**Python Notes :**

Python downloaded from python.org is written is in C also known as Cpython where as there are other instances as well know as Jython or IronPython(Ipython)

Augmented Assignment : Guess = guess + 1 can be written as guess += 1 thus doesn’t create a variable and assign but modifies the original variable in place itself

Data types numeric(int, float, complex) sequence(str, list, tuple), mapping(dict), set(set, frozenset), Boolean, binary(byte, bytearray), None(none),special(not implemented, ellipsis) collection

Else can also be used with a for loop : only if for loop terminates without a “break”

Immutable : strings whose value changes by creating a new variable as well as the id changes

Mutable : list dict set bytearray : whose value can be changed (actually gets updated) where as the id remains constant. Changing / updating the content without creating a new variable

Method : ss\_list.append(‘val’) methods always require a object (ss\_list), append is the method and argument are in parenthesis, it uses the value that needs to be appended to the ss list object

.sort() (is a method) will modify the list it is called on without new variable creation

Sorted() will sort the list but will place the content in a new variable, sorted (set(data)) will convert a set to a list

Tuple : aad = (“mayank”,”mbrdi”,56221) is a tuple and is immutable and has a fixed size …its value cant be changed using aad[1] = “hhhh” it consumes less memory compared to list can be

Unpacking : it can be done to tuple and lists but if list size increase it will crash so unpacking goes better with tuple

aad\_list = [(“mayank”,”mbrdi”,56321), (“maya”,”mbrdi”,56322)]  
For index, (name,comp,id) in enumerate aad\_list we will be directly able to access name company and id respectively

Pass by reference : when pass a mutable object in a function

Pass by value : when passing a immutable object in a function

Dictionary is a key value pair and is known as Hashmap in java

To access an item in dictionary we dict\_name[‘Key\_name’] but will give an error if key doesn’t exist so use dict\_name.get(‘Key\_name’) or u can use dict\_name.setdefault(‘Key\_name’, value) to set a default item if not present.

While iterating over a dictionary using for loop prefer dict\_name.items()

dict\_name.pop() pops the last element added, dict\_name.pop(‘key name’, ‘not found’) prints not found if the key isnt there instead of giving error or u can use popitem() for LIFO

list(dict\_name) will give a list of keys in a dictionary

clear() will crear the entire dictionary

dict\_name.update(dict\_name\_2) will update the dict\_name dictionary with new values and new keys if added

in : if in is used with a list it checks the values in the list where as if in is used with dict it checks with the keys of dict and not values of the dict

Shallow copy deep copy   
import copy

original\_list = [1, [2, 3], 4]

shallow\_copy = copy.copy(original\_list)

print(original\_list)  # [1, [2, 3], 4]

print(shallow\_copy)   # [1, [2, 3], 4]

shallow\_copy[1][0] = 99

shallow\_copy[1].append(77)

shallow\_copy.append(33)

print(original\_list)  # [1, [99, 3, 77], 4]

print(shallow\_copy)   # [1, [99, 3, 77], 4, 33]

original\_list\_2 = [1, [2, 3], 4]

deep\_copy = copy.deepcopy(original\_list\_2)

print(original\_list\_2)  # [1, [2, 3], 4]

print(deep\_copy)      # [1, [2, 3], 4]

deep\_copy[1][0] = 99

deep\_copy[1].append(77)

deep\_copy.append(33)

print(original\_list\_2)  # [1, [2, 3], 4]

print(deep\_copy)      # [1, [99, 3, 77], 4, 33]

A **shallow** copy copies the references to object. That means the copy will refer to the same objects as the original. If you copy an object that contains a list, for example, the copy and the original both refer to the same list. \_\_\_\_\_ A **deep** copy will create a copy of all contained objects.

import hashlib   
hashlib helps u to encode a given set of text like password in a particular format like SHA256, SHA156 ….  
u can store the hash and compare it for password authentication   
hash\_text = hashlib.sha256(text.encode(‘utf8’))  
print(f”SHA256: {hash\_text.hexdigest()}”)

sets pets\_set = {‘dog’,’rabbit’,’mouse’,’elephant’,’monkey’} these are unordered and can different ordering each time, fetching using index isnt possible and will give error

set can be use to check if an item is there or not. Ans is much faster if u have to check the same item in a list

list(dict) will give a list of keys

dict.fromkeys(data) will create a dict from the given data and remove similar items and put default value as none

interger\_set.remove(10) 10 must exist to be removed else will throw error

interger\_set.discard(10) 10 may or may not be present to be discarded …wont throw error

wild\_animal.union(farm\_animals) will give both the animals combined similarly we have .intersection and .difference

