Python data types and Simple Operations

```
1
```

Following are python data type

- Number
- String
- Boolean
- List
- Tuple
- Dictionary

```
# Python sets the data type based on variable assigned to it

data = 500  #  The data is a number. data is to be seen as a variable

print(data)

data = 'I am a string stte now' # the data is a string now
print(data)


```

```
# The only characters in variables are letters, numbers and underscores
# variables cannot start with numbers.

this_is_valid_variable = 2

# 123_variable = 3  # invalid variable
# 123-variable = 3   # invalid variable


# python is case sensitive new_variable and New_variable are two different varia

new_variable = 2
New_variable = 3
```

int (plain integers): integers are just positive or negative whole numbers.

long (long integers): long integers are integers of infinite size.

In Python 3 int and long have been merged. We only need to use int data type.

float (floating point real values): floats represent real numbers, but are written with decimal points (or scientific notaion) to divide the whole number into fractional parts.

complex (complex numbers): represented by the formula a + bJ, where a and b are floats, and J is the square root of -1 (the result of which is an imaginary number). Complex numbers are used sparingly in Python.

```
# Python Number Data Type


a = 100  # A int datatype

print (type(a))

a = 2**123  # A long integer
print (a)
print (type(a))

print ('Type of a is {}'.format(type(a)))

a = 97.32 # A float value
print ('Type of a is {}'.format(type(a)))

a =3.14j  # A complex number
print ('Type of a is {}'.format(type(a)))


```

```
1
```

```
# Simple mathematical operations


a = 2
b = 3

c = a+b
print ('The sum is',c)

print ('\nAdding 2+3 = {}'.format(a+b)) # addition of numbers

print ('\n***Adding {}+{} = {}'.format(a,b,a+b)) # addition of numbers

print ('\n===Adding ',a,"+",b, "=", a+b) # addition of numbers


```

```
a = 2.2
b = 1

print ('\nAdding 2.2+1 = {}'.format(a+b)) # addition of float and number. The re

x = a+b
print(type(x))
```

```
a = 2
b = 3

print ('\nMultiplying 2*3= {}'.format(a*b)) # Multiplication of two numbers
```

```

a = 2
b = 5

print ('\nDivision  5/2= {}'.format(b/a)) # Divisions of two numbers
print ('\nQuotient  5//2= {}'.format(b//a)) # Getting quotient from division of
print ('\nRemainder 5 % 2= {}'.format(b%a)) # Getting Remainder from division o


```

```

a = 5
b = 6


print ('\nExponential {}**{}= {}'.format(a,b,a**b)) # Exponential of two numbers



```

```
Exponential 5**6= 15625
```

```
# inplace operators

data = 2
data +=3   # Equivalent to data = data + 3
print (data)
```

```
5
```

Python Strings

```
# String data type

# Strings can be created using single quote ', double quotes '' or triple quotes

first_name = 'Vishal'
print (first_name)
last_name = "Singh"
print (last_name)

print ('\n')

long_message = """ This is a long message which will span multiple lines
by pressing enter.

Line 1  =
Line 2
                """
print (long_message)
```

```
Vishal
Singh


 This is a long message which will span multiple lines
by pressing enter.

Line 1  =
Line 2
```

```
1
```

```
# Strings can be accessed as whole Strings or as a Substring

var1 = 'Python Language'

print (var1)

print (var1[0])


print (var1[0:3]) #prints character from position 0 to 2

print('Last Position is',var1[-2])  # last position is -1

print("****")
print (var1[-3:-1]) #prints character from position -3 to last minus 1

print( var1[-3:])

print( var1[:])


```

```
Python Language
```

```
P
Pyt
Last Position is g
****
ag
age
Python Language
```

```
# As single quote and double quotes are used to mark a string we need to use a
# special way if we want to use them in string. This process is known as escapi

# This is done by placing a backslash character in front of them



str = """Vishal's house is in Mumbai"""
print (str)

str = "This is very \"exciting\"   '
print (str)

```

```
  File "<ipython-input-10-5813a505b591>", line 8
    str = "This is very \"exciting\"   '
                                       ^
SyntaxError: EOL while scanning string literal
```

SEARCH STACK OVERFLOW

```
# String Concatenation

new_string = 'Hello '+'World'
print(new_string)

new_string = 'Hello '+"World"  # Does not matter if string is created with a si
print(new_string)

```

```
Hello World
Hello World
```

```
new_string = '2'+'3'
print(new_string)  # sees string as character.


new_string = '2'+ 3 # one cannot mix data types
print(new_string)

```

```
23
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-26-758cda03929a> in <module>()
      3
      4
----> 5 new_string = '2'+ 3 # one cannot mix data types
      6 print(new_string)

TypeError: must be str, not int
```

SEARCH STACK OVERFLOW

```
#multiplication of strings

str = "data "*3
print (str)

str = "4"*3
print (str)

str = "data"* 3.2
```

```
data data data
444
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-2acfccdc2ff3> in <module>()
      6 print (str)
      7
----> 8 str = "data"* 3.2

TypeError: can't multiply sequence by non-int of type 'float'
```

SEARCH STACK OVERFLOW

```
# Converting string to different data types

str = '2' +'3'
print (str)

data = int('2') +int('3')
print (data)

print (type(data))
```

Taking Inputs

```
# To take an input from user the function input is used


data = input("Enter a number")
```

```
data = input("Enter a number ")
print (data)
print (type(data))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-1ee6efaff297> in <module>()
----> 1 data = input(int("Enter a number"))
      2
      3 #data = input("Enter a number")
      4 print (data)
      5 print (type(data))

ValueError: invalid literal for int() with base 10: 'Enter a number'
```

SEARCH STACK OVERFLOW

```
int_data_one = int(input("Enter a number "))
int_data_two = int(input("Enter a number "))

print ('The type of int_data_one and int_data_two is {} {}'.format(type(int_data_
print ('The sum is {}'.format(int_data_one + int_data_two))

```

```
Enter a number 1
Enter a number 2
The type of int_data_one and int_data_two is <class 'int'>
The sum is 3
```

```
# Assigning variables

data = "I am alive"
print (data)

# y
# print(y)

del data
print (data)
```

```
I am alive
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-9d621335be45> in <module>()
      7
      8 del data
----> 9 print (data)

NameError: name 'data' is not defined
```

SEARCH STACK OVERFLOW

Python Boolean

```
# There are two Python Boolean values True and False

boolean_data = True
print (boolean_data)
print (type(boolean_data))

boolean_data = False
print (boolean_data)
print (type(boolean_data))
```

```
True
<class 'bool'>
False
<class 'bool'>
```

```
# for comparison we use the equal operator ==
print (5 == 7)
```

```
False
```

```
# Another comparison operator is not equal respresented as !=


print ('2!=3 is {}'.format(2!=3))
print ('3!=3 is {}'.format(3!=3))

data = ('One'=='One')
print ('One==One is {}'.format(data))


print ('One==One is {}'.format('One'=='One'))




data =('One'!='Two')
print ('One!=Two is {}'.format(data))


data =  ('One'!='One')
print ('One!=One is {}'.format(data))
```

```
2!=3 is True
```

```
# Comparison for greater or smaller
```

```
2
3 val = 7 > 5
4 print ('7>5 is {}'.format(val))
5
6 print ('7>5 is {}'.format(str()))
7
8
9
10 val = 10 < 10
11 print ('10<10 is {}'.format(val))
12
13
14 val = 10 <= 10
15 print ('10<=10 is {}'.format(val))
16
```

```
7>5 is True
10<10 is False
10<=10 is True
```

```
1 a = "7<5"
2 print (int(a))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-56f2fdca7962> in <module>()
      1 a = "7<5"
----> 2 print (int(a))

ValueError: invalid literal for int() with base 10: '7<5'
```

