1. Introduction

Python uses object-oriented Programming (OOP) paradigm. The OOP paradigm uses concepts of *classes* and *objects* in programming. It also aims to implement real-world concepts such as *inheritance, polymorphism, encapsulation* etc. in the programming. The main concept of OOP is to bind both data and the functions that work on the data together as a single unit so that no other part of the code can access this data. The combined unit of data and function is called the object.

2. Object-oriented Principles

Object-oriented programming languages use different concepts such as object, class, inheritance, polymorphism, abstraction, and encapsulation.

2.1. Classes and Objects

The concept of *classes* and *objects* is the heart of the OOP. A class is a framework that specifies what data and what functions will be included in objects of that class. It serves as a plan or blueprint from which individual objects are created. An object is a software bundle of related data and functions created from a class. For example, we can think of the class **Student**. This class has some data like name, address, marks, and so on. Similarly it can have the functions like find_percentage(), find_division(), etc. Now here we can see that from this class, we can create any number of objects of the type Student. In programming we say that an object is an instance of the class.

2.2. Encapsulation

Encapsulation is the process of combining the data and functions into a single framework called class. Encapsulation helps preventing the modification of data from outside the class by properly assigning the access privilege to the data inside the class. So, the term ***data hiding*** is possible due to the concept of encapsulation, since the data are hidden from the outside world.

2.3. Inheritance

Inheritance is the process of acquiring certain attributes and behaviors from parents. For examples, cars, trucks, buses, and motorcycles inherit all characteristics of vehicles. Object-oriented programming allows classes to inherit commonly used data and functions from other classes. If we derive a class from another class, some of the data and methods can be inherited so that we can reuse the already written and tested code in our program, simplifying our program.

2.4. Polymorphism

Polymorphism means the quality of having more than one form. The representation of different behaviors using the same name is called polymorphism. However, the behavior depends upon the attribute the name holds at particular moment. For example, if we have the behavior called communicate for the object vertebrates, then the communicate behavior applied to objects say dogs is quite different from the communicate behavior to objects human. So here the same name communicate is used in multiple process one for human communication, what we simply call talking. The other is used in communication among dogs, we may say barking, definitely not talking.

2.5. Abstraction

Abstraction is the essence of OOP. Abstraction means the representation of the essential features without providing the internal details and complexities. In OOP, abstraction is achieved by the help of class, where data and functions are combined to extract the essential features only. For example, if we represent chair as an object it is sufficient for us to understand that the purpose of chair is to sit. So, we can hide the details of construction of chair and things required to build the chair.

3. Defining Class

Everything in Python is an object. An object has states (variables) and behaviors (methods). To create an object, you define a class first. And then, from the class, you can create one or more objects. The objects are instances of a class. Class is a blueprint or code template for object creation. Using a class, you can create as many objects as you want.

In Python, class is defined by using the **class** keyword. The syntax to create a class is given below.

```
class ClassName:
    <statement-1>
    <statement-2.
    .
    .
    .
    <statement-N>
```

For example, to create a Student class with name and age as instance variables and display() as an instance method, we write,

```
class Student:
    def __init__(self, n, a):
        self.name = n
        self.age = a
    def display(self):
        print(self.name, ' is ', self.age, ' years old.')
s1 = Student('Ram', 34)
s2 = Student('Shyam', 39)
s1.display()
s2.display()
```

To create an object from the Student class, you use the class name followed by parentheses(), like calling a function. For example,

```
s1 = Student('Ram', 34)
```

Python also allows us to delete objects. We can delete objects created from a class using **del** keyword. For example,

```
del s1
```

3.1. Adding Instance Variables

If the value of a variable varies from object to object, then such variables are called instance variables. For every object, a separate copy of the instance variable will be created. Instance variables are not shared by objects. Every object has its own copy of

the instance variable. This means that for each object of a class, the instance variable value is different. Instance variables are declared inside a method using the **self** (recommended convention) keyword. The following code defines the Student class with two instance variables name and age. Here, *name* and *age* are instance variables.

```
class Student:
    def __init__(self, n, a): # name and age are instance variables
        self.name = n
        self.age = a
```

