GKTCS INNOVATIONS

Object Oriented Programming Using Python

Index

GKTCS INNOVATIONS

1. Introduction to Object Oriented Programming in Python

GKTCS INNOVATIONS

Object Oriented Programming is a way of computer programming using the idea of "objects" to represents data and methods. It is also, an approach used for creating neat and reusable code instead of a redundant one.

2. Difference between Object-Oriented and Procedural Oriented Programming

GKTCS INNOVATIONS

Object-Oriented Programming (OOP)	Procedural-Oriented Programming (Pop)
It is a bottom-up approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of Access modifiers 'public', private', protected'	Doesn't use Access modifiers
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs
It supports inheritance	It does not support inheritance

3. What are Classes and Objects?

GKTCS INNOVATIONS

A class is a collection of objects or you can say it is a blueprint of objects defining the common attributes and behavior. Now the question arises, how do you do that?

Class is defined under a "Class" Keyword.

Example:

```
class class1(): // class 1 is the name of the class
```

GKTCS INNOVATIONS

Creating an Object and Class in python:

Example:

```
class employee():
    def __init__(self,name,age,id,salary):   //creating a function
        self.name = name // self is an instance of a class
        self.age = age
        self.salary = salary
        self.id = id

emp1 = employee("harshit",22,1000,1234) //creating objects
emp2 = employee("arjun",23,2000,2234)
print(emp1.__dict__)//Prints dictionary
```

4. Object-Oriented Programming methodologies:

GKTCS INNOVATIONS

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

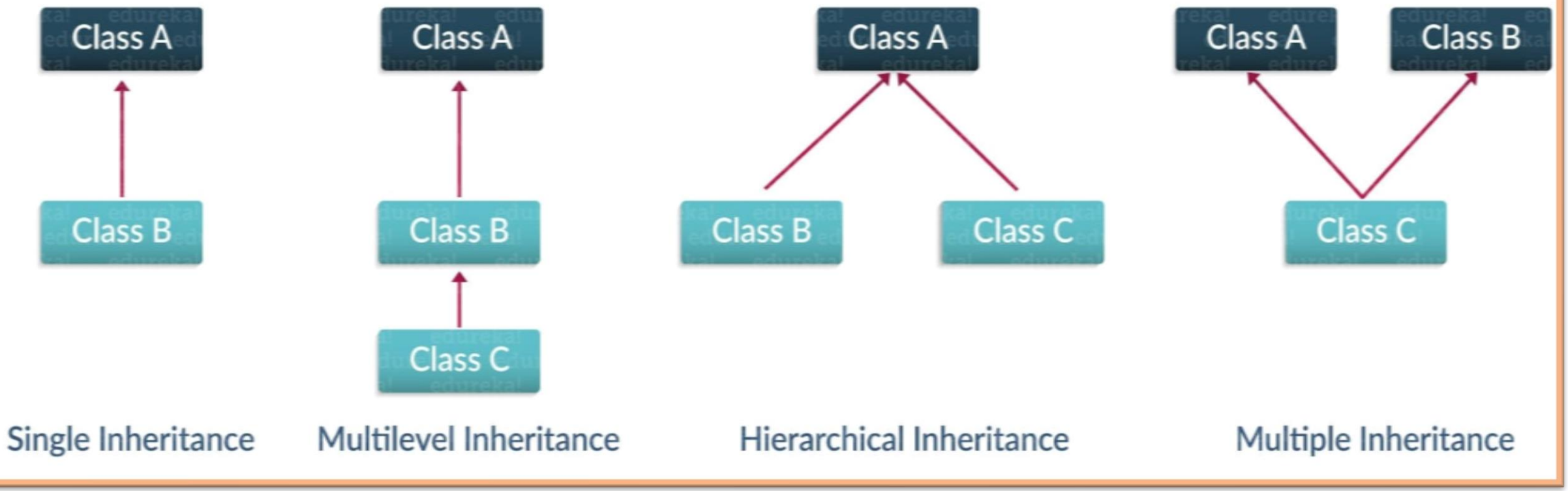
GKTCS INNOVATIONS

Inheritance:

Ever heard of this dialogue from relatives "you look exactly like your father/mother" the reason behind this is called 'inheritance'. From the Programming aspect, It generally means "inheriting or transfer of characteristics from parent to child class without any modification". The new class is called the derived/child class and the one from which it is derived is called a parent/base class.

GKTCS INNOVATIONS

Types Of Inheritance



GKTCS INNOVATIONS

Single Inheritance:

Single level inheritance enables a derived class to inherit characteristics from a single parent class.

