Class notes

Python lecture 1 videos

~~Clo Cob~~ | Colab (Automatically save in google drive)
↳ (code cell + Text cell)

a = 1 → int } Number | var = True } Bodeans
b = 1.0 → float | var = False
c = 'text' → string

local variable + Global variable
↓ ↓
Inside body outside all bodies

Data structure

List : list_of_numbers = [1, 2, 3, 4, 5]
print(list_of_numbers) # will print the list
list_of_numbers.remove(1) # 1 will be removed
other functions after (.).

Tuples : tuple_of_numbers = (1, 2, 3, 4, 5)
print( ) # will print tuple
Tuple will ~~demo~~ not be modified, It can not be changed

Set : fruits = {'a', 'o', 'b'}
~~print()~~ print() # It will print without following any order

Dictionary

person = {
"name": "Karim",
"height": 5.7,
"address": "Dhaka",
}

print (person['name']) # It will print Kanim
print (person['address']) # Dhaka will print

If-else

if condition == True: #checking bollolean value is right or not
    print('success') # other labels will work with indention
else:
    print('fail') # If previous condition if does not work, then

#loop

counter = 0
while counter <=10:
    print (counter) #print from 0 to 10 in one column
    counter = counter + 1 (pink)

colors = ['orange', 'yellow', [illegible], 'green']
print(colors[0])
" (" [1])
" (" [2])
" (" [3])

for x in colors:
    print(x) # printing colors with for loop

#function

def my-function():
    print('I do some addition') # Declaration

my-function() # print 'I do some addition'

~~Pass~~

```
def my_function(a,b):
    print('I do some addition')
    print(a+b)
    return 'success'
my_function(2,3) # It will print 'addition part' and also returned value will print
```

#class | objects

```
class fruit():
    #property
    def __init__(self, color, available):
        self.color = color
        self.available = available

    # function
    def eaten_or_not(self, availabe, choice):
        if choice == 'eaten':
            self.availabe = False
        else: pass    # 'pass' in single code

banana = fruit("yellow", True)

#print(banana.available) -> True

banana.eaten_or_not('availabe', 'eaten')
#print(banana.available) -> False .
```

We can create more objects.

```
orange = fruit("orange", True)
apple = fruit("Red", True)
```

```
#Inhenitamee

class Penson():
    def __init__(self, nationality):
        self.nationality =nationality
    def doing_whatth():
        print("I ame sleeping")

penson = Penson('Bangladeshi')
print(penson.nationality) # Bangladeshi
```

#doing inhenitanee

```
class Cat(Penson): #cat class inhenits Penson.
    pass  #It will do noting

catI = Cat('Hmenican')
print(catI.nationality) # Amenican
```

Libnanies

```
import math
n=3
area = math.pi *(n*n)
print(area) # 28.974...8
```

Python lecture #2

Variable → Container # Assigning on initializing

Types (number, string, boolean)

letter, digits and underscores allowed (can not start with digit)
upper + lower case

Data structure

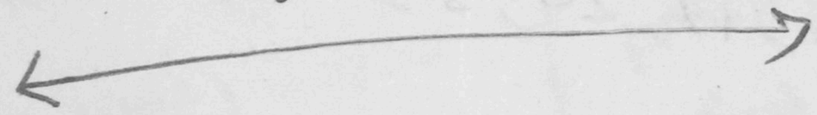
- List

List = [0, 1, 2, 3, 4, 5]
0 1 2 3 4 5

List[0] = List[0:] = [0, 1, 2, 3, 4, 5] ## start from 0 to end
List[1] = List[:] = [0, 1, 2, 3, 4, 5]
List[2]
List[3]
List[4]

List[2:4] = [2, 3] # print 2 (start) to less than 4 (end)
List[1:3] = [1, 2] # start → 1, end → 4
List[:4] = [0, 1, 2, 3] # ends in 3 less than 4 (end)

length = 5



'p' 'r' 'o' 'b' 'e'
index → 0 1 2 3 4
neg-index → -5 -4 -3 -2 -1

Slice ~~test~~ Notation (:)

↳ used to extract a substring.