SEARCH STACK OVERFLOW

Python Control Structures

An if statement is used to run code based on certain condition

for e.g.

if expression:

```
statements
```

Python uses indentation to decide a block of code

```
1 if 2>1:
2   print('2 is greater than 1') # since the if statment is true this block of co
3   print("One more")
4
5
6   print ('Finished') # print is not part of if block as it is indented as same
```

```
7
8
9
```

An else follows an if statement

if expression:

```
  statements
```

else:

```
  statements
```

```
1 # taking input values
2 int_data_one = int(input("Enter first number ")) #int ? ,Bcz here we used numbe
3 int_data_two = int(input("Enter second number "))
4
5 if (int_data_one> int_data_two ):
6   print ('{} is greater than {}'.format(int_data_one,int_data_two))
7 else:
8   print ('{} is not greater than {}'.format(int_data_one, int_data_two))
9
```

The elif statment is for chaining if and else statement

```
1 num = int(input("Enter first number "))
2
3 if num == 900:
4   print ('num is 900 **** big number') #elif ?, here we continue the if process
5 elif num == 100:
6   print ('num is 100 --- small number')
7 else:
8   print ('value of num is unknown')
9
10
```

```
1 num = int(input("Enter first number "))
2
3 if num == 900:
4   print ('num is 900 **** big number')
5
6 else:
7   print ('value of num is unknown')
8
```

```
1
```

```
# Boolean logic is used to make complicated conditions using and,or and not

# A=true,  B=true  then  (A and B) is true
# A=true,  B=false then  (A and B) is false
# A=false, B=true  then  (A and B) is false
# A=false, B=false then  (A and B) is false


cond = (1==1 and 2==2)
print ("1==1 and 2==2 is {}".format(cond))


cond = (1==1 and 2==3)
print ("1==1 and 2==3 is {}".format(cond))


cond = (5==6 and 2==2)
print ("5==6 and 2==2 is {}".format(cond))


cond = (5==6 and 2==3)
print ("5==6 and 2==3 is {}".format(cond))
```

```
# A=true,  B=true  then  (A or B) is true
# A=true,  B=false then  (A or B) is true
# A=false, B=true  then  (A or B) is true
# A=false, B=false  then (A or B) is false


cond = (1==1 or 2==2)
print ("1==1 or 2==2 is {}".format(cond))


cond = (1==1 or 2==3)
print ("1==1 or 2==3 is {}".format(cond))


cond = (5==6 or 2==2)
print ("5==6 or 2==2 is {}".format(cond))


cond = (5==6 or 2==3)
print ("5==6 or 2==3 is {}".format(cond))
```

```
1==1 or 2==2 is True
1==1 or 2==3 is True
```

```
5==6 or 2==2 is True
5==6 or 2==3 is False
```

```
# not means the opposite

cond = (1==1)
notcond = not cond

print ("not 1==1 is {}".format(notcond))
```

```
not 1==1 is False
```

```
a ="This"
```

```
False
```

```
# Operator Precedence
# Python follows the standard mathematical precedence.
# It is always better to use brackets to enforce logical ordering

val = False == False or True  # This happens as == has higher predence than or
print (val)

val = False == (False or True)  # A bracket override the natural precedence
print (val)

```

```
True
False
```

```
# while statement is like if statement. The crucial difference between if and wl
# that while executes multiple times. It is a type of loop

i = 0
while i<2:
  print (i)
  i= i+1

```

```
0
1
```

```
# A infinite loop continues indefnitely

while True:
  print ('running infinitely')

```

```
  File "<ipython-input-40-4ed413079a9f>", line 4
    return
    ^
SyntaxError: 'return' outside function
```

```
SyntaxError: 'return' outside function
```

SEARCH STACK OVERFLOW

```
# A break statement is used to break from a loop
i = 0
while i<10:
  if i==5:
    break
  print (i)
  i= i+1
```

```
Hello
0
Hello
1
Hello
2
Hello
3
Hello
4
```

```
# A break statement is used to break from a loop
i = 0
while True:
  if i==5:
    break
  print (i)
  i= i+1
```

```
# A continue statement stops the flow and goes back again to the starting point
# The difference between continue and break is
# break - breaks out of the block
# continue - goes back to start of the block

i = 0
while i<20:
  if (i % 2==0):
    i=i+1
    print ("Inside continue",i)
    continue
  i=i+2
  print (i)
  i= i+1
```

```
Inside continue 1
3
Inside continue 5
7
```

```
,
```

```
1
```

```
Inside continue 13
```

Lists and Range

```
# A list denotes a container object in python
# A list is used to store indexed list of items
# A list can contain different types of items for e.g. string, intger or other
word_list = ['This', 'is', 'good']  # A list of 3 items

print (word_list[0]) # prints the first  item. Index starts at 0.
print (word_list[1]) # prints the second item
print (word_list[2]) # print the third item


# An empty list is created with empty bracket
empty = []
print (empty)
```

```
# A list can contain items of multiple types

generic_list = [1,"String",[1,2,3], 5]

print ('Value at {} is {} of type {}'.format(0,generic_list[0],type(generic_lis
print ('Value at {} is {} of type {}'.format(1,generic_list[1],type(generic_lis
print ('Value at {} is {} of type {}'.format(2,generic_list[2],type(generic_lis
print ('Value at {} is {} of type {}'.format(3,generic_list[3],type(generic_lis


# nested list access

print ('Accessing {} value from nested list {} = {}'.format(1,generic_list[2] ,g
```

```
# indexing out of bounds causes error

generic_list = [1,"String",[1,2,3], 5]

generic_list[4] # This will cause error
```

```
# Elements at certain index can be reassigned
```

```
nums = [4,8,16,24]

print ("The list is {}".format(nums))

nums[1] = -222

print ("The list after modification is {}".format(nums))
```

```
```

```
# Lists can be added and multipled

nums = [6,7,8]

new_list = nums * 3

print (new_list)
```

```
[6, 7, 8, 6, 7, 8, 6, 7, 8]
```

```
# A in operator is used to check if an element is present in list

words =["The", "age", "of", 5,"reason"]

x = "The" in words

print(x)

x = "Unreason" in words


print(x)