GKTCS INNOVATIONS

Example:

```
class employee1()://This is a parent class
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

class childemployee(employee1)://This is a child class
    def __init__(self, name, age, salary,id):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id
emp1 = employee1('harshit',22,1000)
print(emp1.age)
```

Output: 22

GKTCS INNOVATIONS

Multilevel Inheritance:

Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

Example:

```
class employee()://Super class
    def __init__(self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary
class childemployee1(employee)://First child class
    def __init__(self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary
```

```
class childemployee2(childemployee1)://Second child class
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
emp1 = employee('harshit',22,1000)
emp2 = childemployee1('arjun',23,2000)

print(emp1.age)
print(emp2.age)
```

Output: 22,23

GKTCS INNOVATIONS

Hierarchical Inheritance:

Hierarchical level inheritance enables more than one derived class to inherit properties from a parent class.

Example:

```
class employee():
    def __init__(self, name, age, salary):     //Hierarchical Inheritance
        self.name = name
        self.age = age
        self.salary = salary
```

```
class childemployee1(employee):
    def __init__(self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary

class childemployee2(employee):
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
emp1 = employee('harshit',22,1000)
emp2 = employee('arjun',23,2000)
```

GKTCS INNOVATIONS

Multiple Inheritance:

Multiple level inheritance enables one derived class to inherit properties from more than one base class.

Example:

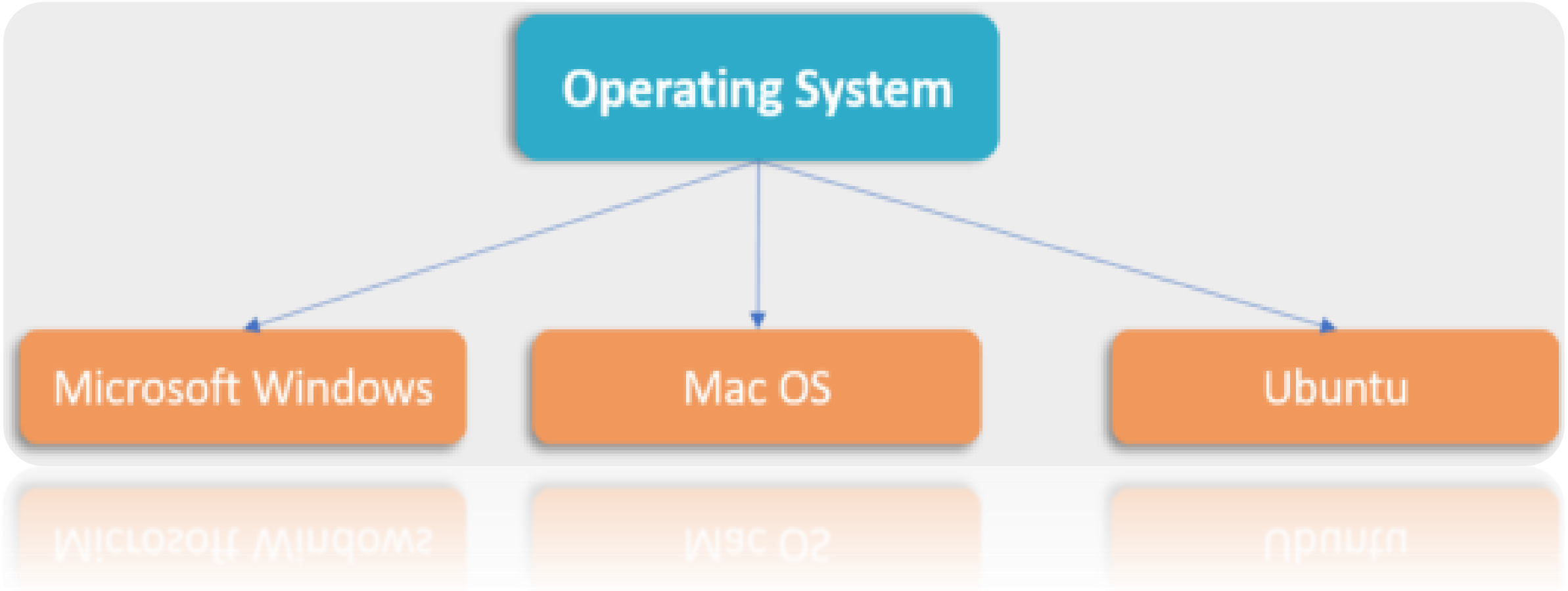
```
class employee1(): //Parent class
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
```

```
class employee2(): //Parent class
    def __init__(self,name,age,salary,id):
     self.name = name
     self.age = age
     self.salary = salary
     self.id = id

class childemployee(employee1,employee2):
    def __init__(self, name, age, salary,id):
     self.name = name
     self.age = age
     self.salary = salary
     self.id = id

emp1 = employee1('harshit',22,1000)
emp2 = employee2('arjun',23,2000,1234)
```

GKTCS INNOVATIONS

Polymorphism:

You all must have used GPS for navigating the route, Isn't it amazing how many different routes you come across for the same destination depending on the traffic, from a programming point of view this is called 'polymorphism'. It is one such OOP methodology where one task can be performed in several different ways. To put it in simple words, it is a property of an object which allows it to take multiple forms.