When you create a Student object, Python automatically calls the __init__() method to initialize the instance attributes. In the __init__() method, the **self** is the instance of the Student class.
To access an instance variable through objects, you use the dot notation or getattr() method. For example,

```
s1 = Student('Ram', 34)
s2 = Student('Shyam', 39)
print(s1.name)
print(getattr(s2, 'name'))
```

We can also add instance variables to individual objects after creating them. We cannot add an instance variable to a class because instance variables belong to objects. Adding an instance variable to one object will not be reflected in remaining objects because every object has a separate copy of the instance variable. For example,

```
s1 = Student('Ram', 34)
s2 = Student('Shyam', 39)
s1.address = 'Kathmandu' # adding instance variable address to the object s1
print(s1.address)
```

To delete instance variables of individual objects, we use the **del** statement or **delattr()** function. For example,

```
s1 = Student('Ram', 34)
s2 = Student('Shyam', 39)
del s1.name # deleting name variable from object s1
delattr(s2, 'age') # deleting age variable from object s2
```

Remember: You can also add instance variables to your class using instance method and without using the __init__() method. For example,

```
class Student:
    def set_values(self, n, a):
        self.name = n
        self.age = a
s = Student()
s.set_values("Ram", 22)
```

3.2. Adding Instance Methods

Instance methods are used to access or modify the instance variables. To call an instance method, we use dot notation. If we use instance variables inside a method,

such methods are called instance methods. It must have a self as the first parameter to refer to the current object. For example,

```
class Student:
    def __init__(self, n, a):
        self.name = n
        self.age = a
    def display(self): # instance method
        print(self.name, ' is ', self.age, ' years old.')
s1 = Student('Ram', 34)
s2 = Student('Shyam', 39)
s1.display()
s2.display()
```

Python also allows us to add or delete instance methods to individual objects. When we add instance methods to an object, other objects don't have access to that method. For example,

```
import types
class Student:
    def __init__(self, n, a):
        self.name = n
        self.age = a
    def display(self): # instance method
        print(self.name, ' is ', self.age, ' years old.')
def welcome(self):
    print('Hello', self.name, 'welcome to Kathmandu.')
s1 = Student('Ram', 34)
s2 = Student('Shyam', 39)
s1.welcome = types.MethodType(welcome, s1) # adding instance method welcome() to s1
s1.welcome()
del s1.welcome # delattr(s1, 'welcome')
```

3.3. Adding Class Variables

Unlike instance variables, class variables are shared by all instances of the class. They are helpful if you want to define class constants or variables that are common to all objects of the class. A class variable is declared inside of class, but outside of any instance method. For example,

```
class Student:
    school_name = 'ABC School' # class variable
    def __init__(self, n, a):
        self.name = n
        self.age = a
    def display(self):
        print(self.name, ' is ', self.age, ' years old.')
s1 = Student("Ram", 24)
s2 = Student("Shyam", 29)
print(s1.name, s1.age, s1.school_name)
print(s2.name, s2.age, s2.school_name)
```

Python also allows us to add or delete class variables to the class after creating the class. When we add class variables, these variables are shared by all instances of the class. For example,

```
class Student:
    school_name = 'ABC School'
    def __init__(self, n, a):
        self.name = n
        self.age = a
    def display(self):
        print(self.name, ' is ', self.age, ' years old.')
s1 = Student('Ram', 24)
s2 = Student('Shyam', 29)
Student.country = 'Nepal' # adding class variable
print(s1.name, s1.age, s1.school_name, s1.country)
print(s2.name, s2.age, s2.school_name, s2.country)
del Student.country # removing class variable
# delattr(Student, 'country')
```

3.4. Adding Class Methods

Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a class method using the **@classmethod** decorator or **classmethod()** function. We use the **cls** (recommended convention) keyword as a first parameter in the class method to access class variables. Therefore, the class method gives us control of changing the class state. The class method can only access the class attributes, not the instance attributes.