```

```
True
False
```

```
# A not operator can be used to check if item is not present

nums = [1,4,8]

x = 10 not in nums
print (x)
```

```
# List append function

nums = [5,8,10]
nums.append(14)

print (nums)
```

```
# List Plus Function
nums_one = [5,8,10]
nums_two = [15,81,101]

nums_three = nums_one +nums_two

print(nums_three)
```

```
# length of list

nums = [5,8,10]
print (len(nums))
```

```
# inserting at specific index

nums = [5,8,10]
nums.insert(2,100)

print (nums)
```

```
# index method finds the first occurence of list item and returns the index

nums = [5,8,10,12,10]

ind = nums.index(10)

print (ind)


ind = nums.index(100) # A value Error is raised when item is not present in lis

print (ind)
```

```
nums = [5,8,10,12,10]

print ("The maximum is {}".format(max(nums)))

print ("The minimum is {}".format(min(nums)))

#print ("The number of times 10  is present is {}".format(nums.count(10)))

nums.reverse()

print ("The reverse of list is {}".format(nums))

# using rever
revnums = reversed(nums)
```

```

print ("The reverse of list is {}".format(list(revnums)))

# remove 10

nums.remove(10)

print ("The list after removing 10 is {}".format(nums)) # only removes the firs
```

```
# using pop

nums = [10,2,3,4]
x = nums.pop(1)
print (x,nums)

x = nums.pop()
print (x,nums)
```

```
# vowels list
vowels = ['a', 'e', 'i', 'o', 'i', 'u']

# count element 'i'
count = vowels.count('i')

# print count
print('The count of i is:', count)

# count element 'p'
count = vowels.count('p')

# print count
print('The count of p is:', count)
```

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

for i, j in enumerate(words):
    print (i, j)

```

```

```

```
# List Sorting

nums = [5,8,10,12,10]
print (nums.sort())
print (nums)


nums = [115,8,10,112,10]
x = sorted(nums)
print(x)


nums = [5,8,10,12,10]
print (nums.sort(reverse=True))
print (nums)


nums = [115,8,10,112,10]
x = sorted(nums,reverse=True)
print(x)

```

```
# List copying

old_list = [1, 2, 3]
new_list = old_list

# add element to list
new_list.append('a')

print('New List:', new_list )
print('Old List:', old_list )

#The problem with copying the list in this way is that if you modify the new_li


#However, if you need the original list unchanged when the new list is modified
#shallow copy.

# mixed list
lista = ['cat', 0, 6.7]

# copying a list
new_list = lista.copy()

# Adding element to the new list
new_list.append('dog')

```

```
# Printing new and old list
print('Old List: ', lista)
print('New List: ', new_list)
```

```
# removing all values of 10

nums = [5,8,10,12,10]
while 10 in nums:
  nums.remove(10)

print ("The list after removing 10 is {}".format(nums)) # This removes all the
```

```
# Range function creates a list of numbers
num = list(range(10))
print (num)

num = list(range(1,10))
print (num)

num = list(range(1,10,2)) # positive step
print (num)


num = list(range(10,0,-1)) # negative step
print (num)



for i in range(1,10):
  print (i)

```

```
# For Loop

data = [1,4,10]
for d in data:
  print (d)

print("\n")

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

```

```
# Clear function clears all the values of a list
fruits = ['apple', 'banana', 'cherry', 'orange']

fruits.clear()
print(fruits)
```

Python Function and Modules

Python Function and Modules

Functions help to reuse code. The bring modularity and simplicity to code

```
# functions in python are created using def statement
```

```
def name_of_great_philospher():
  print ('Bertrand Arthur William Russell')

name_of_great_philospher()
```

```
# functions need to be defined before they are called
name_of_philospher()

def name_of_philospher():
  print ('Bertrand Arthur William Russell')
```

```

```

```
# One can pass arguments to functions
# name is an argument for the function name_of_philosphers

def name_of_philospher(name, country):
  print (name,country)


name_of_philospher('Bertrand Arthur William Russell', 'United Kingdom')

```

```
#function can  return values

def name_of_philospher(name, country):
  str = "The philosopher {} lived in  country {}".format(name,country)
  return str


name = name_of_philospher('Bertrand Arthur William Russell', 'United Kingdom')
print (name)

```

```
# function can return multiple values

def add_and_substract(num1,num2):
    add = num1+num2
    diff = num1-num2
    return add,diff

add,diff = add_and_substract(10,2)

```

```
print ("The sum is =",add)
print ("The difference is =",diff)
```

```
# Using named arguments

def add_and_substract(num1,num2):
    add = num1+num2
    diff = num1-num2
    return add,diff

add,diff = add_and_substract(num1=10,num2=2)

print ("The sum is =",add)
print ("The difference is =",diff)

add,diff = add_and_substract(num2=2,num1=10)
print ("The sum is =",add)
print ("The difference is =",diff)
```

```
#any code after return will not be executed

def add_and_substract(num1,num2):
    add = num1+num2
    diff = num1-num2
    return add,diff
    print ('After Function')

add,diff = add_and_substract(num1=10,num2=2)
print ("The sum is =",add)
print ("The difference is =",diff)
```

```
#docsstring are documentation strings. They are created by using a multiline st
#after first line of function

def add_and_substract(num1,num2):
    """
     The function adds and subtsracts two numbers
    """
    add = num1+num2
    diff = num1-num2
    return add,diff


add,diff = add_and_substract(num1=10,num2=2)
print ("The sum is =",add)
print ("The difference is =",diff)
```

```
# Functions are just like normal variables. They can be assigned and
# reassigned to variables

def add_and_substract(num1,num2):
    """
     The function adds and subtsracts two numbers
    """
    add = num1+num2
    diff = num1-num2
    return add,diff

operation = add_and_substract

num1, num2 = operation(5,2)
print (num1, num2)
```

```
# function can also be used as arguments to other functions

def add_and_substract(num1,num2):
    """
     The function adds and subtsracts two numbers
    """
    add = num1+num2
    diff = num1-num2
    return add,diff

def add_summ_diff(func, num1, num2):
      x,y = func(num1,num2)
      return x+y


val = add_summ_diff(add_and_substract,10,3)
print (val)
```

```
# The special syntax, *args and **kwargs in function definitions is used to pass

# The single asterisk form (*args) is used to pass a non-keyworded, variable-len
# and the double asterisk form is used to pass a keyworded, variable-length argu


def add_numbers(*args):

  for i in args:
    print (i)

```

```
add_numbers(1,2,3,4,5)



def add_numbers(**kwargs):
    for key in kwargs:
        print ("another keyword arg: {}: {}".format(key, kwargs[key]))


add_numbers(num1=1,num2=2,num3=3,num4=4,num5=5)
```

```
# modules can be thought as collection of functions
# the way to use module is via import function

import random

for i in range(6):
  value = random.randint(1,8)
  print (value)
```

```
# One can also import a specific function

from random import randint

for i in range(6):
  value = randint(1,8)
  print (value)
```

```
# one can import a module under a different name or object

from random import randint as my_rand_func

for i in range(6):
  value = my_rand_func(1,8)
  print (value)
```

There are main three types of modules

- Modules which one write on own
- Third Party Modules
- Python Standard Library

Standard library includes string,re,datetime and many more.

The third party modules can be installed using pip.

