B.Sc.(Data Science) I sem

Problem Solving and Python Programming

Unit-IV

1. Define Object Oriented Programming (OOP) and explain its concepts.

OOP is a way of organizing code that uses objects and classes to represent real-world entities and their behavior. This approach helps in building modular, maintainable and scalable applications. For example, an object could represent a person with properties like a name, age and address and behaviors such as walking, talking, breathing and running.

OOP comprises the following essential concepts:

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

➢ **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

➢ **Object:** A unique instance of a data structure that's defined by its class. An object is any entity that has attributes and behaviors. An object comprises both data members (class variables and instance variables) and methods.

➢ **Polymorphism:** Polymorphism allows us to perform a single action in different ways. It allows us to have more than one method with the same name, as long as they have different parameters.

➢ **Encapsulation:** Encapsulation is the technique of binding data and methods within a single class. Encapsulation prevents unauthorized access to the data. By defining methods to control access to attributes and its modification, encapsulation helps maintain data integrity and promotes modular, secure code.

➢ **Inheritance:** Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called a child or derived class) to inherit attributes and methods from another class (called a parent or base class). This promotes code reuse, modularity, and a hierarchical class structure.

➢ **Data abstraction:** Data abstraction is one of the most essential concepts of Python OOPs which is used to hide irrelevant details from the user and show the details that are relevant to the users. It means hiding implementation details and exposing only the essential functionality of an object.

2. **Write a program to create objects of a class.**

```
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade
    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")
student1 = Student("Mahesh", 16, "10th") # Create objects of Student class
student2 = Student("Ramesh", 17, "11th")
student1.display_info()  # Access object methods and attributes
student2.display_info()
```

3. **Write about constructor and instance in Python.**

constructor: A constructor in Python is a special method used to **initialize an object when it is created**. It allows you to assign values to the attributes of a class automatically at the time of object creation. In Python, the **constructor** is defined using the **__init__()** method. It is called automatically as soon as an object of the class is created, and it usually takes parameters to set the initial state of the object.
An example of Constructor:

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

student1 = Student("Mahesh", 16)          # __init__() is called automatically
print(student1.name, student1.age)
```

instance:

An instance of a class is a specific object created from that class. When a class acts as a blueprint, an instance is an actual object that has real values assigned to the attributes defined in the class. Each instance can have different values for its attributes, but it shares the same structure and methods defined in the class.

In the following code,

student1 = Student("Mahesh", 16) creates an instance of a class

4. **Write about Functional Programming.**

Functional Programming:

Python supports a form of programming called Functional Programming (FP) that involves programming with functions where functions can be passed, stored and returned. FP decomposes a problem into a set of functions. In FP, every function is understood solely in terms of its inputs and its outputs.

It is of four types:

- ❖ Lambda
- ❖ Iterators
- ❖ Generators
- ❖ List Comprehensions

Lambda:

Small anonymous functions can be created with the lambda keyword. Lambda functions are created without using def keyword and without a function name. They are syntactically restricted to a single expression.

The basic syntax for lambda function is:

lambda argument_list: expression

Here, lambda is a keyword, argument_list is a comma separated list of arguments, and expression is an arithmetic expression using these arguments lists. A colon separates both argument_list and expression. No need to enclose argument_list within brackets.

```
>>> c= lambda a, b: a + b
>>> c(10,20)   # Result is 30
```

Iterators:

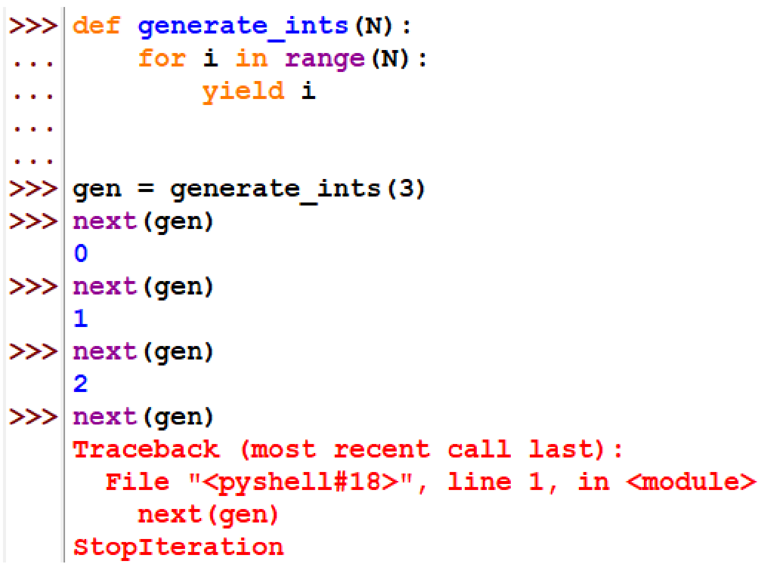
Iteration is a general term for taking each item of something, one after another. iterable and iterator have specific meanings. An **iterable** is an object that has an __iter__() method that returns an iterator. **Lists, dictionaries, tuples and strings are iterable in Python**. An **iterator** is an object with a __next__() method. Whenever a for loop is used in Python, the __next__() method is called automatically to get each item from the iterator, thus going through the process of iteration

