

Python RegEx

A regular expression is a series of characters used to search or find a pattern in a string. In other words, a regular expression is a special sequence of characters that form a pattern. The regular expressions are used to perform a variety of operations like searching a substring in a string, replacing a string with another, splitting a string, etc.

The Python programming language provides a built-in module `re` to work with regular expressions. The `re` is a built-in module that gives us a variety of built-in methods to work with regular expressions. In Python, the regular expression is known as RegEx in short form.

When we want to use regular expressions, we must import the `re` module. See the example below.

Example

```
import re
result = re.search('smart', 'www.btechsmartclass.com')
if result:
    print('Match found!')
else:
    print('Match not found!!')
```

Creating Regular Expression

The regular expressions are created using the following.

1. Metacharacters
2. Special Sequences
3. Sets

Metacharacters:

Metacharacters are the characters with special meaning in a regular expression. The following table provides a list of metacharacters with their meaning.

Metacharacters	Meaning
[]	A set of characters
\	A special sequence begin
.	Any character excluding newline
^	Pattern starts with
\$	Pattern ends with
*	Zero or more characters
+	One or more characters
{ }	Exactly the specified number of characters
	Either or
()	Grouping

Special Sequences:

A special sequence is a character prefixed with `\`, and it has a special meaning. The following table gives a list of special sequences in Python with their meaning.

Special Sequences	Meaning
\A	the specified characters are at the beginning of the string
\b	the specified characters are at the beginning or at the end
\B	the specified characters are present, but NOT at the beginning or at the end
\d	the string contains digits
\D	the string does not contain digits
\s	the string contains a white space character
\S	the string does not contain a white space character
\w	the string contains any characters from a to Z, digits from 0-9, and the underscore _ character
\W	the string does not contain any characters from a to Z, digits from 0-9, and the underscore _ character
\Z	the specified characters are at the end of the string

Sets:

A set is a set character enclosed in [], and it has a special meaning. The following table gives a list of sets with their meaning.

Set	Meaning
[aeiou]	Matches with one of the specified characters are present
[d-s]	Matches with any lower case character from d to s
[^aeiou]	Matches with any character except the specified
[1234]	Matches with any of the specified digit
[3-8]	Matches with any digit from 3 to 8
[a-zA-Z]	Matches with any alphabet, lower or UPPER

Built-in methods of re module:

The re module provides the following methods to work with regular expressions.

```
search()
findall()
sub()
split()
```

search() in Python:

The search() method of re object returns a Match object if the pattern found in the string. If there is more than one occurrence, it returns the first occurrence only.

Example:

```
import re
print(re.search('program', 'www.pythonprogramming.com'))
print(re.search('[pig]', 'www.pythonprogramming.com'))
```

output:

```
<re.Match object; span=(10, 17), match='program'>
```

```
<re.Match object; span=(4, 5), match='p'>
```

There are the following methods associated with the search.

1. span(): It returns the tuple containing the starting and end position of the match.
2. string(): It returns a string passed into the function.
3. group(): The part of the string is returned where the match is found.
4. **import re**

example:

```
str = "How are you. How is everything"
matches = re.search("How", str)
print(matches.span())
print(matches.group())
print(matches.string)
```

Output:

```
(0, 3)
```

```
How
```

```
How are you. How is everything
```

findall() in Python:

The findall() method of re object returns a list of all occurrences.

Example:

```
import re
print(re.findall('program', 'www.pythonprogramming.com'))
print(re.findall('[pot]', 'www.pythonprogramming.com'))
```

output:

```
["program,]
```

```
['p','t','o','p','o','o']
```

sub() in Python:

The sub() method of re object replaces the match pattern with specified text in a string. The syntax of sub() method is sub(pattern, text, string).

The sub() method does not modify the actual string instead, it returns the modified string as a new string.

Example:

```
import re
webStr = 'www.pythonprogramming.com'
print(re.sub('.com', '.in', webStr))
print(webStr)
```

output:

```
www.pythonprogramming.in
```

split() in Python:

The **split()** method of **re** object returns a list of substrings where the actual string is being split at each match.

Example

```
import re
webStr = 'www.pythonprogramming.com'
print(re.split('\.', webStr))
```

output:
[www.pythonprogramming.com]

Object Oriented Programming in Python

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. the program is divided into self-contained objects or several mini-programs. Every Individual object represents a different part of the application having its own logic and data to communicate within themselves.

Difference between Object-Oriented and Procedural Oriented Programming

Object-Oriented Programming (OOP)	Procedural-Oriented Programming (Pop)
It is a bottom- approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of <i>Access modifiers</i> ‘ public’, private’, protected’	Doesn’t use <i>Access modifiers</i>
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

What is Class:

- In Python everything is an object. To create objects we required some Model or Plan or Blue Print, which is nothing but class.
- We can write a class to represent properties (attributes) and actions (behaviour) of object.
- Properties can be represented by variables
- Actions can be represented by Methods.

We can define a class by using class keyword.

Syntax:

```
class className:  
    """ documentation string """
```

Example:

```
class student(): //student is the name of the class
```

Objects:

Objects are an instance of a class. It is an entity that has state and behaviour.

State represents the attributes and behaviour represents the functionality or actions

In a nutshell, it is an instance of a class that can access the data.

For example students is a object

State: attributes for students---- name, age, gender, mobile no., etc.

Behaviour: functionality of student—reading, learning, attend_classes, etc.

Syntax to create object: referencevariable = classname()

Example: s = student()

What is Reference Variable:

The variable which can be used to refer object is called reference variable. By using reference variable, we can access properties and methods of object.