.symmetric difference, set\_1 ^ set\_2 gives the non common elements of 2 sets

Animals.issuperset(birds) +> true birds.issubset(animals) +> true (answers in Boolean)

def func(p1, p2, \*args, k, \*\*kwargs):

    print("positional {},{}".format(p1,p2))

    print("args used to pack items and send {}".format(args))

    print("k == {}".format(k))

    print("pack key value pairs and send {}".format(kwargs))

func(1,2,3,4,5,6,k=33,name='mayank', branch='cs')

getter and setter is used retrieve and set items without calling these methods these methods start with def \_get\_lives(self) and def \_set\_lives(self, lives) lives = property(\_get\_lives, \_set\_lives)

Super()

class ParentClass:

    def \_\_init\_\_(self, name):

        self.name = name

    def say\_hello(self):

        print(f"Hello from {self.name}")

class ChildClass(ParentClass):

    def \_\_init\_\_(self, name, age):  # overrides the parent class constructer

        super().\_\_init\_\_(name)  # Call the parent class constructor to set name

        self.age = age

    def say\_hello(self):     # overrides the parent class constructer

        super().say\_hello()  # Call the parent class method using super and overrides

        print(f"I'm {self.name} and I'm {self.age} years old")

child = ChildClass("Alice", 30) # Create an instance of the ChildClass

child.say\_hello()               # Call the methods of the child class an 1st prints parent then child func

composition: [better than inheritance ] Composition in Python, in the context of object-oriented programming, Composition is an alternative to inheritance and allows for greater flexibility and reusability in your code. In composition, you build a class by incorporating one or more instances of other classes, rather than inheriting their behaviours. This allows you to create objects that consist of various components, each with its own specific functionality. Composition promotes a "has-a" relationship, where an object contains or is composed of other objects.

class Engine:

    def start(self):

        print("Engine started")

class Wheels:

    def roll(self):

        print("Wheels rolling")

class Car:

    def \_\_init\_\_(self):

        self.engine = Engine()

        self.wheels = Wheels()

    def drive(self):

        self.engine.start()

        self.wheels.roll()

my\_car = Car()

my\_car.drive()

Aggregation : Aggregation in Python, in the context of object-oriented programming, refers to a relationship between two classes where one class contains an instance of another class as a member, but the contained class is not part of the whole identity of the containing class. Aggregation is a form of association, and it represents a "has-a" relationship between objects.In aggregation, the contained class (the "part") can exist independently and may be used in various contexts. It is not exclusively owned by the containing class (the "whole"). This relationship is often referred to as a "weak" relationship because the contained object can be shared among multiple containing objects.

class Department:

    def \_\_init\_\_(self, name):

        self.name = name

class Employee:

    def \_\_init\_\_(self, name, department):

        self.name = name

        self.department = department  # Aggregation: Employee has a Department

    def display\_info(self):

        print(f"Name: {self.name}")

        print(f"Department: {self.department.name}")

hr\_department = Department("HR")# Creating instances of the Department class

it\_department = Department("IT")

employee1 = Employee("Alice", hr\_department)# Creating instances of the Employee class with aggregation

employee2 = Employee("Bob", it\_department)

employee1.display\_info()# Displaying employee information

employee2.display\_info()

Generators and yield : it is used as an iterator where the range isnt fixed ie. It generates the range on during the execution of the program and is better than using range or for loop coz it wont preallocate the memory. Next is used instead of for loop for deliberate execution

def even\_numbers():

    n = 0

    while True:

        yield n

        n += 2

my\_generator = even\_numbers()

for item in my\_generator: # or use next(my\_generator)

    print(item) # prints even number indefinitely

list comprehensions : concise way of creating lists designed to create new lists without causing side effects to the original data, but in case u append or perform any mutable operation it will ruin the list, eg number = [1,2,3,4,5,6] and u acnt add an else to it in if condition  
squares = [x\*\*2 for x in numbers] even\_numbers = [x for x in numbers if x % 2 == 0]  
new\_list = [expression for item in iterable if condition]