Operating System
Microsoft Windows
Mac OS
Ubuntu

GKTCS INNOVATIONS

Polymorphism is of two types:

- Compile-time Polymorphism
- Run-time Polymorphism

GKTCS INNOVATIONS

Compile-time Polymorphism:

A compile-time polymorphism also called as static polymorphism which gets resolved during the compilation time of the program. One common example is "method overloading"

Example:

GKTCS INNOVATIONS

```
class employee1():
    def name(self):
        print("Harshit is his name")
    def salary(self):
        print("3000 is his salary")
    def age(self):
        print("22 is his age")

class employee2():
    def name(self):
        print("Rahul is his name")
    def salary(self):
        print("4000 is his salary")
    def age(self):
        print("23 is his age")
```

```
def func(obj)://Method Overloading
    obj.name()
    obj.salary()
    obj.age()

obj_emp1 = employee1()
obj_emp2 = employee2()
func(obj_emp1)
func(obj_emp2)
```

Output:

```
Harshit is his name
3000 is his salary
22 is his age
Rahul is his name
4000 is his salary
23 is his age
```

GKTCS INNOVATIONS

Run-time Polymorphism:

A run-time Polymorphism is also, called as dynamic polymorphism where it gets resolved into the run time. One common example of Run-time polymorphism is "method overriding".

GKTCS INNOVATIONS

Example:

```
class employee():
    def __init__(self,name,age,id,salary):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id
    def earn(self):
        pass

class childemployee1(employee):
    def earn(self): //Run-time polymorphism
        print("no money")
```

```
class childemployee2(employee):
    def earn(self):
        print("has money")

c = childemployee1
c.earn(employee)
d = childemployee2
d.earn(employee)
```

Output: no money, has money

Abstraction:

Suppose you booked a movie ticket from bookmyshow using net banking or any other process. You don't know the procedure of how the pin is generated or how the verification is done. This is called 'abstraction' from the programming aspect, it basically means you only show the implementation details of a particular process and hide the details from the user.

Module 2
Theoretical Assignment :

1.Data science is a domain that deals with the collection, analysis and interpretation of data, specifically for business purposes. It involves statistics, machine learning, artificial intelligence and database systems techniques altogether. Python is one of the most popular programming languages used in data science owing to its simplicity and flexibility. In this blog lets us discuss the Role of Python in Data Science with its various types of libraries, applications, etc.

It is an interpreted high-level programming language that was created by Guido Van Rossum in 1991 at CWI (Centrum Wiskunde & Informatica), Netherlands, as an alternative to Perl and Ruby on Rails which were already popular at that time but had some limitations like lack of OOP support or slow execution speed due to interpreter nature of both languages respectively.

Python Libraries For Data Science

It is a programming language that Guido van Rossum developed in 1989. It is used for general-purpose programming, but it has also become popular in the field of Data Science because of its ease of use and flexibility.

Python libraries are tools that extend the functionality of Python and make it easier to perform specific tasks such as data manipulation or machine learning. Several libraries are available for Data Science tasks, such as Numpy, Pandas, SciPy etc., which we will discuss later in this article, along with Python Libraries and their applications in various fields like Machine Learning & Deep Learning etc.

Role Of Python In Data Science For Data Analysis And Visualization

Data analysis and visualization are essential aspects of Data Science at the present time. Presently, Python has several libraries that make it easy to analyze and visualize data. Here are some of the most commonly used libraries for data analysis and visualization in Python:

Python Training in Chennai
Exploring Datasets

Exploring datasets is an essential step in data analysis. Python’s Pandas library provided that tools for reading and writing data in various formats, such as CSV, Excel, and SQL databases. It is particularly useful for working with tabular data, such as data in spreadsheets or databases. Pandas also provide powerful tools for data exploration, cleaning, and preparation.

Data Cleaning And Preprocessing

Data cleaning and preprocessing are critical steps in data analysis. Python's Pandas library provided that data cleaning and preprocessing tools, such as removing duplicates, dealing with missing values, and transforming data. Pandas also provides powerful tools for data transformation, such as pivoting, merging, and reshaping data.