Ex: Name [0:2] = = 'Fu'
Name [2:5] = = 'dge'
Name [:4] = = 'Fudg'
Name [:] = = 'Fudge'
Name [1:-1] = = 'udg'

0 1 2 3 4
F u d g e

a = [9, 8, 7, 6, 5, 4]

a[0:3] [9, 8, 7]

a[:4] [9, 8, 7, 6]

a[1:] [8, 7, 6, 5, 4]

a[:] [9, 8, 7, 6, 5, 4]

a[2:2] []

a[0:6:2] [9, 7, 5]

a[::-1] [4, 5, 6, 7, 8, 9] # Reverse form

a = [10, 12, 13, 17]

Appending values to extent a list

a = [10, 12, 13, 17]
a.append(20) # using append function
a = a + [20] # using Assignment operator

a = a + [30, 40]

print(a) # 10, 12, 13, 17, 20, 30, 40

#Extent list using slicing

a[:0] = [30]
a[:0] = [40, 50]
print(a) # . . . —

Method

append() # Add an element
elear() # Removes all element
eopy() # Returns a eopy of list
eount() # Return Total number of element
extend() # Add elements
index() # Return the inder number
insert() # Adds an element with a speeifie position
pop() # Removes the element within a speeifie position.
nemove() # Removes the finst item with a specifie value.
nevense() # Reverse the onden
sont() # sont in #seending onden

Staeks → LIFO
Queue → FIFO

```
queue = []
queue.append('a')
queue.append('b')
queue.append('e')
print("Initial queue")
print(queue)
print("\n Elements dequeued
    from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0)) # Mentioning index number
print('\nAften Removing Element')
print(queue)
```

```
staeks = []
staeks.append('a') # Add a
staeks.append('b') # Add b
staeks.append('e') # Add e
print('Initial staeks')
print(staeks) # ['a','b','e']
print('\n Element popped from staeks:')
print(staeks.pop()) # e
print(staeks.pop()) # b
print(staeks.pop()) # a
print('\n staeks aften elements are popped')
print(staeks) # Empty
```

After removeing all element queue.pop(0) with gives an ennor called IndexEnnor because queue is empty.

Tuples → Inside ()

0	1	2	3	4	5
1	2	3	4	5	6
~~-5~~	~~-4~~	~~-3~~	~~-2~~	~~-1~~	
-6	-5	-4	-3	-2	-1

Indexing Tuples

Tuple = (0,1,2,3,4,5)

Tuple[0] = 0

Touple[0:] = (0,1,2,3,4,5)

'' same as List operation.

''

Tuple[5] = [5]

#We can not manipulate Tuple elements. It is immutable in nature unlike list.

Methods

count() # count element

index() # Find an element

len() # Find lenght

min() # Find minimum

max() # Find maximum

sum() # sum all elements

sort() # sort all elements

loop tuple # loop all element in Tuple

Tuple vs List

( ) [ ]

immutable | mutable

33 methods | 46 methods

can create keys in dictionary | can not ~~create~~ use keys in ~~list~~ dictionary.

~~Mutable | Ordered~~

List	✓	✗	✓	✓
Tuple	✗	✓	✓	✓
set	✓	✗	✗	✗

	Mutable	Ordered	Indexing/Slicing	Duplicate Elements
List	✓	✓	✓	✓
Tuple	✗	✓	✓	✓
set	✓	✗	✗	✗

set

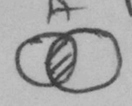
~~comment~~ within { }

S={20, 'Jessa', 35.75}

Unordered, Unchangeable, Unique elements

↓ immutable

Heterogeneous → can contain data of all types.

Set operation		
Union		= A∪B (within A or B)
Intersection		= A∩B (within A and B)
Difference		= A\B (A except B)
Symmetric Difference		= A Δ B (within A and B but not both)