Example: Using **@classmethod** decorator

```
class Student:
    school_name = 'ABC School'
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print(self.name, ' is ', self.age, ' years old.')
    @classmethod
    def change_school(cls, sn):
        cls.school_name = sn
s1 = Student('Ram', 24)
s2 = Student('Shyam', 29)
print(s1.name, s1.age, s1.school_name)
print(s2.name, s2.age, s1.school_name)
Student.change_school('XYZ School')
print(s1.name, s1.age, s1.school_name)
print(s2.name, s2.age, s1.school_name)
```

Example: Using **classmethod()** function

```
class Student:
    school_name = 'ABC School'
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age
    def display(self):
        print(self.name, ' is ', self.age, ' years old.')
    def change_school(cls, sn):
        cls.school_name = sn
s1 = Student('Ram', 24)
s2 = Student('Shyam', 29)
print(s1.name, s1.age, s1.school_name)
print(s2.name, s2.age, s2.school_name)
Student.change_school = classmethod(Student.change_school)
Student.change_school('XYZ School')
print(s1.name, s1.age, s1.school_name)
print(s2.name, s2.age, s2.school_name)
```

Python also allows us to add or delete class methods to the class after creating the class. For example,

```
class Student:
    school_name = 'ABC School'
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print(self.name, ' is ', self.age, ' years old.')
def change_school(cls, sn):
    cls.school_name = sn
s1 = Student('Ram', 24)
s2 = Student('Shyam', 29)
print(s1.name, s1.age, s1.school_name)
print(s2.name, s2.age, s2.school_name)
Student.change_school = classmethod(change_school) # Adding class method
Student.change_school('XYZ School')
print(s1.name, s1.age, s1.school_name)
print(s2.name, s2.age, s2.school_name)
del Student.change_school # deleting class method
# delattr(Student, 'change_school')
```

3.5. Adding Static Methods

A static method is a general utility method that performs a task in isolation. A static method is bound to the class and not the object of the class. Therefore, we can call it using the class name. A static method doesn't have access to the class and instance variables because it does not receive an implicit first argument like self and cls. Therefore, it cannot modify the state of the object or class. The static method can be called using ClassName.method_name() as well as by using an object of the class. Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a static method using the **@staticmethod** decorator or **staticmethod()** function.

Example: Using **@staticmethod** decorator

```
class Employee:
```

```
    @staticmethod
    def sample(x):
        print('Inside static method', x)
# call static method
Employee.sample(10)
# can be called using object
emp = Employee()
emp.sample(10)
```

Example: Using **staticmethod()** function

```
class Employee:
    def sample(x):
        print('Inside static method', x)
Employee.sample = staticmethod(Employee.sample)
# call static method
Employee.sample(10)
# can be called using object
emp = Employee()
emp.sample(10)
```

4. Constructors

In object-oriented programming, a constructor is a special method used to create and initialize an object of a class. When we create objects in Python, the __new__ method creates objects internally. And, using the __init__ method we can implement constructors to initialize objects. The __init__ method is actually a magic method, which is run right after the object is created. For example,

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

Types of constructors:

- **Default constructor:** The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument (self) which is a reference to the instance being constructed.
- **Parameterized constructor:** Constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

5. Method Overloading

Method overloading is the process of defining two or more methods with the same name but with different numbers of parameters or different types of parameters, or both. Like other languages do, python does not support method overloading by default. But there are different ways to achieve method overloading in Python. The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method. In the code below, we have defined two add methods but we can only use the second add method. We may define many methods of the same name and different arguments, but we can only use the latest defined method.

```
def add(a, b):
    return a + b
def add(a, b, c):
    return a + b + c
print("Sum =", add(4, 5, 7))
# print("Sum =", add(4, 5)) # will produce an error
```

Thus, to overcome the above problem we can use different ways to achieve the method overloading.