```
# for e.g to install google-api-python-client
```

```
!pip install google-api-python-client
```

Python Exceptions

```
# Exceptions occur when something wrong happens in the code. The program stops
num1 = 4
num2 = 0

print (num1/num2)

print ("Exiting")   #This line will not be printed
```

Exceptions can be raised for multiple reasons. For e.g.

- ImportError - An import fails
- IndexError - A list is indexed with out of range error
- SytanxError - Some syntax error
- TypeError - A function is called with a wrong type of variable
- ValueError - A function is called with inappropriate value

```
#import nump1

#list = [1,2,3]
#list[4]        # Index Error


#list = {1,2,3] # Syntax error


#x = int('One') # This causes a value error

def add_num(num1,num2):
    return int(num1)+int(num2)   #value error


add_num("One","Two")
```

```
# To handle exceptions we use try/except statement. Try contains code that migh


try:
    num1 = 8
```

```
  num1 = 8
  num2 = 0
  x = num1/num2           # This line throws exception. The code stops immediate
  print ("This statement will not be printed") # This line is not executed.

except ZeroDivisionError:
   print ("Zero Division Error")

```

```
# A try statement can have multiple except blocks

try:
  variable = 10
  print(variable+"hello")
  print(variable/2)

except ZeroDivisionError:
  print("Divided by Zero")
except(ValueError, TypeError):
  print("Error Occured")

```

```
# An except statement without any exception specified will catch all errors

try:

  4/0

except:
  print ("Error Occured")
```

```
# To ensure that code runs no matter a finally block will be used
try:
  print("Step 1")
  4/0                           # An exception happen here
except ZeroDivisionError:       # The exception is caught here
  print ("Divided by Zero")     # This is printed

finally:
  print("This will be always executed")  # This will be executed whether there
```

```
# Code in finally statement always runs even if there is an exception in above l

try:
  print (8/0)
except ZeroDivisionError:
  print (unknown_data)  # This will trigger an exception
finally:
  print ("This will always be executed")
```

```
# One can raise exceptions by using the raise statement
print ("data")
```

```
raise ValueError
print ("2")
```

```
# Exceptions can be raised by giving details about arguments

name = "123"
raise NameError("Invalid Name")
```

```
# In except blocks the raise statement can be used without arguments. It re rai
# has occured

try:
  num = 5/0
except:
  print ("Error Occured")
  raise
```

```
def my_func():
  try:
    4/0   # This raises ZeroDivisionError

  except Exception as ex:
    template = "An exception of type {0} occurred. Arguments:\n{1!r}"
    message = template.format(type(ex).__name__, ex.args)
    print (message)
    raise # The ZeroDivisionError is propogated out


def another_func():
  try:
    my_func()
  except ZeroDivisionError: # The ZeroDivisionError is caught
      print ('Error')

another_func()
```

```
class MyException(Exception):

    def __init__ (self,code):
        self.code = code



def my_func():
    try:
        4/0   # This raises ZeroDivisionError
    except ZeroDivisionError:
        raise MyException(100)
```

```
14
15
16 try:
17     my_func()
18 except MyException as ex:
19     print (ex)
20
```

File Handling Python

```
 1
 2 # In Python a file can be either text or binary
 3 # Text files are structured as a sequence of lines, where each line includes a s
 4 # Each line is terminated with a special character, called the EOL or End of Lin
 5 # There are several types but the most common is the comma {,} or newline charac
 6
 7 my_file = open("philosphers.txt", "w") # open file for writing
 8
 9 lines = ['The great philosphers of the world\n', 'John Locke\n', 'Kant\n']
10 my_file.writelines(lines)
11 my_file.close()
12
13 my_file = open("philosphers.txt", "r") # open file for reading
14 file_data = my_file.read()
15 print(type(file_data))
16 print (file_data)
17 my_file.close()
18
19
20
21
```

```
 1 my_file = open("philosphers.txt", "w") # open file for writing mode will erase a
 2 my_file.write("John Locke\n")
 3 my_file.close()
 4
 5 my_file = open("philosphers.txt", "r") # open file for reading
 6 file_data = my_file.read()
 7 print (file_data)
 8 my_file.close()
 9
10
11 my_file = open("philosphers.txt", "a") # open file for writing in append mode
12 my_file.write("Immanuel Kant\n")
13 my_file.close()
14
15
16 my_file = open("philosphers.txt", "r") # open file for reading
17 file_data = my_file.read()
18 print (file_data)   # will not print John Locke and Immanuel Kant
19 my_file.close()
20
```

```
# This kind of style always ensures that files are closed

try:
  my_file = open("philosphers.txt", "w") # open file for writing
  my_file.write("The great philosphers of the world")
finally:
  my_file.close()
```

```
# This code will work only in pycharm.


import os

package_dir = os.path.dirname(os.path.abspath(__file__))
print(package_dir)
thefile = os.path.join(package_dir,'philosphers.txt')
print(thefile)

my_file = open(thefile, "r") # open file for reading
file_data = my_file.read()
print(type(file_data))
print (file_data)
my_file.close()

```

```
# Another way of working with files is
with open("philosphers.txt") as f:
  print(f.read())
```

```
# A none object is used to represent the absence of a value.
# it is like null in any other programming language

x = (None == None)
print (x)

x = (None == True)
print (x)

x = (None == False)
print (x)


def afunc():           # A function which does not return a value returns none.
   print("Hello")

print (afunc())        # A none is returned
```

Dictionaries

```
# Dictionaries are data structures used to map key to values

philosphers = {"United Kingdom":"Bertrand Russell", "Germany":"Karl Marx"}

print (philosphers["United Kingdom"])
print (philosphers["Germany"])
```

```
# Accessing a Index which is not part of dictionary results in a keyerror

philosphers = {"United Kingdom":"Bertrand Russell", "Germany":"Karl Marx"}

print (philosphers["India"])  # This will result in a KeyError
```

```
# Only immutable objects can be used as keys. Using a mutable object as key wil


d = {
    [1,2,3]:"One Two Three"
}

print (d)
```

```

philosphers = {"United Kingdom":"Bertrand Russell", "Germany":"Karl Marx", "Ind

philosphers["India"] = "AryaBhatt"              #One can assign a existing key
philosphers["United Kingdom"] = "John Stuart Mill" #One can assign a existing k
philosphers["United States"] = "John Dewey" # One can create a new key and assi


print (philosphers)
```

```

# One can determine whether a key is present in dictionary or not using in and

philosphers = {"United Kingdom":"Bertrand Russell", "Germany":"Karl Marx", "Ind

print ("India" in philosphers)
print ("Brazil" in philosphers)
print ("Brazil" not in philosphers)


print(philosphers)
```

```

```

```
# One can determine whether a key is present in dictionary or not using in and n

philosphers = {"United Kingdom":"Bertrand Russell", "Germany":"Karl Marx", "Indi

print (philosphers.keys())

print(type(philosphers.keys()))

print (philosphers.values())
print(type(philosphers.values()))

print (philosphers.items())
print(type(philosphers.items()))

```

```

```

```
# A get method is like an index with added advantage that one can specify a defa
# if the index is not found This prevents keyerror

philosphers = {"United Kingdom":"Bertrand Russell", "Germany":"Karl Marx", "Indi

print (philosphers.get("Russia","No philospher set for Russia"))  # No keyerror
print (philosphers.get("Russia"))  # By default a None value will be returned
print (philosphers["Russia"])   # A keyerror occurs

```

JSON