~~dat~~

A = {'P', 'R', 'Q', 'S', 'T', 'A'}

B = {'P', 'J', 'S', 'R', 'Q', 'H'}

~~coA~~

A.union(B) # in code

{'R', 'H', 'J', 'P', 'R', 'A', 'Q', 'S', 'T'}

A.interseton(B) # in code

{'R', 'P', 'Q'}

Dictionary

starts with { }

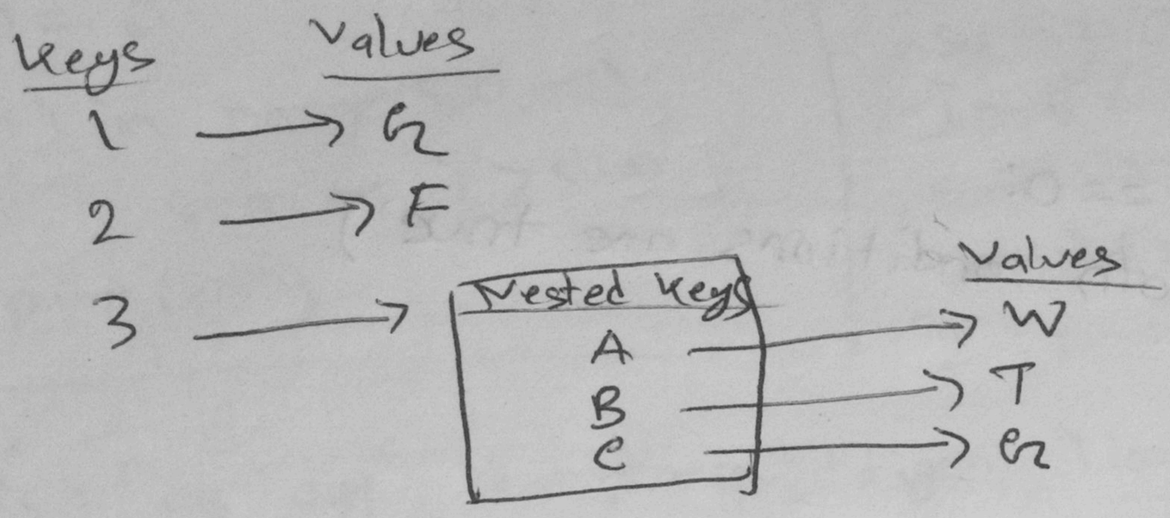
Dict = {1: "P", 2: "J"}

keys → value

1 → P

2 → J

print(Dict[1]) # printing key values



Unorden collection of items. (Unique ⇒ { }
key: ~~item~~value pair
to retrieves values when keys are known.

Arithmatic operators → +, -, *, /, //, %, **

logical operators → and, or, not

not and or ↑

comparison operators → ==, !=, >, <, >=, <=

control flow:

~~if (<exp>):~~

if exp:
    ##expression
print(...)

Exercise

i=3, J=5, K=7

```
if i<J:
    if J<K:
        i=7
    else:
        J=K
else
    if J≥K:
        i
```

```
else:
    i=K
print(I,J,K) # will print 5,5,7
```

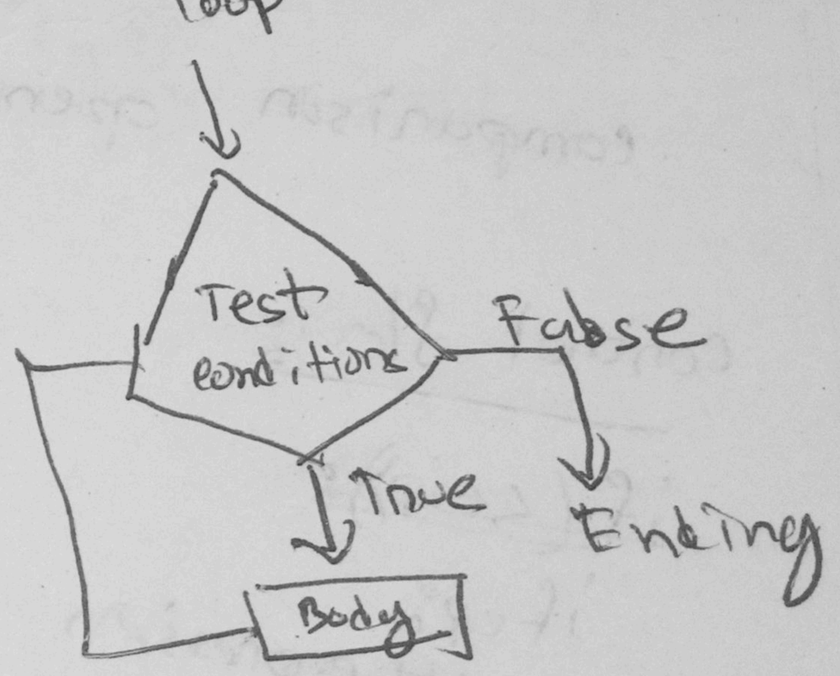
```
s=30000
t=10000
if s>t and t%2==0:
        print('Both conditions are true')

x=200
y=1000
if ((x<y) or (y%11==0)):
    print('At least one condition is true')

s=200
n=400
print("s is not equal to n") if s!=n else print("s is equal to n")
```

Loops

for and while

for var in iterable:
    - - - -
    - - - -

variable → (list, tuple)




```
n=sum(range(2,4))

while n:
    print(n, end=' ')
    n=n-1
    break
else:
    print('End', end=' ')
```

```
while(-1):
    print('OK')
```

```
for i in range(5,0,-2):
    print(i)
```

```
sum = 0
for i in range(5, 0, -2):
    sum += i > sum
print(sum)
```

```
sum = 0
i = 5
sum += i > sum
print(sum)
```

sum = 1
i = 3
sum = 2
i = 1

2

2

```
animals = ["dog", "eat", "sheep", "tigen", "lion"]

for u in animals:
    print(u)
    if u == "tigen":
        break;
```

break statement is used to breaks the infinite loop

consider **pass** keywond skip some statements in a loop body

```
count = 0
while count < 5:
    print(count)
    count += 1
```

functions

```
def greet():
    #code (define it)

greet() #calling
```

```
def add_numbers(num1, num2):
    sumd = num1 + num2
    print("sum: ", sum)

add_numbers(5, 4) # will print 9
```

```
def find_square(num):
    result = num * num
    return result

square = find_square(3)
print('square:', square) # Output will be 9
```

```
def fahn_to_celsius(temp):
    return ((temp - 32) *(5/9))
```

function object (store the result)