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

```
        self.marks=marks
def talk(self):
    print("Hello My Name is:",self.name)
    print("My Rollno is:",self.rollno)
    print("My Marks are:",self.marks)
s1=Student("VIVAN",1219,75)
```

```
s1.talk()
```

output: Hello My Name is:VIVAN

My Rollno is: 1219

My Marks are: 75

Self variable:

Self is the default variable or is an implicit variable which is always pointing to current object (like this keyword in Java) provided by PVM. By using self we can access instance variables and instance methods of object.

Note:

1. self should be first parameter inside constructor

def __init__(self):

2. self should be first parameter inside instance methods

def talk(self):

Example:

```
class Student:
    def __init__(self,name,rollno,marks):
        self.name=name
        self.rollno=rollno
        self.marks=marks
    def display(self):
        print("address of self object is:",id(self))
        print("Hello My Name is:",self.name)
        print("My Rollno is:",self.rollno)
        print("My Marks are:",self.marks)
s1=Student("VIVAN",1219,75)
print("address of s1 object is: ",id(s1))
s1.display()
print("***40)
s2=Student("PRASANNA",1220,80)
print("address of s2 object is: ",id(s2))
s2.display()
print
```

output:

```
address of s1 object is: 1803294529928
address of self object is: 1803294529928
Hello My Name is: VIVAN
My Rollno is: 1219
My Marks are: 75
*****
address of s2 object is: 1803294530952
address of self object is: 1803294530952
Hello My Name is: PRASANNA
My Rollno is: 1220
My Marks are: 80
```

Constructor:

- Constructor is a special method in python.
- The name of the constructor should be __init__(self)
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize instance variables.
- Per object constructor will be executed only once.
- Constructor can take atleast one argument(atleast self)
- Constructor is optional and if we are not providing any constructor then python will provide default constructor.

Example:

```
def __init__(self,name,rollno,marks):
    self.name=name
    self.rollno=rollno
    self.marks=marks
```

Program to demonstrate constructor will execute only once per object:

```
class Test:
    def __init__(self):
        print("Constructor exeuction...")
    def m1(self):
        print("Method execution...")
t1=Test()
t2=Test()
t3=Test()
t1.m1()
```

output:

```
Constructor exeuction...
Constructor exeuction...
Constructor exeuction...
Method execution...
```

Differences between Methods and Constructors:

Method	Constructor
1. Name of method can be any name	1. Constructor name should be always __init__
2. Method will be executed if we call that method	2. Constructor will be executed automatically at the time of object creation
3. Per object, method can be called any number of times.	3. Per object, Constructor will be executed only once
4. Inside method we can write business logic	4. Inside Constructor we have to declare and initialize instance variables

Types of Variables:

Inside Python class 3 types of variables are allowed.

1. Instance Variables (Object Level Variables)
2. Static Variables (Class Level Variables)
3. Local variables (Method Level Variables)

1. Instance Variables:

If the value of a variable is varied from object to object, then such type of variables are called instance variables.

For every object a separate copy of instance variables will be created.

Where we can declare Instance variables:

1. Inside Constructor by using self variable
2. Inside Instance Method by using self variable
3. Outside of the class by using object reference variable

Inside Constructor by using self variable:

We can declare instance variables inside a constructor by using self keyword. Once we creates object, automatically these variables will be added to the object.

Example:

```
class Employee:
    def __init__(self):
        self.eno=100
        self.ename='VIVAN'
        self.esal=10000
e=Employee()
print(e._dict_)
```

output:

```
{'eno': 100, 'ename': 'VIVAN', 'esal': 10000}
```

Inside Instance Method by using self variable:

We can also declare instance variables inside instance method by using self variable. If any instance variable declared inside instance method, that instance variable will be added once we call that method.

Example:

```
class Employee:
    def __init__(self):
        self.eno=100
        self.ename='prasanna'
    def method1(self):
        self.esal=10000
e=Employee()
print(e.__dict__)
e.method1()
print(e.__dict__)
```

output:

```
{'eno': 100, 'ename': 'prasanna'}
{'eno': 100, 'ename': 'Prasanna', 'esal': 10000}
```

Outside of the class by using object reference variable:

We can also add instance variables outside of a class to a particular object.

Example:

```
class Employee:
    def __init__(self):
        self.eno=100
        self.ename='Prasanna'
    def method1(self):
        self.esal=10000
e=Employee()
e.method1()
e.age=28
print(e.__dict__)
```

output:

```
{'eno': 100, 'ename': 'Prasanna', 'esal': 10000, 'age': 28}
```

How to access Instance variables:

We can access instance variables **within the class by using self variable** and **outside of the class by using object reference**.

Example:

```
class Employee:
    def __init__(self):
        self.eno=100
        self.ename='Prasanna'
    def display(self):
        print(self.eno)
        print(self.ename)
e=Employee()
e.display()
print(e.eno,e.ename)
```

output:

```
100
Prasanna
100
Prasanna
```

How to delete instance variable from the object:

1. **Within a class** we can delete instance variable as follows

```
del self.variableName
```

2. **From outside** of class we can delete instance variables as follows


```
del objectreference.variableName
```

Example:

```
class Employee:
    def __init__(self):
        self.eno=100
        self.ename='Prasanna'
        self.eage=28
        self.esal=30000
    def display(self):
        del self.eage
e=Employee()
print(e.__dict__)
e.display()
print(e.__dict__)
del e.ename
print(e.__dict__)
```

output:

```
{'eno': 100, 'ename': 'Prasanna', 'eage': 28, 'esal': 30000}
{'eno': 100, 'ename': 'Prasanna', 'esal': 30000}
{'eno': 100, 'esal': 30000}
```

Note: The instance variables which are deleted from one object, will not be deleted from other objects.

```
class Employee:
    def __init__(self):
        self.eno=100
        self.ename='Prasanna'
        self.eage=28
        self.esal=30000
e1=Employee()
e2=Employee()
del e1.ename
print(e1.__dict__)
print(e2.__dict__)
```

output:

```
{'eno': 100, 'eage': 28, 'esal': 30000}
{'eno': 100, 'ename': 'Prasanna', 'eage': 28, 'esal': 30000}
```

If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.