- **Method 1 (Not the efficient method):** By adding * before the parameter name in the function definition. For example,

```
def add(*a):
    sum = 0
    for x in a:
        sum = sum + x
    return sum
print("Sum =", add(2, 3))
print("Sum =", add(5, 6.7, 8))
```

We can also use ** before the parameter name in the function definition as given in the example below.

```
def add(**a):
    sum = 0
    for x in a:
        sum = sum + a[x]
    return sum
print("Sum =", add(a = 2, b = 3))
print("Sum =", add(a = 5, b = 6.7, c = 8))
```

- **Method 2 (Not the efficient one):** Using default arguments as given in the example below.

```
def add(a = 0, b = 0, c = 0):
    return a + b + c
print("Sum =", add(2))
print("Sum =", add(2, 3))
print("Sum =", add(4, 7, 2.5))
```

- **Method 3 (Efficient one):** By using multiple dispatch decorator as given in the example below.

```
from multipledispatch import dispatch
@dispatch(int, int)
def add(a, b):
    return a + b
@dispatch(int, int, int)
def add(a, b, c):
    return a + b + c
@dispatch(float, float, float)
def add(a, b, c):
    return a + b + c
print("Sum =", add(2, 3))
print("Sum =", add(2, 3, 2))
```

```
print("Sum =", add(2.2, 3.4, 2.3))
```

6. Inheritance

Inheritance enables extending the capability of an existing class to build a new class, instead of building from scratch. The existing class is called base or parent class, while the new class is called child or sub class. Inheritance comes into picture when a new class possesses the 'IS A' relationship with an existing class.

The child class inherits data definitions and methods from the parent class. This facilitates the reuse of features already available. The child class can add a few more definitions or redefine a base class method. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

This feature is extremely useful in building a hierarchy of classes for objects in a system. It is also possible to design a new class based upon more than one existing classes. This feature is called multiple inheritance.

While defining the child class, the name of the parent class is put in the parentheses in front of it, indicating the relation between the two. Instance attributes and methods defined in the parent class will be inherited by the object of the child class. The general mechanism of establishing inheritance is illustrated below.

```
class parent:
   statements
class child(parent):
   statements
```

For example,

```
class Person:
   def __init__(self, fname, lname):
      self.firstname = fname
      self.lastname = lname
   def printname(self):
      print(self.firstname, self.lastname)
class Student(Person):
   def __init__(self, fname, lname, year):
      super().__init__(fname, lname) #Person.__init__(self, fname, lname)
      self.graduationyear = year
   def welcome(self):
      print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)
s = Student("Mike", "Olsen", 2019)
s.welcome()
s.printname()
print(s.firstname)
```

Remember: Every class in Python is derived from the **object** class. It is the most base class in Python. So technically, all other classes, either built-in or user-defined, are derived classes and all objects are instances of the object class.

7.Types of Inheritance

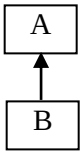
A class can inherit members from one or more classes and from one or more levels. On the basis of this concept, inheritance may take following different forms.

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance

7.1. Single Inheritance

In single inheritance, a class is derived from only one existing class. It enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

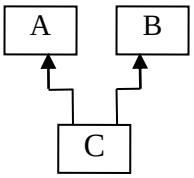
```
class A:
    members of A
class B (A):
    own members of B
```



7.2. Multiple Inheritance

When a class is derived from more than one base class, this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class.

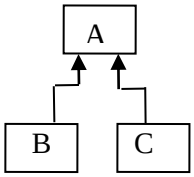
```
class A:
    members of A
class B:
    members of B
class C (A, B):
    own members of C
```



7.3. Hierarchical Inheritance

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance.