```
import json

john_row = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ["Ann","Billy"],
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}

hari_row = {
  "name": "hari",
  "age": 30,
```

```
  "married": True,
  "divorced": False,
  "children": ["Ann","Billy"],
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}

person_table = []
person_table.append(john_row)
person_table.append(hari_row)


print(json.dumps(person_table, indent =4))
```

Tuples

```
# Tuples are similar to list except that they are immutable. Once assigned valu
# They are created using paranthesis



philosphers = ("Bertrand Russell", "Karl Marx", "Arya Bhatt")
print (philosphers[0])

#philosphers[0] = "John Locke" # This will result in an error


def multiple_ret_params():
  return 1,2,3


x = multiple_ret_params()
print(type(x))  # returns a tuple


def func(*arg):
    print("The  argument type is",type(arg))  # This will print tuple
    pass

func(1,2,3)
```

```
# Tuples can also be created without parentheses by just separating with commas

philosphers = "Bertrand Russell", "Karl Marx", "Arya Bhatt"
print (philosphers[2])
```

```
print (philosphers[2])
```

Sets

```
first_set = {"a", "b", "c","a"}
#first_set.add("d")
#first_set.add("a")

print (first_set)
```

```
first_set = {"a", "b", "c"}
second_set = {"d", "e", "f"}

x = first_set.union(second_set)
print  (x)
```

```
first_set = {"a", "b", "c"}
second_set = {"a", "b", "f"}

first_set.union(second_set)
```

```
first_set = {"a", "b", "c"}
second_set = {"a", "b", "f"}

first_set.intersection(second_set)
```

```
first_set = {"a", "b", "c"}
second_set = {"a", "b", "f"}

print(first_set -(second_set))

print(first_set.difference(second_set))
```

```
1
```

List Slices

```
# List slices provides a advanced mechanism for accessing values from list
# using colon notation

```

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

print (squared[2:6]) # prints the value at index 2,3,4,5
print (squared[:6])  # prints the value at index 0,1,2,3,4,5
print (squared[3:])  # prints all the value from 3 till end

print (squared[0:1])  # prints the value at index 0
```

```
# List slices can also have the third argument which represents the step

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

print (squared[::2]) # prints alternate values

# 0 1 2 3 04 05 06 07 08
# 0 1 4 9 16 25 36 49 64

print (squared[2:8:3]) # prints from 2 to 7 with step of 3
```

```
# List slices can also be negative

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

print (squared[1:-1])   # Goes from start to one to -1(last), -1 is not include
print (squared[:-1])

print (squared[:-1:2])


#Reversing a list

print (squared[9::-1])

print (squared[len(squared)::-1])

print (squared[::-1])
```

```
# One can also copy a list using list slicing.


list = ['cat', 0, 6.7]

```

```
# copying a list using slicing
new_list = list[:]

# Adding element to the new list
new_list.append('dog')

# Printing new and old list
print('Old List: ', list)
print('New List: ', new_list)
```

List Comprehension

```
# List comprehensions are useful way for creating lists with a simple rule

# Pythonic way
square = [a* a for a in range(5)]
print (square)


# Without list comprehension
square_list =[]
for a in range(5):
  square_list.append(a*a)
print(square)
```

```
# An if statement can also be used in list comprehension

# Pythonic way
square = [a* a for a in range(5) if a%3 == 0]
print (square)


# Without list comprehension
square_list =[]
for a in range(5):
  if a%3 == 0:
    square_list.append(a*a)

print(square)
```

Dictionary Comprehension

```

# Python code to demonstrate dictionary
# comprehension
```

```
4
5 # Lists to represent keys and values
6 keys = ['a','b','c','d','e']
7 values = [1,2,3,4,5]
8
9 print (list(zip(keys, values)))
10
11 # but this line shows dict comprehension here
12 myDict = { k:v for (k,v) in zip(keys, values)}
13
14 # We can use below too
15 # myDict = dict(zip(keys, values))
16
17 print (myDict)
```

Formatting of Strings

```
1 nums = [1,2,3]
2
3 out = "The numbers are {},{},{}".format(nums[0],nums[1],nums[2])
4 print (out)
5
6
7
8 # named arguments can also be used
9 out = "The numbers are {a},{b},{c}".format(b=nums[1],a=nums[0],c=nums[2])
10 print (out)
11
12
13
```

Python String Functions

```
1
2
3
4 str = ",".join(["Life","is","good"])  # joins all the words in list with comma
5 print (str)
6
7 str = "Life is good".replace("good","strange")  # replace all occurences of gooc
8 print (str)
9
10 str = "Hello World. Hello Python".replace("Hello","Namaste")  # replace all occu
11 print (str)
12
13
14 bool = "Hello World. Hello Python".startswith("Hello")  # Checks if string star
15 print (bool)
16
```

```
bool = "Hello World. Hello Python".endswith("Python")  # Checks if string ends \
print (bool)


str = "The world is flat"
x = "is" in str
print (x)


str = "The world is flat ld"
x = str.find('ld')
print (x)

str = "The world is flat"
x = str.find('is flat')
print (x)
```

```
upper = "hello world".upper()
print (upper)

lower = "HELLO world".lower()
print (lower)

lower = "hello world".capitalize()
print (lower)

list = "Lion, Tiger,Elephant".split(",")
print(list)


list = "Lion * Tiger  * Elephant".split("*")
print(list)

list = "Lion * Tiger  * Elephant".split()
print(list)
```

- Combining String Split and Comprehension"link text

```
list = "Lion, Tiger,Elephant".split(",")
print(list)

myDict = { k:v for (k,v) in enumerate(list)}
print(myDict)
```

Python Numeric Functions

```
list = [1,4,6,7]

print ("The minimum is",min(list))
print ("The maximum is",max(list))
print ("The sum is",sum(list))


print (abs(-10))

```

Double-click (or enter) to edit

Python List Functions

```
#Return Value from any()
#The any method returns:

# True if at least one element of an iterable is true
# False if all elements are false or if an iterable is empty


l = [22,33,45,67,54]

print(any(l))  # Print True

l = [0]
print(any(l))  # Print False

l = [-1]
print(any(l))  # Print True


l = [0, False]
print(any(l))  # Print False

l = [0, False, 5] #Print True
print(any(l))


l = []          #Print False
print(any(l))



```

```
# To find if any one of the number is greater than 30

l = [22,33,45,67,54]

print([ i > 30 for i in l])
```

```
print([ i > 30 for i in l])

any([ i > 30 for i in l])
```

```
# Any with strings

s = "This is good"
print(any(s))

# 0 is False
# '0' is True
s = '000'
print(any(s))

s = ''
print(any(s))   # only empty string is false
```

```
# Any with dictonaries

d = {0: 'False'}
print(any(d))   #False

d = {0: 'False', 1: 'True'}
print(any(d))   #True

d = {0: 'False', False: 0}
print(any(d))  #False

d = {}
print(any(d))  #False

# 0 is False
# '0' is True
d = {'0': 'False'}
print(any(d))  #true
```

```
#Return Value from all()
#The all() method returns:

#True - If all elements in an iterable are true
#False - If any element in an iterable is false


# all values true
l = [1, 3, 4, 5]
print(all(l))    #Prints True

# all values false
l = [0, False]
print(all(l))    #Prints False

# one false value
l = [1, 3, 4, 0]
print(all(l))      #Prints False
```

```
print(all(l))      #Prints False

# one true value
l = [0, False, 5]
print(all(l))       #Prints False

# empty iterable
l = []
print(all(l))     #Prints True
```

```

# All with string

s = "This is good"
print(all(s))

# 0 is False
# '0' is True
s = '000'
print(all(s))

s = ''
# print(all(s))
```

```
# All with dictionaries

s = {0: 'False', 1: 'False'}
print(all(s))

s = {1: 'True', 2: 'True'}
print(all(s))

s = {1: 'True', False: 0}
print(all(s))

s = {}
print(all(s))

# 0 is False
# '0' is True
s = {'0': 'True'}
print(all(s))
```

```
# Enumerate function

nums = [4,8,12,16]

# t is a tuple of value and index

for t in enumerate(nums):
    print(t)
print ('-----------------')