Arguement

fore = lambda x, y : x+y

bd

keyword

Expression

```
u = lambda a, b : a*b
print (u(3,4))
u = lambda a, b : a + b
print(u(3,4))
list_numbers = [1, 2, 3, 4]
list_numbers = map(lambda u : u*10, list numbers)

for num in list_numbers:
    print (num, end=" ") #10 20 30 40
```

** A lambda function can contain singe expression in Python*

```
def greet(name):
    print (f"Hello, {name}")

greet(name = "Aliee")
```

OOP

class Objects

properties

method

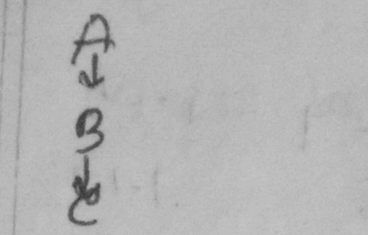
OOP (class - Object)

Encapsulation

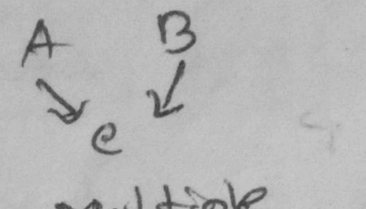
Abstraction

Polymorphism

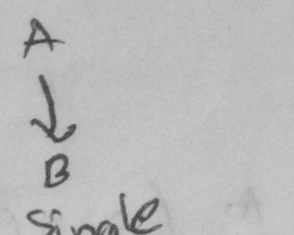
Inheritance



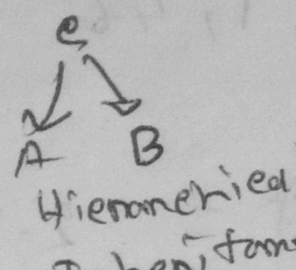
Multi level Inheritance



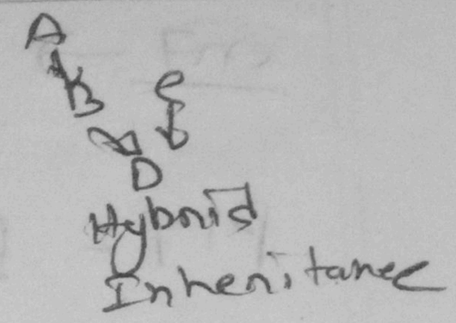
Multiple Inheritance



Single Inheritance



Hierarchical Inheritance



Hybrid Inheritance

Types of Inheritance in Python

Polymorphism → having many forms

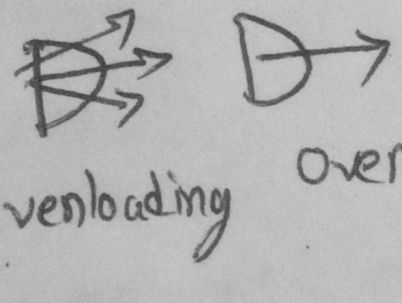
↳ same function name being used in different types (differences in data types, number of arguements)

len → string, items, keys

```
u = "Hello world"
print(len(u)) #string
```

```
mytuple = {"apple", "banana", "cherry"}
print(len(mytuple)) # items
```

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(len(thisdict)) # ~~dicts~~ keys
```



Overloading

Overriding

Overriding

```
class A:
    def first(self):
        print("First function of class A")

    def second(self):
        print("second functior of class A")

class B(A):
    #overridden
    def first(self):
        print("overriden of class A in class B")
    def display(self):
        print("Display function of child class")

if (__name__ == "__main__"):
        child_obj = B()
        print("method overridding \n")
        child_obj.first()

        A().first()
```

Abstract class → can not create instance [object]
we can inherit them

Parent

[Template for other classes with abstract methods]
↳ must be overridden

Parent class

```
def taste():
    pass

def color():
    pass
```

Fruit

```
def taste()
    print("tangy")
def color()
    print("tangerine")
```

child (Apple)

```
def taste()
    print("sweet")
def color()
    print("yellow")
```

child (Mango)

Encapsulation

class → Method / Variable

bundling of data members and functions inside a single class.

objects are self-sufficient functioning pieces and can work independently

Encapsulation
↳ provides ~~sec~~ security
↳ well defined, readable code

Member	own class	derived class	Object
Private	Y	N	N
Protected	Y	Y	N
Public	Y	Y	Y

derived
||
inherited.