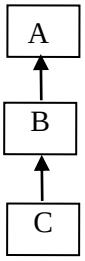
```
class A:
    members of A
class B (A):
    own members of B
class C (A):
    own members of C
```



7.4. Multilevel Inheritance

The mechanism of deriving a class from another subclass class is known as multilevel inheritance. The process can be extended to an arbitrary number of levels.

```
class A:
    members of A
class B (A):
    own members of B
```



```
class C (B):
    own members of C
```

8. Method Overriding

Method overriding allows a child class to provide a specific implementation of a method that is provided by one of its parent classes. For example,

```
class Employee:
   def __init__(self, name, base_pay):
      self.name = name
      self.base_pay = base_pay
   def get_pay(self):
      return self.base_pay
class SalesEmployee(Employee):
   def __init__(self, name, base_pay, sales_incentive):
      super().__init__(name, base_pay)
      self.sales_incentive = sales_incentive
   def get_pay(self):
      return self.base_pay + self.sales_incentive
john = SalesEmployee('John', 5000, 1500)
print(john.get_pay())
```

9. Access Modifiers

All members (variables and methods) in a Python class are **public** by default. Public members can be accessed from outside the class environment. For example,

```
class Student:
   def __init__(self, n, a):
      self.name = n
      self.age = a
s = Student("Ram", 29)
print(s.name)
```

Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it. This enables specific resources of the parent class to be inherited by the child class. Python's convention to make members **protected** is to add a prefix _ (single underscore) to it. This is just a convention and we can still access these members from outside the class environment. Hence, the responsible programmer would refrain from accessing and modifying instance variables prefixed with _ from outside its class. For example,

```
class Student:
   def __init__(self, n, a):
      self._name = n
      self.age = a
s = Student("Ram", 29)
print(s._name)
```

The double underscore __ prefixed to a variable makes it **private**. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError. For example,

```
class Student:
    def __init__(self, n, a):
        self.__name = n
        self.age = a
s = Student("Ram", 29)
print(s.__name) # Error
```

Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the member with a single or double underscore to emulate the behavior of protected and private access specifiers respectively.
Python performs name mangling of private variables. Every member with a double underscore will be changed to _class__member. So, it can still be accessed from outside the class, but the practice should be refrained. For example,

```
class Student:
    def __init__(self, n, a):
        self.__name = n
        self.age = a
s = Student("Ram", 29)
print(s._Student__name)
```

10. Abstract Class

In object-oriented programming, an abstract class is a class that cannot be instantiated. To use features of the abstract class, we have to create a subclass from this abstract class. Typically, abstract classes are used to create a blueprint for other classes.
A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation.
Python doesn't directly support abstract classes. But it does offer a module that allows you to define abstract classes. To define an abstract class, you use the abc (abstract base class) module. The abc module provides you with the infrastructure for defining abstract base classes. For example,

```
from abc import ABC, abstractmethod
class Person(ABC): # abstract class
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname
    @abstractmethod
    def display(self): # abstract method
        pass
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
    def display(self):
```

```
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
# p = Person('Mike', 'Olsen') # Error
s = Student("Mike", "Olsen", 2019)
s.display()
```

11. Operator Overloading

Operator overloading in Python allows the same operator to have different meaning according to the context. For example, to overload the + operator, we will need to implement __add__() function in the class as given below.

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return f"({self.x},{self.y})"
    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1 + p2)
```

What actually happens is that, when we use p1 + p2, Python calls p1.__add__(p2) which in turn is Point.__add__(p1,p2). After this, the addition operation is carried out the way we specified.

Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	**Expression**	**Internally**
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)

Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

12. Magic Methods

Magic methods in Python are the special methods that start and end with the double underscores. They are also called **dunder** methods. Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action. For example, when we add two numbers using the + operator, internally, the __add__() method will be called. The __init__() method will be called automatically when we create object of a class.

Built-in classes in Python define many magic methods. Use the dir() function to see the number of magic methods in class. For example, dir(int) lists all the attributes and methods defined in the int class.

13. Exception Handling

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error. When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

If you have some suspicious code that may raise an exception, you can write your program by placing the suspicious code in a **try** block. After the try block, include an **except** block to handles the exception.

You can also use **else** block and **finally** block along with try and except blocks. You can use the **else** block to define a block of code to be executed if no errors were raised. The **finally** block, if specified, will be executed regardless if the try block raises an error or not. For example,

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
try:
    c = a / b
    print(c)
except ZeroDivisionError as e:
    print(e)
else:
    print("Nothing went wrong")
finally:
    print("This is finally block")
```