```

Object Oriented Programming

```
# Object oriented programming considers objects storing data  and methods
# A program is considered as interaction of objects

# A class is objects blueprint - description or defnition
# Classes are created using keyword class and indented block
# which contains class methods
```

```
class Philospher:

  # The class has two attributes name and country


  # init method is called when the object is getting created
  # It has self argument passed as the first parameter
  # self  represents the instance

  def __init__(self, name,country):   # This is the contructor
    self.name = name
    self.country = country


russell = Philospher('Bertrand Rusell', 'United Kingdom')
print(russell.name,russell.country)

panini =  Philospher('Panini', 'India')
print(panini.name,panini.country)
```

```
# Classes can have methods

class Philospher:

  def __init__(self, name,country):   # This is the contructor
    self.name = name
    self.country = country

  def show_details(self):
    print('name = {},country = {}'.format(self.name,self.country))

russell = Philospher('Bertrand Russell', 'United Kingdom')
russell.show_details() # Note that one need not pass self
```

```

```

```
# There can also be class level attributes

class Philospher:

  count = 0   # class level attribute. It is attached to all instances

  def __init__(self, name,country):   # This is the contructor
    self.name = name
    self.country = country
    Philospher.count = Philospher.count+1  # The class level attribute is increa

  def show_details(self):
    print('name = {},country = {}'.format(self.name,self.country))


russell = Philospher('Bertrand Russell', 'United Kingdom')
panini =  Philospher('Panini', 'India')

print ('The Philospher count is',Philospher.count)

print ('The Philospher count is',russell.count) # A instance refrence to count p
print ('The Philospher count is',panini.count) # A instance refrence to count p

```

```
# Trying to access an attribute which does not exist causes an attribute error


class Philospher:

  count = 0

  def __init__(self, name,country):   # This is the contructor
    self.name = name
    self.country = country
    Philospher.count = Philospher.count+1

  def show_details(self):
    print('name = {},country = {}'.format(self.name,self.country))


russell = Philospher('Bertrand Russell', 'United Kingdom')
russell.town # Town does not exist

```

```
class JwelleryShop:

      def __init__(self, name,location):
        self.name = name
        self.location = location

```

```
x =  JwelleryShop('Bhima', ' Jayanagar')
print(x.name, x.loaction)

y =  JwelleryShop('AVR', ' J P Nagar')
print(y.name, y.loaction)
```

Inheritance

```
#Inheritance allows class to share functionality between classes


class Philospher:

  count = 0

  def __init__(self, name,country):   # This is the contructor
    self.name = name
    self.country = country
    Philospher.count = Philospher.count+1

  def show_details(self):
    print('name = {},country = {}'.format(self.name,self.country))


class GreekPhilospher(Philospher):

  def  __init__(self, name,country,details):
      Philospher.__init__(self, name,country)  # The base class constructor is c
      self.details = details

  def show_details(self):
    print('name = {},country = {}, philosophy={} '.format(self.name,self.country


plato = GreekPhilospher('Plato','Greece','After Death Philosophy')
plato.show_details()
```

```
# The function super can be used to call the parent class function

class Philospher:

  count = 0

  def __init__(self, name,country):   # This is the contructor
    self.name = name
    self.country = country
    Philospher.count = Philospher.count+1
```

```
11
12   def show_details(self):
13     print('name = {},country = {}'.format(self.name,self.country))
14
15
16 class GreekPhilospher(Philospher):
17
18   def  __init__(self, name,country,details):
19       Philospher.__init__(self, name,country)  # The base class constructor is c
20       self.details = details
21
22   def show_details(self):
23     super().show_details()   # Super invokes the parent Philospher show detail f
24     print('The special philosophy is {} '.format(self.details))
25
26
27 plato = GreekPhilospher('Plato','Greece','After Death Philosophy')
28 plato.show_details()
29
```

Magic Methods

```
1 # Magic methods are special methods which have double underscores at begining an
2 # end of their names. They are known as dunders
3
4 class MegaNumber:
5
6   def __init__(self, x,y):  # constructor also treated as magic operation
7     self.x = x
8     self.y = y
9
10   def __add__(self, othermeganumber):
11       return MegaNumber(self.x+ othermeganumber.x, self.y+othermeganumber.y)
12
13   def myaddition (self, othermeganumber):
14       return MegaNumber(self.x+ othermeganumber.x, self.y+othermeganumber.y)
15
16
17 first  =  MegaNumber(10,12)
18
19 print(first.x)
20
21 second =  MegaNumber(20,40)
22 print(second.x)
23
24 print('**',first.x+second.x, first.y+second.y)
25
26 third = first.myaddition(second)
27
28 third = first + second  # the + operation gets executed by __add__
29
30 print(third.x, third.y)
```

```
31
32
```

```
1
```

There are many more magic methods

sub for -

mul for mul

truediv for /

Please look at documentation for all the magic methods.

There are also magic methods for making classes act like containers

len for len() **getitem** for indexing **setitem** for assigning to indexed values

iter for iteration over objects *_contains* for in

Object Lifecycle

```
# An object follows the lifecycle of
# creation, usage and destruction

# The first stage is definition of class
# The next stage is instantiation where _init_ is called
# Before  _init_ a __new__ is called


class Philospher:  # => Defnition of class


  def __init__(self, name,country):   # This is the contructor
    self.name = name
    self.country = country
    print ("Inside  Constructor")


  def __new__(cls, *args, **kwargs):
    print ("Executing __new__")
    instance = super(Philospher, cls).__new__(cls)
    return instance

russell = Philospher('Bertrand Russell', 'United Kingdom')   # This triggers ca
print(russell.name,russell.country)
```

```
# When an object is destroyed the memory allocated is freed up.

# Destruction of an object only occurs when its refrence counts reaches zero
```

```
# Destruction of an object only occurs when its refrence counts reaches zero
# Reference count means number of variable and objects refrencing to the object
# del statement reduces the count of an object

# when an object refrences count is 0, python garbage collects it.


b = 90  # Create Object
print (b)

c = b   # Increase ref, ref = 1
print (c)

d = [b] # Increase ref, ref = 2
print (d)

del b # decreases ref but still two objects are refering it

print (c)
print (d)

c = 80   #remove refrence to b
d[0] = -1 #remove refrence to b. At the stage the memory allocated for b can be
```

Data Hiding

The key principle of data hiding is that implementation details of the classes should not be exposed out. One want to use the object and not get bogged with internal details of the object. This is generally implemented using the concepts of public, private and protected.

