')
else:
    print('Content written successfully')
    f.close()
```

Now, let's see what happens when we don't have write permission? (opening only with 'r'):

```
# here the ‘r’ represents to read ,, and the next following command is of to write so this puts up for error part off the program
try:
    f = open('testfile','r')
    f.write('Test write an this')
except IOError:
    #This will check for an IOError exception and then execute this print statement
    print('Error: Could not find file or read data')
else:
    print('Content written successfully')
    f.close()
```

Notice, how we only printed a statement! The code still ran and we were able to continue doing actions and running code blocks. This is extremely useful when you have to account for possible input errors in your code. You can be prepared for the error and keep running code, instead of your code just breaking as we saw above.

We could have also just said except: if we weren't sure what exception would occur. For example:

```
try:
    f = open('testfile','w')
    f.write('Test write an this')
except IOError:
    #This will check for an IOError exception and then execute this print statement
    print('Error: Could not find file or read data')
else:
    print('Content written successfully')
    f.close()
```

Now, we don't actually need to memorize the list of exception types! Now what if we keep wanting to run code after the exception occurred? This is where **finally** comes in. ##finally The finally: Block of code will always be run regardless if there was an exception in the try code block. The syntax is:

```
try:
   Code block here
   ...
   Due to any exception, this code may be skipped!
finally:

        This code block would always be executed.
```

For example:

```
try:
    f = open("testfile","w")
    f.write("test write with a statement")
finally:
    print("Always execute finally code blocks")
```

We can use this in conjunction with except. Let's see a new example that will take into account a user putting in the wrong input:

In [21]:

```
def askint():
    try:
        val = int(input("Please enter an integer: "))
    except:
        print("Looks like you did not enter an integer!!!")
    finally:
        print("Finally, I executed!!")
    print(val)
```

askint() # function calling if the input is not integer it'll go to the error{exception part} part if not it'll execute the val in the print part

```
# this program doesn't have a loop so if we push it hard it'll send an error
askint()
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-23-cc291aa76c10> in <module>
----> 1 askint()

<ipython-input-21-b205c2871d30> in askint()
      6     finally:
      7         print("Finally, I executed!!")
----> 8     print(val)

UnboundLocalError: local variable 'val' referenced before assignment
```

Check how we got an error when trying to print val (because it was properly assigned). Let's find the right solution by asking the user and checking to make sure the input type is an integer:

```
def askint():
    try:
        val = int(input("Please enter an integer: "))
    except:
        print("Looks like you did not enter an integer!!!")
        val = int(input("Please enter an integer: "))
    finally:
        print("Finally, I executed!!")
    print(val)

# in this line of codes there is 2 time loop
# if the 1st is wrong u can try again if that is also the wrong out put it'll put up an error
askint()
Please enter an integer: .36
Looks like you did not enter an integer!!!
Please enter an integer: .3
Finally, I executed!!
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-46-3acfe937f37a> in askint()
      2     try:
----> 3         val = int(input("Please enter an integer: "))
      4     except:

ValueError: invalid literal for int() with base 10: '.36'

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
<ipython-input-47-799df34241e9> in <module>
----> 1 askint()
      2

<ipython-input-46-3acfe937f37a> in askint()
```

```
      4     except:
      5         print("Looks like you did not enter an integer!!!")
----> 6         val = int(input("Please enter an integer: "))
      7     finally:
      8         print("Finally, I executed!!")

ValueError: invalid literal for int() with base 10: '.3'
```

Hmmm...that only did one check. How can we continually keep checking? We can use a while loop!

```
def askint():
    while True:
        try:
            val = int(input("please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!!")
            continue
        else:
            print("yep that's an integer!")
            break
        finally:
            print("Finally, I executed!!")
        print(val)
```

in this line of codes we have a while loop till the input is not integer it'll be keep on in the loop

```
askint()
please enter an integer: 45.3
Looks like you did not enter an integer!!
Finally, I executed!!
please enter an integer: 4.5
Looks like you did not enter an integer!!
Finally, I executed!!
please enter an integer: 2.6
Looks like you did not enter an integer!!
Finally, I executed!!
please enter an integer: 2.3
Looks like you did not enter an integer!!
Finally, I executed!!
please enter an integer: .23
Looks like you did not enter an integer!!
Finally, I executed!!
please enter an integer: 24
yep that's an integer!
Finally, I executed!!
```

Database connectivity and operations using Python.

For Example, the following is the example of connecting with MySQL database "my_database1" and creating table grades1 and inserting values inside it.

```
#!/usr/bin/python

import sqlite3
#connecting with the database.

db = sqlite3.connect("my_database1.db")
# Drop table if it already exists using execute() method.
db.execute("drop table if exists grade 1")
# the double quotes are sql queries to execute in python
#db.execute is a function that helps in manipulation of the table

# Create table as per requirement
db.execute("create table grade1(id int, name text, score int)")

#inserting values inside the created table
db.execute("insert into grade1(id, name, score) values(101, 'Jhon',99)")
db.execute("insert into grade1(id, name, score) values(102, 'Gary',90)")
db.execute("insert into grade1(id, name, score) values(103, 'James',80)")
db.execute("insert into grade1(id, name, score) values(104, 'Cathy',85)")
db.execute("insert into grade1(id, name, score) values(105, 'Kris',95)")



db. commit()

# .commit() is used to save the process in the database


results = db.execute("select * from grade1 order by id")
for row in results:
    print(row)
print("-"*60)
(101, 'Jhon', 99)
(102, 'Gary', 90)
(103, 'James', 80)
(104, 'Cathy', 85)
(105, 'Kris', 95)
------------------------------------------------------------
results = db.execute("select * from grade1 where name = 'Gary'")
for row in results:
    print(row)
print("-"* 60)
(102, 'Gary', 90)
------------------------------------------------------------
results = db.execute("select * from grade1 where score >=90")
for row in results:
    print(row)
print("-"* 60)
(101, 'Jhon', 99)
```

```
(102, 'Gary', 90)
(105, 'Kris', 95)
------------------------------------------------------------
results = db.execute("select name, score from grade1 order by score desc ")
for row in results:
    print(row)
print("-"* 60)#descending order
('Jhon', 99)
('Kris', 95)
('Gary', 90)
('Cathy', 85)
('James', 80)
------------------------------------------------------------
results = db.execute("select name,score from grade1 order by score ")
for row in results:
    print(row)
print("-"* 60) # by default ascending order
('James', 80)
('Cathy', 85)
('Gary', 90)
('Kris', 95)
('Jhon', 99)
------------------------------------------------------------
```