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions. For example,

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a / b
```

```
    print(c)
except ValueError as v:
    print(v)
except ZeroDivisionError as e:
    print(e)
else:
    print("Nothing went wrong")
finally:
    print("This is finally block")
```

You can also use the same except statement to handle multiple exceptions. For example,

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a / b
    print(c)
except (ValueError, ZeroDivisionError) as v:
    print(v)
```

You can also use the except block with no exceptions defined. This kind of except block catches all the exceptions that occur. Using this kind of except block is not considered a good programming practice though it catches all exceptions. This style does not make the programmer identify the root cause of the problem that may occur. For example,

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a / b
    print(c)
except:
    print("There is exception")
```

13.1. Raise an Exception

Python also provides the **raise** keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution. The following code accepts a number from the user. The try block raises a ValueError exception if the number is outside the allowed range.

```
try:
    x=int(input('Enter a number upto 100: '))
    if x > 100:
        raise ValueError(x)
except ValueError:
    print(x, "is out of allowed range")
else:
    print(x, "is within the allowed range")
```

13.2. Creating Custom Exceptions

To create a custom exception class, you define a class that inherits from the built-in **Exception** class or one of its subclasses such as ValueError class.
Suppose you need to develop a program takes age input. The minimum and maximum values of age are 1 and 150. If user enters a value that is not in this range, you want to raise a custom exception AgeError.

```
class AgeError(Exception):
    def __init__(self, a):
        self.age = a
    def __str__(self):
        return f'{self.age} is not in a valid age'
def age_test(age):
    if age < 1 or age > 150:
        raise AgeError(age)
age = int(input("Enter age:"))
try:
        age_test(age)
except AgeError as v:
    print(v)
else:
    print(age, "is a valid age")
```

14. Modules and Packages

14.1. Modules

A module is a piece of software that has a specific functionality. We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code. We can define our most used functions in a module and import it, instead of copying their definitions into different programs.
A Python module is a file that contains Python code. To create a module just save the code you want in a file with the file extension .py. For example, consider a file example.py with the following code.

```
def add(a, b):
    return a + b
def mul(a, b):
    return a * b
```

After creating the module, we can use the module by using the import statement as follows.

```
import example
print(example.add(5, 2))
```

You can create an alias when you import a module, by using the **as** keyword. For example,

```
import example as e
print(e.add(5, 2))
```

We can also import specific names from a module without importing the module as a whole. For example.

```
from example import mul
print(mul(5, 2))
```

We can import all names(definitions) from a module using the asterisk (*). For example,

```
from example import*
print(add(5, 2))
print(mul(5, 2))
```

There is a built-in function to list all the function names (or variable names) in a module. For example,

```
import example
print(dir(example))
```

Remember: Python has tons of standard modules. These files are in the Lib directory inside the location where you installed Python. Standard modules can be imported the same way as we import our user-defined modules. Some common standard modules are: os, sys, math, random, collection etc.

14.2. Packages

We don't usually store all of our files on our computer in the same location. We use a well-organized hierarchy of directories for easier access. Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear. Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.

If a file named __init__.py is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

Suppose we have a package **tu.sms** and a module **example.py** inside **sms** folder with the code below.

```
def add(a, b):
    return a + b
def mul(a, b):
    return a * b;
```

Now, we can import and use the **example** module in our program using the package name. For example,

```
import tu.sms.example
print(tu.sms.example.add(5, 7))
```

We can also use module name without package prefix as given below.

```
from tu.sms import example
print(example.add(5, 7))
```

You can also create an alias when you import a module, by using the **as** keyword. For example,

```
from tu.pmc import example as e
```

print(e.add(5, 7))

We can also import specific names from a module without importing the module as a whole. For example.

from tu.sms.example import add
print(add(5, 7))

We can import all names(definitions) from a module using the asterisk (*). For example,

*from tu.sms.example import**
print(add(5, 7))