Python does not implement strict private methods as we will see. It does not have protected methods.

```
# Weak private method and attributes have single underscore at begining
# This is a signal that it should not be used by external code.
# It is only a matter of convention.


class Philospher:  # => Defnition of class

  def __init__(self, name,country):   # This is the contructor
    self._name = name              # Weak Attribute
    self._country = country        # Weak Attribute

  def __repr__ (self):  # magic method for representing object as string
    str = "Name ={}, Country ={}".format(self._name,self._country)
    return str

```

```
russell = Philospher('Bertrand Rusell', 'United Kingdom')

print (russell)

print(russell._name,russell._country)  # We can still access the _name and _cou
```

```
# Strong private methods and attributes have double underscore at begining of t
# This causes their names to be mangled and hence cannot be accessed fromm outs

class Philospher:  # => Defnition of class

  def __init__(self, name,country):   # This is the contructor
    self.__name = name
    self.__country = country

  def __repr__ (self):  # magic method for representing object as string
    str = "Name ={}, Country ={}".format(self.__name,self.__country)
    return str

russell = Philospher('Bertrand Rusell', 'United Kingdom')


print(russell._Philospher__name) # By using mangled structure one can access th
print(russell.__name)  # This cannot be accessed from outside
```

Class and Static Methods

```
# Class methods have class parameter passed to it.
# Instead of accepting a self parameter, class methods take a cls parameter tha

class Philospher:

  count = 0   # class level attribute. It is attached to all instances

  def __init__(self, name,country):   # This is the contructor
    self.name = name
    self.country = country
    Philospher.count = Philospher.count+1  # The class level attribute is incre

  @classmethod
  def show_count(cls):
    print('The number of philosphers are {}'.format(cls.count))



  def print_in_fancy_style(philospher): #static method
```

```
    # static method do not have self passed.
    print('****** The philospher name is {} ***'.format(philospher.name))

  def show_count_static(): #static methods don't have access to class methods
    print('The number of philosphers are {}'.format(count))




russell = Philospher('Bertrand Russell', 'United Kingdom')
panini =  Philospher('Panini', 'India')

Philospher.show_count()
Philospher.print_in_fancy_style(russell)

Philospher.show_count_static()  # This produces error
```

Properties

```
# Python has concept called property
# which helps to convert class_methods to read only attributes
# Reimplement setters and getters into an attribute


class Philospher:

  # The class has two attributes name and country


  # init method is called when the object is getting created
  # It has self argument passed as the first parameter
  # self  represents the instance

  def __init__(self, name,country):   # This is the contructor
    self.name = name
    self.country = country


  @property
  def name_and_country(self):
    """
      Return the  name and country
    """
    return "%s %s" % (self.name, self.country)


russell = Philospher('Bertrand Russell', 'United Kingdom')
```

```
print (russell.name_and_country)

russell.name_and_country = 'Bertrand Russell United Kingdom' # This will not wo

#print (russell.name_and_country())  # will give error


```

```
# Reimplement setters and getters into an attribute


from decimal import Decimal

#############################################################################
class Fees(object):
    """"""

    #----------------------------------------------------------------------
    def __init__(self):
        """Constructor"""
        self._fee = None

    #----------------------------------------------------------------------
    def get_fee(self):
        """
        Return the current fee
        """
        return self._fee

    #----------------------------------------------------------------------
    def set_fee(self, value):
        """
        Set the fee
        """
        if isinstance(value, str):
            self._fee = Decimal(value)
        elif isinstance(value, int):
            self._fee = value

    fee = property(get_fee, set_fee)

f = Fees()
f.fee="1"
print (f.fee)

f.fee = 2
print(f.get_fee())


```

```
from decimal import Decimal
import types

#############################################################################
class Fees(object):
```

```
    """""""

    #--------------------------------------------------------------------
    def __init__(self):
        """Constructor"""
        self._fee = None

    #--------------------------------------------------------------------
    @property
    def fee(self):
        """
        The fee property - the getter
        """
        return self._fee

    #--------------------------------------------------------------------
    @fee.setter
    def fee(self, value):
        """
        The setter of the fee property
        """
        if isinstance(value, str):
            print ('For String')
            self._fee = Decimal(value)
        elif isinstance(value, int):
            print ('For Decimal')
            self._fee = value
f = Fees()

f.fee = "2"
print(f.fee)


f.fee = 5
print(f.fee)


```

Regular Expressions

a, X, 9, < -- ordinary characters just match themselves exactly. The meta-characters which do not match themselves because they have special meanings are: . ^ $ * + ? { [ ] \ | ( )

. (a period) -- matches any single character except newline '\n'

\w -- (lowercase w) matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_]. Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word. \W (upper case W) matches any non-word character.

\b -- boundary between word and non-word

\s -- (lowercase s) matches a single whitespace character -- space, newline, return, tab, form [

\n\r\t\f]. \S (upper case S) matches any non-whitespace character.

\t, \n, \r -- tab, newline, return

\d -- decimal digit [0-9] (some older regex utilities do not support but \d, but they all support \w and \s)

^ = start, $ = end -- match the start or end of the string

[ ] ↳ *59 cells hidden*

Executing Shell Commands Python

```
import os

cmd = "Dir "

returned_value = os.system(cmd)  # returns the exit code in unix

print (type(returned_value))
print('returned value:', returned_value)

```

```
import subprocess

cmd = "Dir"

returned_value = subprocess.call(cmd, shell=True)  # returns the exit code in u
print (type(returned_value))
print('returned value:', returned_value)
```

```
1
```

```
import subprocess


cmd = "Dir *.*"

# returns output as byte string
returned_output = subprocess.check_output(["Dir","*.*"],shell=True)

# using decode() function to convert byte string to string
print(type(returned_output))
print(type(returned_output.decode("utf-8")))
print('Returned Output is:', returned_output.decode("utf-8"))
```

```
1
```

```
s="123"
print(s[0])
```

```

s[0] ='x'


```

```
import sys

def main():
    # print command line arguments
    print (type(sys.argv))
    for arg in sys.argv[1:]:
        print (arg, type(arg))

if __name__ == "__main__":
    main()
```

```
import sqlite3

conn = sqlite3.connect(':memory:')
print ("Opened database successfully")



conn.execute('''CREATE TABLE COMPANY
         (ID INT PRIMARY KEY     NOT NULL,
         NAME           TEXT    NOT NULL,
         AGE            INT     NOT NULL,
         ADDRESS        CHAR(50),
         SALARY         REAL);''')

print ("Table created successfully");


conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print ("Records created successfully");


cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print("ID = ", row[0])
   print("NAME = ", row[1])
   print ("ADDRESS = ", row[2])
```

```
    print ("SALARY = ", row[3], "\n")

print ("Operation done successfully");
conn.close()
```

```
# This will not work. We will have to use outside in conda  environment

# will need to install ipywidgets
# !pip install ipywidgets

import ipywidgets as widgets
from ipywidgets import HBox, VBox
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display
%matplotlib inline

@widgets.interact_manual(
    color=['blue', 'red', 'green'], lw=(1., 10.))
def plot(freq=1., color='blue', lw=2, grid=True):
    t = np.linspace(-1., +1., 1000)
    fig, ax = plt.subplots(1, 1, figsize=(8, 6))
    ax.plot(t, np.sin(2 * np.pi * freq * t),
            lw=lw, color=color)
    ax.grid(grid)