```
>>>str="B.Sc."
>>>objs=iter(str)
>>>next(objs)
'B'
>>>next(objs)
'.'
>>>next(objs)
```

```
'S'
>>>next(objs)
'c'
>>>next(objs)
'.'
>>>next(objs)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    next(objs)
StopIteration
```

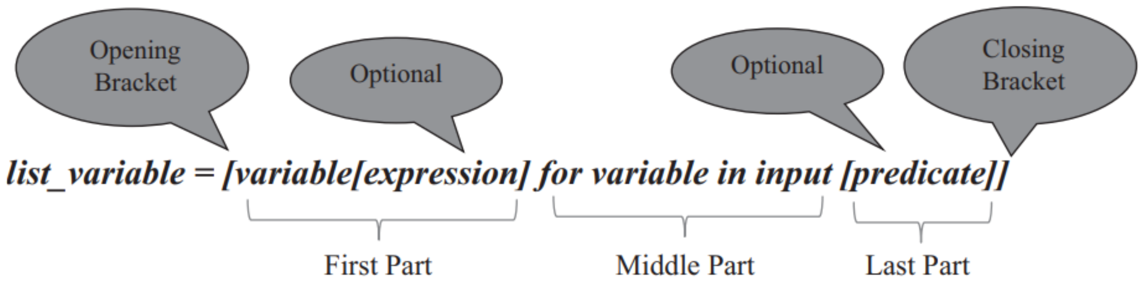
Generators:

Generators are a special c ting iterators. Regular functions compute a value and return it, but generators return an iterator that returns a stream of values. A generator function does not include a return statement.

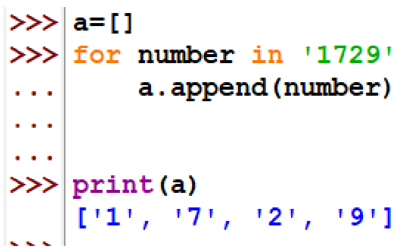


```
>>> def generate_ints(N):
...     for i in range(N):
...         yield i
...
...
>>> gen = generate_ints(3)
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    next(gen)
StopIteration
```

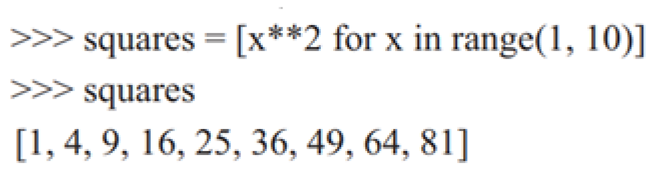
List Comprehensions:
List comprehensions provide a concise way to create lists. Common applications of list comprehensions are to make new lists where each element is the result of some operation applied to each member of another sequence or iterable or to create a subsequence of those elements that satisfy a certain condition.



A list comprehension consists of brackets containing a variable or expression (First Part) followed by a for clause (Middle Part), then predicate True or False using an if clause (Last Part). The components expression and predicate are optional. The new list resulting from evaluating the expression in the context of the for and if clauses that follow it will be assigned to the list variable. The variable represents members of input.
The order of execution in a list comprehension is
(a) If the if condition is not specified, then Middle Part and First Part gets executed
(b) If the if condition is specified, then the Middle Part, Last Part, and First Part gets executed.



```
>>> a=[]
>>> for number in '1729':
...     a.append(number)
...
...
>>> print(a)
['1', '7', '2', '9']
```



```
>>> squares = [x**2 for x in range(1, 10)]
>>> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```