2. Static variables:

- If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such type of variables are called Static variables.
- For total class only one copy of static variable will be created and shared by all objects of that class.
- We can access static variables either by class name or by object reference. But recommended to use class name.

Example:

```
class Employee:
    esal=30000
    def __init__(self):
        self.eno=100
        self.ename='Prasanna'
e1=Employee()
e2=Employee()
print("e1:",e1.eno,e1.ename,e1.esal)
print("e2:",e2.eno,e2.ename,e2.esal)
Employee.esal=40000
```

```
e1.ename="Shankar"
print("e1:",e1.eno,e1.ename,e1.esal)
print("e2:",e2.eno,e2.ename,e2.esal)
```

output:

```
e1: 100 Prasanna 30000
e2: 100 Prasanna 30000
e1: 100 Shankar 40000
e2: 100 Prasanna 40000
```

Various places to declare static variables:

1. In general we can declare static variable within the class directly but declare outside of any method
2. Inside constructor by using **class name**
3. Inside instance method by using **class name**
4. Inside class_method by using either **class name or cls variable**
5. Inside static method by using **class name**

Example:

```
class Employee:
    esal=30000
    def __init__(self): #constructor
        Employee.eno=100
    def m1(self): #instance method
        Employee.ename='Prasanna'
    def m2(cls): # class method
        cls.elocation="B N REDDY"
        Employee.eage=28
    def m3(): #static method
        Employee.edist="RR"
print(Employee.__dict__)
```

How to access static variables:

1. inside constructor: by using either **self or classname**
2. inside instance method: by using either **self or classname**
3. inside class method: by using either **cls variable or classname**
4. inside static method: by using **classname**
5. From outside of class: by using either **object reference or classnae**

Where we can modify the value of static variable:

Anywhere either **with in the class or outside of class** we can modify by using **classname**.
But **inside class method**, by using **cls variable**.

How to delete static variables of a class:

We can delete static variables from anywhere by using the following syntax

```
del classname.variablename
```

But inside classmethod we can also use cls variable

```
del cls.variablename
```

Note:

- By using object reference variable/self we can read static variables, but we cannot modify or delete.
- If we are trying to modify, then a new instance variable will be added to that particular object.
- If we are trying to delete then we will get error.
- We can modify or delete static variables only by using classname or cls variable.

Instance Variable vs Static Variable:

In the case of instance variables for every object a separate copy will be created, but in the case of static variables for total class only one copy will be created and shared by every object of that class.

3. Local variables:

- Sometimes to meet temporary requirements of programmer, we can **declare variables inside a method directly**, such type of variables are called local variable or temporary variables.
- Local variables will be created at the time of method execution and destroyed once method completes.
- Local variables of a method **cannot be accessed from outside of method**.

Example1:

```
class Test:
    def m1(self):
        a=1000
        print("m1 method local variable value is: ",a)
    def m2(self):
        b=2000
        print("m2 method local variable value is: ",b)
t=Test()
t.m1()
t.m2()
```

output:

```
m1 method local variable value is: 1000
m2 method local variable value is: 2000
```

Example2:

```
class Test:
    def m1(self):
        a=1000
        print("m1 method local variable value is: ",a)
    def m2(self):
        b=2000
        print("m1 method local variable value is: ",a)... #NameError: name 'a' is not defined
        print("m2 method local variable value is: ",b)
t=Test()
t.m1()
t.m2()
```

Types of Methods:

Inside Python class 3 types of methods are allowed

1. Instance Methods
2. Class Methods
3. Static Methods

1. Instance Methods:

- Inside method implementation if we are using instance variables then such type of methods are called instance methods.
- Inside instance method declaration, we have to pass self variable.
def m1(self):
- By using self variable inside method we can able to access instance variables.
- Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

```
class Student:
    def __init__(self,name,marks):
        self.name=name
        self.marks=marks
    def display(self):
        print('Hi',self.name)
        print('Your Marks are:',self.marks)
    def grade(self):
```

```

        if self.marks>=60:
            print('You got First Grade')
        elif self.marks>=50:
            print('You got Second Grade')
        elif self.marks>=35:
            print('You got Third Grade')
        else:
            print('You are Failed')
n=int(input('Enter number of students:'))
for i in range(n):
    name=input('Enter Name:')
    marks=int(input('Enter Marks:'))
    s= Student(name,marks)
    s.display()
    s.grade()
    print("*****10)

```

OUTPUT:

```

Enter number of students:4
Enter Name:SHANKAR
Enter Marks:70
Hi SHANKAR
Your Marks are: 70
You got First Grade
*****
Enter Name:DILEEP KUMAR
Enter Marks:55
Hi DILEEP KUMAR
Your Marks are: 55
You got Second Grade
*****
Enter Name:AMAR
Enter Marks:35
Hi AMAR
Your Marks are: 35
You got Third Grade
*****
Enter
Name:PRASANNA
Enter Marks:25
Hi PRASANNA
Your Marks are: 25
You are Failed
*****

```

Setter and Getter Methods:

We can set and get the values of instance variables by using getter and setter methods.