```

```
# This will not work. We will have to use outside in conda  environment

# will need to install ipywidgets
# !pip install ipywidgets


freq_slider = widgets.FloatSlider(
    value=2.,
    min=1.,
    max=10.0,
    step=0.1,
    description='Frequency:',
    readout_format='.1f',
)
#freq_slider


range_slider = widgets.FloatRangeSlider(
    value=[-1., +1.],
    min=-5., max=+5., step=0.1,
    description='xlim:',
    readout_format='.1f',
)
#range_slider

```

```
26
27 grid_button = widgets.ToggleButton(
28     value=False,
29     description='Grid',
30     icon='check'
31 )
32 #grid_button
33
34 color_buttons = widgets.ToggleButtons(
35     options=['blue', 'red', 'green'],
36     description='Color:',
37 )
38 #color_buttons
39
40 title_textbox = widgets.Text(
41     value='Hello World',
42     description='Title:',
43 )
44 #title_textbox
45
46
47 color_picker = widgets.ColorPicker(
48     concise=True,
49     description='Background color:',
50     value='#efefef',
51 )
52 #color_picker
53
54 button = widgets.Button(
55     description='Plot',
56 )
57
58
59
60 def on_button_clicked(b):
61     print("Button clicked.")
62     plot2(b)
63
64 button.on_click(on_button_clicked)
65
66 #button.on_click(plot2())
67
68 #button
69 #@button.on_click
70 #def plot_on_click(b):
71 #    plot2()
72
73
74 def plot2(b=None):
75     print ('Hello')
76     xlim = range_slider.value
77     freq = freq_slider.value
78     grid = grid_button.value
79     color = color_buttons.value
80     title = title_textbox.value
```

```
    bgcolor = color_picker.value

    t = np.linspace(xlim[0], xlim[1], 1000)
    f, ax = plt.subplots(1, 1, figsize=(8, 6))
    ax.plot(t, np.sin(2 * np.pi * freq * t),
            color=color)
    ax.grid(grid)



tab1 = VBox(children=[freq_slider,
                      range_slider,
                      ])
tab2 = VBox(children=[color_buttons,
                      HBox(children=[title_textbox,
                                     color_picker,
                                     grid_button]),
                                     ])
tab = widgets.Tab(children=[tab1, tab2])
tab.set_title(0, 'plot')
tab.set_title(1, 'styling')
VBox(children=[tab, button])
```

```
#https://github.com/timeline.json


import requests

r = requests.get('https://github.com/timeline.json')
print (r.json)
```

```
string = "000001111110000111111100011111111111000000000000001111"
l1 = []
newstr = ""
for i in range(0, len(string) - 1):
    if string[i] == string[i + 1]:
        newstr += string[i]
    else:
        if newstr != "" and len(newstr) > 1:
            newstr += string[i - 1]
            l1.append(newstr)
        newstr = ""

for i in l1:
    print(i, len(i))
```

```
1
```

```
1
```

Advanced Sorting

Advanced Sorting

https://www.afternerd.com/blog/python-sort-list/#sort-tuples

```
l = [17, 5, 12, 1, 14]
l.sort()   # The existing list is sorted in place. No new list is created

print(l)

l = [17, 5, 12, 1, 14]
l.sort(reverse=True)  #sorting in reverse
print(l)

```

```
[1, 5, 12, 14, 17]
[17, 14, 12, 5, 1]
```

```
l = [17, 5, 12, 1, 14]
new_list = sorted(l)  # a new list is created
print(new_list)


l = [17, 5, 12, 1, 14]
new_list = sorted(l, reverse = True)  # a new sorted list is created in reverse
print(new_list)

```

```
[1, 5, 12, 14, 17]
[17, 14, 12, 5, 1]
```

```
l = ["oranges", "apples", "Bananas"]
l.sort()

print(l)  # Banans come first as Python treats upper case to be lower.



l = ["oranges", "apples", "Bananas"]
l.sort(key=str.lower)

print(l)


```

```
['Bananas', 'apples', 'oranges']
['apples', 'Bananas', 'oranges']
```

```
def custom_sort(t):
    return t[1]

L = [("Alice", 25), ("Bob", 20), ("Alex", 5)]
L.sort(key=custom_sort)
print(L)
```

```
L = [("Alice", 25), ("Bob", 20), ("Alex", 5)]
L.sort(key=lambda x: x[1])   # Using Lamdas
print(L)
```

```
1
```

```
# Sorting user define objects

class User:
    def __init__(self, name, age):
        self.name = name
        self.age = age


Bob = User('Bob', 20)
Alice = User('Alice', 30)
Leo = User('Leo', 15)
L = [Bob, Alice, Leo]



L.sort(key=lambda x: x.name)
print([item.name for item in L])


L.sort(key=lambda x: x.age)
print([item.name for item in L])
```

```
['Alice', 'Bob', 'Leo']
['Leo', 'Bob', 'Alice']
```

Functional Programming

https://kite.com/blog/python/functional-programming/

```
addition = lambda a, b: a + b
print(addition(3, 4))
```

```
7
```

```
authors = ['Octavia Butler', 'Isaac Asimov', 'Neal Stephenson', 'Margaret Atwood
sorted(authors, key=len)  # Returns list ordered by length of author name
sorted(authors, key=lambda name: name.split()[-1])  # Returns list ordered alpha
```

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

# Multiply every item by two
print(list(map(lambda x: x * 2, val))) # [2, 4, 6, 8, 10, 12]
```

```
[2, 4, 6, 8, 10, 12]
```

```
list(map(int, ["1", "2", "3"])) # converts every string to an int
```

```
def func1():
    print("From one")

def func2():
    print("From two")

def func3():
    print("From three")

executing = lambda f: f()
list(map(executing, [func1, func2, func3]))
```

```
From one
From two
From three
[None, None, None]
```

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

# Multiply every item by two
reduce(lambda: x, y: x * y, val, 1) # 1 * 1 * 2 * 3 * 4 * 5 * 6

```

```
numbers = [13, 4, 18, 35]
div_by_5 = filter(lambda num: num % 5 == 0, numbers)

# We can convert the iterator into a list
print(list(div_by_5))
```

```
 # Filtering age > 50

 data = [
         {"name": "vishal", "age":50},
         {"name": "Rashmi", "age":30}
       ]

age_greater_than_40 = filter( lambda item: item["age"]>40, data)

print(list(age_greater_than_40))

```

```
[{'name': 'vishal', 'age': 50}]
```

```
import itertools
import operator

data = [1, 2, 3, 4, 5]
```

```
result = itertools.accumulate(data, operator.mul)
for each in result:
    print(each)
```

```
1
2
6
24
120
```

```
import itertools
import operator

data = [1, 2, 3, 4, 5]
result = itertools.accumulate(data, max)
for each in result:
    print(each)
```

```
1
2
3
4
5
```