Data Wrangling And Manipulation

Data wrangling and manipulation are essential steps in data analysis. Python's NumPy library provided that tools for working with arrays, such as indexing, slicing, and reshaping arrays. NumPy also provides tools for mathematical operations on arrays, such as addition, subtraction, multiplication, and division. Python's Pandas library provides tools for data manipulation, such as selecting, filtering, and aggregating data.

Generating Statistical Reports

Generating statistical reports is a critical step in data analysis. Python's SciPy library provided that tools for statistical analysis, such as hypothesis testing, regression analysis, and cluster analysis. Python's Matplotlib library provides tools for data visualization, such as line charts, scatter plots, bar charts, and histograms. Matplotlib is particularly useful for creating high-quality visualizations for scientific publications and reports.

Graphical Representations

Graphical representations are essential for data visualization presently. Python's Seaborn library provides tools for creating statistical graphics, such as heatmaps, pair plots, and facet grids. Seaborn is particularly useful for creating complex visualizations with multiple variables. Python's Plotly library provides tools for creating interactive visualizations, such as scatter plots, line charts, and bar charts. Plotly is particularly useful for creating web-based visualizations that can be shared with others.

Practical Tasks

Task 1 Python Fundamentals

1) n = int(input("Enter a number: "))
total = sum(range(1, n + 1))
print("Sum:", total)

2. Count occurrences of a substring
text = "hello hello world"
substring = "hello"
count = text.count(substring)
print("Count:", count)

3. Count each word’s occurrences
sentence = "this is a test this is"
words = sentence.split()
word_count = {}
for word in words:
  word_count[word] = word_count.get(word, 0) + 1
print(word_count)

4. Swap first 2 characters of two strings
str1 = "abc"
str2 = "xyz"
new_str1 = str2[:2] + str1[2:]
new_str2 = str1[:2] + str2[2:]
print(new_str1 + " " + new_str2)

5. Add 'ing' or 'ly' to string
s = input("Enter a string: ")
if len(s) >= 3:
  if s.endswith("ing"):
    s += "ly"
  else:
    s += "ing"
print(s)

6. Replace 'not...poor' with 'good'
text = "This food is not that poor"
not_pos = text.find("not")
poor_pos = text.find("poor")
if not_pos != -1 and poor_pos > not_pos:
  text = text[:not_pos] + "good" + text[poor_pos+4:]
print(text)

7. GCD of two numbers
import math

```
a = 20
b = 28
print("GCD is:", math.gcd(a, b))

8. Check if list contains a sublist
main_list = [1, 2, 3, 4, 5]
sub_list = [2, 3]
print(set(sub_list).issubset(main_list))

9. Second smallest number
nums = [5, 3, 1, 4]
unique = list(set(nums))
unique.sort()
print("Second smallest:", unique[1])

10. Unique values from list
lst = [1, 2, 2, 3, 3, 4]
unique = list(set(lst))
print(unique)

11. Unzip list of tuples
pairs = [(1, 'a'), (2, 'b')]
nums, chars = zip(*pairs)
print(list(nums))
print(list(chars))

12. Convert list of tuples to dictionary
pairs = [(1, 'a'), (2, 'b')]
d = dict(pairs)
print(d)

13. Sort dictionary by value
d = {'a': 3, 'b': 1, 'c': 2}
sorted_d = dict(sorted(d.items(), key=lambda x: x[1]))
print(sorted_d)
sorted_d_desc = dict(sorted(d.items(), key=lambda x: x[1], reverse=True))
print(sorted_d_desc)
14. Top 3 values in dictionary
d = {'a': 10, 'b': 20, 'c': 5, 'd': 8}
top_3 = sorted(d.values(), reverse=True)[:3]
print(top_3)

15. Fibonacci series
n = int(input("Enter number of terms: "))
fib = [0, 1]
for i in range(2, n):
  fib.append(fib[-1] + fib[-2])
print(fib[:n])
```

16. Frequency of items using dictionary

```
lst = [1, 1, 1, 5, 3, 1, 3, 3, 1, 4, 4, 4, 2, 2, 2]
freq = {}
for item in lst:
  freq[item] = freq.get(item, 0) + 1
print(freq)
```

17. Sum of odd/even factorial series

```
import math
def odd_series(n):
  sum_series = 0
  for i in range(1, n+1, 2):
    sum_series += i / math.factorial(i)
  return sum_series

def even_series(n):
  sum_series = 0
  for i in range(2, n+1, 2):
    sum_series += i / math.factorial(i)
  return sum_series

print("Odd Series Sum:", odd_series(7))
print("Even Series Sum:", even_series(6))
```

18. Factorial using recursion

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)

print(factorial(5))
```

19. Unique values from one list

```
def unique_elements(lst):
  return list(set(lst))

print(unique_elements([1, 2, 2, 3, 4]))
```