Setter Method:

setter methods can be used to set values to the instance variables. setter methods also known as mutator methods.

syntax:

```

def setVariable(self,variable):
    self.variable=variable

```

Example:

```

def setName(self,name):
    self.name=name

```

Getter Method:

Getter methods can be used to get values of the instance variables. Getter methods also known as accessor methods.

syntax:

```

def getVariable(self):
    return self.variable

```

Example:

```
def getName(self):
    return self.name
```

Example:

```
class Student:
    def setName(self,name):
        self.name=name
    def getName(self):
        return self.name
    def setMarks(self,marks):
        self.marks=marks
    def getMarks(self):
        return self.marks
n=int(input('Enter number of students:'))
for i in range(n):
    s=Student()
    name=input('Enter Name:')
    s.setName(name)
    marks=int(input('Enter Marks:'))
    s.setMarks(marks)
    print('Hi',s.getName())
    print('Your Marks are:',s.getMarks())
    print("*****10)
```

Output:

```
Enter number of students:2
Enter Name:PRASANNA
GOUDEnter Marks:70
Hi PRASANNA GOUD
Your Marks are: 70
*****
Enter Name:vamsi krishna
Enter Marks:85
Hi vamsi krishna
Your Marks are: 85
*****
```

Class Methods:

- Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.
- We can declare class method explicitly by using @classmethod decorator.
- For class method we should provide cls variable at the time of declaration
- We can call classmethod by using classname or object reference variable.

Example:

```
class Animal:
    legs=4
    @classmethod
    def walk(cls,name):
        #name is local variable and legs are static variable
        print('{} walks with {} legs '.format(name,cls.legs))
Animal.walk('Dog')
Animal.walk('Cat')
```

Output:

```
Dog walks with 4 legs
Cat walks with 4 legs
```

Instance method Vs class method:

<u>Instance method</u>	<u>Class method</u>
Inside method body if we are using atleast one instance variable then compulsory we should declare that method as instance method	Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method
To declare instance method we are not required to use any decorator	To declare class method compulsory we should use @classmethod decorator
The first argument to the instance method should be self, which is reference to current object and by using self, we can access instance variables inside method	The first argument to the class method should be cls, which is reference to current class object and by using that we can access static variables inside method
Inside instance method we can access both instance and static variables	Inside class method we can access only static variables and we cannot access instance variables
we can call instance method by using object reference variable	We can call classmethod by using classname or object reference variable

Static Methods:

- In general static methods are general utility methods.
- Inside these methods we won't use any instance or class variables.
- Here we won't provide self or cls arguments at the time of declaration.
- We can declare static method explicitly by using **@staticmethod** decorator
- We can access static methods by using classname or object reference variable

Example:

```
class Clac:
    @staticmethod
    def add(x,y):
        print('The Sum:',x+y)
    @staticmethod
    def product(x,y):
        print('The Product:',x*y)
    @staticmethod
    def average(x,y):
        print('The average:',(x+y)/2)
Calc.add(10,20)
Calc.product(10,20)
Calc.average(10,20)
```

Output:

```
The Sum: 30
The Product: 200
The average: 15.0
```

Note: In general we can use only instance and static methods. Inside static method we can access class level variables by using class name.

passing members of one class to another class:

```
class Employee:
    def __init__(self,eno,ename,esal):
        self.eno=eno
        self.ename=ename
        self.esal=esal
    def display(self):
        print('Employee Number:',self.eno)
        print('Employee Name:',self.ename)
        print('Employee Salary:',self.esal)
class Test:
    def modify(emp):
        emp.esal=emp.esal+10000
        emp.display()
e=Employee(112,'PRASANNA',10000)
Test.modify(e)
```

output:Employee Number: 112
Employee Name:
PRASANNAEmployee
Salary: 20000

Inner classes:

- Sometimes we can declare a class inside another class, such type of classes are called inner classes.
- Without existing one type of object if there is no chance of existing another type of object, then we should go for inner classes.

Example: Without existing Car object there is no chance of existing Engine object. Hence Engine class should be part of Car class.

```
class Car:  
.....  
    class Engine:  
        .....
```

Note: Without existing outer class object there is no chance of existing inner class object. Hence inner class object is always associated with outer class object.

```
class Outer:  
    def __init__(self):  
        print("outer class object creation")  
    class Inner:  
        def __init__(self):  
            print("inner class object creation")  
        def m1(self):  
            print("inner class method")  
o=Outer()  
i=o.Inner()  
i.m1()
```

output:

```
outer class object creation  
inner class object creation  
inner class method
```

Data Hiding

- Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type.
- By default all members of a class can be accessed outside of class.
- You can prevent this by making class members private or protected.
- In Python, we use double underscore (__) before the attributes name to make those attributes private.
- We can use single underscore (_) before the attributes name to make those attributes protected.\

Example:

```
class MyClass:  
    __hiddenVar = 0    # Private member of MyClass  
    _protectedVar = 0  # Protected member of MyClass  
    # A member method that changes __hiddenVariable  
    def add(self, increment):  
        self.__hiddenVar += increment  
        print (self.__hiddenVar)  
m = MyClass()  
m.add(4)  
m.add(6)  
# This will causes error  
print (MyClass.__hiddenVariable)  
print (MyClass._protectedVar)
```

output:

```
4  
10  
AttributeError: type object 'MyClass' has no attribute '__hiddenVar'
```

In the above program, we tried to access hidden variable outside the class using object and it threw an exception.