Import timeit to know the time taken by the function t  
print (timeit.timeit(function\_name, number=10000))

Filter function list (filter(function\_name, meals)) filters the meal based on requirement

All and any aa = [1,2,3,4,5,6] all(aa) , any(aa) will return true if all values are true in list and any will return true if any one item is true   
all and any can be used like list comprehensions eg any(plant.type == ‘grass’ for plant in plant\_list)

Pickle : with open as pickle\_file pickle.dump(text, pickle\_file, Protocol = xxxx) where xxxx = Highest protocol, 0, Default protocol for converting things to binary

Shelve is use for persistent data storage. So that data can be even stored and retrieved if the program ends. Follows the same dictionary structure.

The \_\_pycache\_\_ directory is automatically generated by Python in each folder where Python source files (modules) are located. It is a cache directory used to store compiled bytecode files (.pyc files) for improved performance when importing modules. Faster imports, directory allows Python to avoid recompiling the same source files repeatedly when they are imported  
by setting environment variables like PYTHONDONTWRITEBYTECODE, which can be used to prevent Python from creating \_\_pycache\_\_ directory

Name.removeprefix(‘Mr. ’) o: Mayank Prasad   
Name.removesuffix(‘Prasad ’) o: Mr. Mayank

Save in json only if u want o retrieve a data but not the format eg it cant save the content in the form of tuple it converts it to List brackets

When using with open, and it reaches the end of the line u can use data.seek(0) to make it reach the start of line

For row in albums :  
album\_dict = dict(zip(column\_name\_list, album\_list\_with\_multiple\_tuple\_items )) its zips to make a list of columns(3) and a list of almubs with multiple tuples with 3 items each

Readline() reads single line of a text file  
readlines() reads all the contents of a text file and can cause issue if the file is large   
reads() reads entire file into single variable line separators are not included like (,)  
writelines() write the contents of a text file  
write() writes the content to aa text file but only text no int float…

G = global() makes a variable global or global var\_name then use it

Import Events (.py file) and use it like Events.function\_name(par1, par2…)

Import webbrowser webbrowser.open(url)

Lambda functions are short term anonymous function. Good for short term operations   
numbers = [1, 2, 3, 4, 5]  
squared = map(lambda x: x\*\*2, numbers)  
print(list(squared))   
add = lambda x, y: x + y  
result = add(3, 5)  
print(result) instead u can even call a function by replacing the x

Filter(some\_function, some\_list) here some function can be like is even   
map(lambda x: x\*\*2, numbers)   
reduce(lambda a,b: a+b, numbers)

Functions are also objects …so it can also be stored in dictionary

Try …except(condition 1)… except(condition 2)…else(when no except is conditions are handled)…finally…  
finally is always executed

Generator function / yield : yield is the keyword is used in the generator functions, when the iteration times are not fixed so yield is used for getting the next value as it stores the previous state and returns the next each time.

Decorators are used for modifying an already existing function without changing its behavior   
import time

def time\_it(func):

    def wrapper(\*args, \*\*kwargs):

        t1= time.time()

        func(\*args, \*\*kwargs)

        t2 = time.time() - t1

        print('Time: ', t2)

    return wrapper

@time\_it

def div(a,b):

    time.sleep(1)

    print(a/b)

@time\_it

def mul(a,b):

    print(a\*b)

div(4,4)

div(4,5)

mul(3,3)

multi threading :

from threading import \*

class Hi(Thread):

    def run(self):

        for i in range(5000):

            print('Hi')

class Hello(Thread):

    def run(self):

        for i in range(5000):

            print('Hello')

t1 = Hello()

t2 = Hi()

t1.start()

t2.start()

Palindrome string[::-1]. casefold() == string.casefold() execute

Check if sentence is a palindrome read the sentence and remove all the spaces using   
for char in sentence   
if char.isaslnum() append it to a string

Fibbonachi   
curr = 0 next =1  
while True:  
curr, next = curr+next, curr