We can access the value of hidden attribute by a below syntax:

“objectName._Classname_hiddenVariable”

Now see the above program to display the hidden value

```
class MyClass:
    # Hidden member of MyClass
    __a = 0
    # A member method that changes __hiddenVariable
    def add(self, x):
        self.__a+=x
        print (self.__a)
Ob = MyClass()
Ob.add(2)
Ob.add(5)
print (Ob._MyClass_a)
```

output:

```
2
7
7
```


Inheritance

The process by which one class acquires the properties and functionalities of another class is called inheritance. Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class.

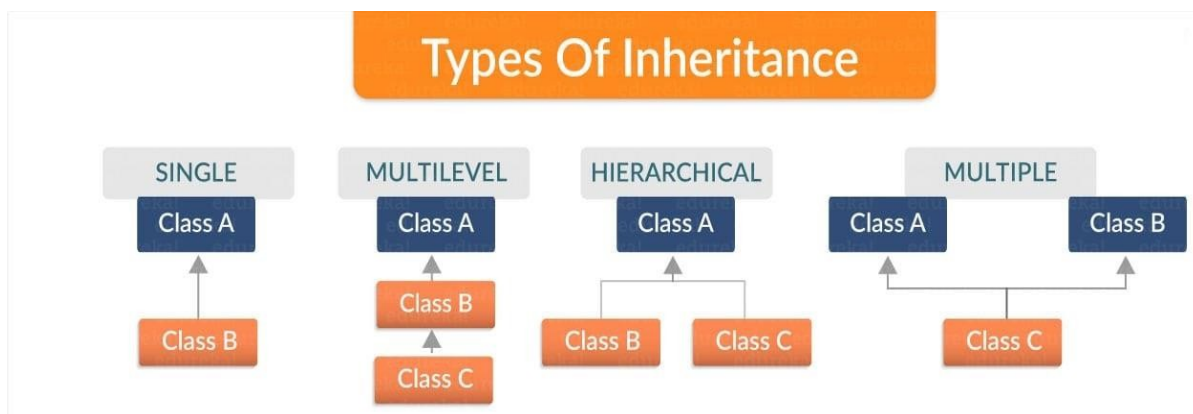


As we can see in the image, a child inherits the properties from the father. Similarly, in python, there are two classes:

1. Parent class (Super or Base class)
2. Child class (Subclass or Derived class)

A class which inherits the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.

Inheritance refers to the ability to create Sub-classes that contain specializations of their parents. It is further divided into four types namely single, multilevel, hierarchical and multiple inheritances. Refer the below image to get a better understanding.



In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

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

Syntax

```
class base_class:
    base_class defination
class derived_class(<base class>):
    derived class definition
```

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Syntax

```
class derive_class(<base class 1>, <base class 2>,.....<base class n>):
    derive_class definition
```

Example 1

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

```
dog barking
Animal Speaking
```

2. Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

The syntax of multi-level inheritance is given below.

Syntax

```
class class1:
    class1 defination
    .....
class class2(class1):
    class2 defination
    .....
class class3(class2):
    class3 defination
    ..... .
```

Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

3. Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class. The syntax to perform multiple inheritance is given below.

Syntax

```
class Base1:
    Base1 defination
class Base2:
    Base2 defination
```

```

class BaseN:
    Base N definition
class Derived(Base1, Base2,..... BaseN):
    derived class definition

```

Example

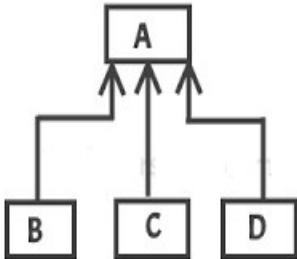
```

class Calc1:
    def Sum(self,a,b):
        return a+b;
class Calc2:
    def Mul(self,a,b):
        return a*b;
class Derived(Calc1,Calc2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Sum(50,60) ..... 110
print(d.Mul(60,50)).....3000
print(d.Divide(100,20)) ..... 5.0

```

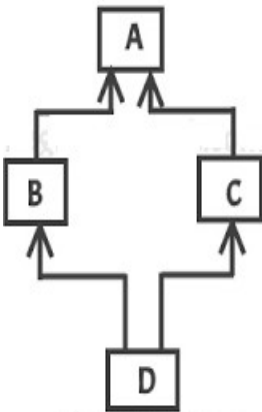
4. Hierarchical Inheritance

Hierarchical inheritance involves multiple inheritances from the same base or parent class.

Flow Chart	Syntax	Example
 <p>Hierarchical Inheritance</p>	<pre> class A: definition of A class B(A): definition of B class C(A): definition of C class D(A): definition of D </pre>	<pre> class Parent: def func1(self): print("this is function 1") class Child1(Parent): def func2(self): print("this is function 2") class Child2(Parent): def func3(self): print("this is function 3") ob = Child1() ob1 = Child2() ob.func1() ob.func2() ob1.func1() ob1.func3() </pre>

5. Hybrid Inheritance

Hybrid inheritance involves multiple inheritance (combination of any two inheritances) taking place in a single program.

Flow Chart	Syntax	Example
 <p>Hybrid Inheritance</p>	<pre> class A: definition of A class B(A): definition of B class C(A): definition of C class D(B,C) definattion of D </pre>	<pre> class A: def func1(self): print("this is function one") class B(A): def func2(self): print("this is function 2") class C(A): def func3(self): print(" this is function 3"): class D(B,C): def func4(self): print(" this is function 4") ob = D() ob.func1() ob.func2() </pre>

The issubclass(sub,sup) method:

The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

Example

```
class Calc1:
    def Sum(self,a,b):
        return a+b;
class Calc2:
    def Mul(self,a,b):
        return a*b;
class Derived(Calc1,Calc2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(issubclass(Derived,Calc2))
print(issubclass(Calc1,Calc2))
```

Output:

```
True
False
```

The isinstance(obj, class) method:

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Example

```
class Calc1:
    def Sum(self,a,b):
        return a+b;
class Calc2:
    def Mul(self,a,b):
        return a*b;
class Derived(Calc1,Calc2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(isinstance(d,Derived))
print(isinstance(d,Calc1))
print(isinstance(d,Calc2))
```

Output:

```
True
True
True
```

Method overriding:

What ever members available in the parent class are by default available to the child class through inheritance. If the child class not satisfied with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding. Overriding concept applicable for both methods and constructors

Example:

```
class Bank:
    def getroi(self):
        return 10
class SBI(Bank):
    def getroi(self):
        return 7
class ICICI(Bank):
    def getroi(self):
        return 8
```

```

b1 = Bank()
b2 = SBI()
b3 = ICICI()
print("Bank Rate of interest:",b1.getroi())
print("SBI Rate of interest:",b2.getroi())
print("ICICI Rate of interest:",b3.getroi())

```

output:

```

Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8

```

Constructor overriding:

```

class P:
    def __init__(self):
        print('Parent Constructor')
class C(P):
    def __init__(self):
        print('Child Constructor')
c=C()

```

output: Child Constructor

In the above example, if child class does not contain constructor then parent class constructor will be executed. From child class constructor we can call parent class constructor by using **super() method**

#Program to call Parent class constructor by using super():

```

class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
class Employee(Person):
    def __init__(self,name,age,eno,esal):
        super().__init__(name,age)
        self.eno=eno
        self.esal=esal
    def display(self):
        print('Employee Name:',self.name)
        print('Employee Age:',self.age)
        print('Employee Number:',self.eno)
        print('Employee Salary:',self.esal)
e1=Employee('Prasanna',28,1001,25000)
e1.display()
print("*****20")
e2=Employee('shankar',28,1002,26000)
e2.display()

```

output:

```

Employee Name:
PrasannaEmployee Age:
28
Employee Number: 1001
Employee Salary: 25000
*****
Employee Name: shankar
Employee Age: 28
Employee Number: 1002
Employee Salary: 26000

```

Exception Handling

What is an error: Errors are the conditions that cannot be handled and irrecoverable. Every time when Error occurs, the program terminates suddenly and unexpectedly.

In any programming language there are 2 types of errors are possible.

1. Syntax Errors
2. Runtime Errors

1. Syntax Errors: The errors which occurs because of invalid syntax are called syntax errors.

Eg 1:

```
>>> if a < 3
      File "<interactive input>", line 1
      if a < 3
          ^
```

SyntaxError: invalid syntax i.e We can notice here that a colon(:) is missing in the if statement.

Eg 2:

```
print "Hello"
```

SyntaxError: Missing parentheses in call to 'print'

Note: Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

2. Runtime Errors: Also known as exceptions. While executing the program if something goes wrong because of end user input or programming logic or memory problems etc.. then we will get Runtime Errors.

Eg1:

```
print(10/0) ==> ZeroDivisionError: division by zero
print(10/"ten") ==> TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Eg2:

```
x=int(input("Enter Number:"))
print(x)
```

output: Enter Number:ten

ValueError: invalid literal for int() with base 10: 'ten'

Exception Handling concept applicable for Runtime Errors but not for syntax errors.

What is an Exception: An unwanted and unexpected event that disturbs normal flow of program is called exception.

- Whenever an exception occurs, the program halts the execution, and thus the further code is not executed. Therefore, an exception is the error which python script is unable to tackle with.
- The main objective of exception handling is Graceful Termination of the program(i.e we should not block our resources and we should not miss anything).
- Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally.

Error /Syntax errors	Exceptions /Runtime errors
<p>1. Errors are the conditions that cannot be handled and irrecoverable. Every time when Error occurs, the program terminates suddenly and unexpectedly.</p> <p>2.It is also called as syntax errors i.e The errors which occurs because of invalid syntax are called syntax errors.</p> <p>3. Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.</p> <p>4. following are the example for syntax errors</p> <ul style="list-style-type: none">➤ leaving out a keyword,➤ putting a keyword in the wrong place,➤ leaving out a symbol, such as a colon, comma or brackets,➤ misspelling a keyword➤ incorrect indentation	<p>1. An unwanted and unexpected event that disturbs normal flow of program is called exception. Whenever an exception occurs, the program halts the execution, and thus the further code is not executed</p> <p>2. It is also called as runtime errors i.e While executing the program if something goes wrong because of end user input or programming logic or memory problems etc.. then we will get Runtime Errors.</p> <p>3. exceptions are handled by using try, except and finally blocks</p> <p>4. following are the example for exceptions/runtime errors</p> <ul style="list-style-type: none">➤ division by zero➤ performing an operation on incompatible types➤ using an identifier which has not been defined➤ accessing a list element, dictionary value or object attribute which doesn't exist➤ trying to access a file which doesn't exist

Types of Exceptions:

In Python there are 2 types of exceptions are possible.

1. Predefined Exceptions
2. User Defined Exceptions

1. Predefined Exceptions:

Also known as in-built exceptions. The exceptions which are raised automatically by Python virtual machine whenever a particular event occurs, are called pre defined exceptions.

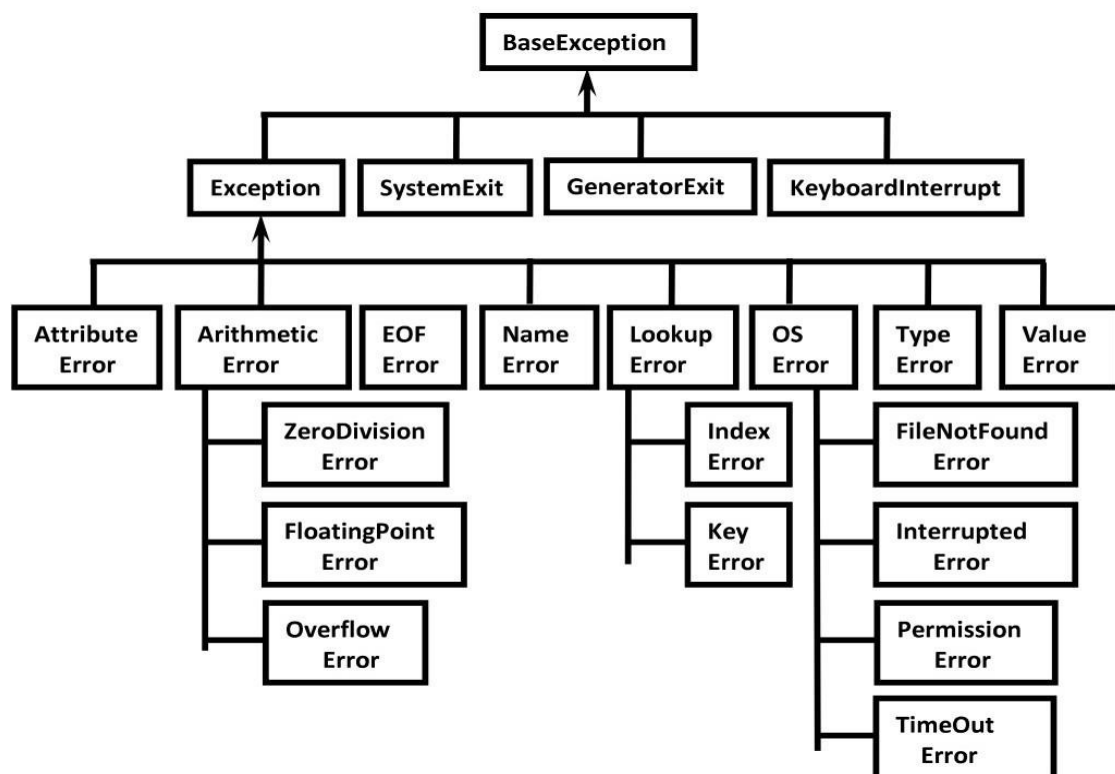
Examples:

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

2. User Defined Exceptions:

- Also known as Customized Exceptions or Programatic Exceptions
- Some time we have to define and raise exceptions explicitly to indicate that something goes wrong ,such type of exceptions are called User Defined Exceptions or Customized Exceptions
- Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "raise" keyword.

Python's Exception Hierarchy



- Every Exception in Python is a class.
- All exception classes are child classes of BaseException.i.e every exception class extends BaseException either directly or indirectly. Hence BaseException acts as root for Python Exception Hierarchy.

Exception handling in python:

In python we can handle the exceptions by using following key words

- 1) try 2) except 3)finally

If the program contains **suspicious code or risky code** that may throw the exception, we must place that code in the try block. The try block must be followed with the except statement which contains a block of code that will be executed if there is some exception in the try block.

<div>try</div> <div>Run this code</div> <div>except</div> <div>Run this code if an exception occurs</div>	Syntax: try: Risky Code except XXX: Handlingcode/Alternative Code Other code...
---	---

<u>without try-except:</u> <pre>print("hello") print(10/0) print("haii")</pre> Output hello ZeroDivisionError: division by zero Abnormal termination/Non-Graceful Termination	<u>with try-except:</u> <pre>print("hello") try: print(10/0) except ZeroDivisionError: print(10/2) print("haii")</pre> Output hello 5.0 haii Normal termination/Graceful Termination
--	--

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

Syntax: try: #block of code except Exception1: #block of code else: #this code executes if no except block is executed	Example: try: a = int(input("Enter a:")) b = int(input("Enter b:")) c = a/b; print("a/b = %d"%c) except Exception: print("can't divide by zero") else: print("Hi I am else block") Output: Enter a:10 Enter b:2 a/b = 5 Hi I am else block
---	--

try with multiple except blocks:

The way of handling exception is varied from exception to exception. Hence for every exception type a separate except block we have to provide. i.e try with multiple except blocks is possible and recommended to use.

Syntax:

Eg:

```
try:
    stmt1
    stmt2
    stmt3
except exception1:
    stmt4
except exception2:
    stmt5
```

If try with multiple except blocks available then based on raised exception the corresponding except block will be executed.

Example:

```
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ZeroDivisionError :
    print("Can't Divide with Zero")
except ValueError:
    print("please provide int value only")
```

Output:

```
Enter First Number: 100
Enter Second Number: 0
Can't Divide with Zero
```

```
Enter First Number: 20
Enter Second Number: abc
please provide int value only
```

If try with multiple except blocks available then the order of these except blocks is important. Python interpreter will always consider from top to bottom until matched except block identified.

Example:

```
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ArithmeticError :
    print("ArithmeticError")
except ZeroDivisionError :
    print("ZeroDivisionError")
```

Output:

```
Enter First Number: 25
Enter Second Number: 0
ArithmeticError
```

Single except block that can handle multiple exceptions:

- We can write a single except block that can handle multiple different types of exceptions.
except (Exception1,Exception2,exception3,...): or
except (Exception1,Exception2,exception3,...) as msg :
- Parenthesis are mandatory and this group of exceptions internally considered as tuple.

Example:

```
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except (ArithmeticError,ValueError) as msg:
    print("Plz Provide valid numbers only and problem is: ",msg)
```

Output:

```
Enter First Number: 25
Enter Second Number: 0
Plz Provide valid numbers only and problem is: division by zero
```

```
Enter First Number: 20
Enter Second Number: ten
Plz Provide valid numbers only and problem is: invalid literal for int() with base 10: 'ten'
```

Default except block:

We can use default except block to handle any type of exceptions. In default except block generally we can print normal error messages.

Syntax:

```
except:
    statements
```

Example:

```
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ArithmeticError:
    print("Can't Divide with Zero")
except:
    print("Plz Provide valid numbers only")
```

Output:

```
Enter First Number: 20
Enter Second Number: 0
Can't Divide with Zero
```

```
Enter First Number: 20
Enter Second Number: ten
Plz Provide valid numbers only
```

finally block

The speciality of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled.

The syntax to use the finally block is given below.

```
try:
    Risky Code
except:
    Handling Code
finally:
    Cleanup code/compulsory executed code
```

How finally blocks works in different cases

Case-1: If there is no exception try: print("hello this is try block") except: print("hello this is except block") finally: print("hello this is finally block") output: hello this is try block hello this is finally block	Case-2: If there is an exception raised but handled try: print("hello this is try block") print(10/0) except ZeroDivisionError: print("hello this is except block") finally: print("hello this is finally block") output: hello this is try block hello this is except block hello this is finally block	Case-3: If there is an exception raised but not handled try: print("hello this is try block") print(10/0) except ValueError: print("hello this is except block") finally: print("hello this is finally block") output: hello this is try block hello this is finally block ZeroDivisionError: division by zero
---	---	---

Control Flow in try-except and try-except-finally

<u>Control Flow in try-except:</u> try: stmt-1 stmt-2 stmt-3 except XXX: stmt-4 stmt-5	<u>Control flow in try-except-finally</u> try: stmt-1 stmt-2 stmt-3 except: stmt-4 finally: stmt-5 stmt6
case-1: If there is no exception 1,2,3,5 and Normal Termination case-2: If an exception raised at stmt-2 and corresponding except block matched 1,4,5 Normal Termination case-3: If an exception raised at stmt-2 and corresponding except block not matched 1, Abnormal Termination case-4: If an exception raised at stmt-4 or at stmt-5 then it is always abnormal termination.	Case-1: If there is no exception 1,2,3,5,6 Normal Termination Case-2: If an exception raised at stmt2 and the corresponding except block matched 1,4,5,6 Normal Termination Case-3: If an exception raised at stmt2 but the corresponding except block not matched 1,5 Abnormal Termination Case-4: If an exception raised at stmt4 it is always abnormal termination but before that finally block will be executed. Case-5: If an exception raised at stmt-5 or at stmt-6 always abnormal termination

Nested try-except-finally blocks:

We can take try-except-finally blocks inside try or except or finally blocks.i.e nesting of tryexcept-finally is possible.

Syntax:

```
try:
    stmt-1
    stmt-2
    try:
        stmt-3
        stmt-4
    except:
        stmt-5
except:
    stmt-6
stmt-7
stmt-8
```

General Risky code we have to take inside outer try block and too much risky code we have to take inside inner try block. Inside Inner try block if an exception raised then inner except block is responsible to handle. If it is unable to handle then outer except block is responsible to handle.

Example:

```
try:
    print("outer try block")
    try:
        print("Inner try block")
        print(10/0)
    except ZeroDivisionError:
        print("Inner except block")
    finally:
        print("Inner finally block")
except:
    print("outer except block")
finally:
    print("outer finally block")
```

Output:

```
outer try block
Inner try block
Inner except block
Inner finally block
outer finally block
```

Raising exceptions

An exception can be raised by using the raise clause in python. The syntax to use the raise statement is given below.

Syntax:

raise Exception_class,<value>

1. To raise an exception, raise statement is used. The exception class name follows it.
2. An exception can be provided with a value that can be given in the parenthesis.
3. To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception

Example1:

```
try:
    age = int(input("Enter the age?"))
    if age<25:
        raise ValueError;
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

Output:

```
Enter the age?28
the age is valid
*****
Enter the age?16
The age is not valid
```

Example2:

```
try:
    a = int(input("Enter a?"))
    b = int(input("Enter b?"))
    if b is 0:
        raise ArithmeticError;
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

Output:

```
Enter a?20
Enter b?4
a/b = 5.0
*****
Enter a?20
Enter b?0
The value of b can't be 0
```

Custom Exception

The python allows us to create our exceptions that can be raised from the program and caught using the except clause.

Every exception in Python is a class that extends Exception class either directly or indirectly.

Syntax:

```
class classname(predefined exception class name):
```

```
    def __init__(self,arg):  
        self.msg=arg
```

Eg:

```
class TooYoungException(Exception):
```

```
    def __init__(self,arg):  
        self.msg=arg
```

TooYoungException is our class name which is the child class of Exception

Example:

```
class TooYoungException(Exception):
    def __init__(self,arg):
        self.msg=arg
class TooOldException(Exception):
    def __init__(self,arg):
        self.msg=arg
age=int(input("Enter Age:"))
if age>21 and age<=40:
    raise TooYoungException("You will get match details soon by email!!!")
elif age>40:
    raise TooOldException("Your age already crossed marriage age...no chance of getting marriage")
else:
    print("your age is below 21 plz contact after crossing 21!!!")
```

Output:

Enter Age:18
your age is below 21 plz contact after crossing 21!!!

Enter Age:25
TooYoungException: You will get match details soon by email!!!

Enter Age:45
TooOldException: Your age already crossed marriage age...no chance of getting marriage