**PYTHON**

**CONTENT:**

**Data Types:**

* LIST
* TUPLE
* DICTIONARY
* SET
* STRING
* INTEGER

**Conditional statement:**

* IF-ELIF-ELSE

**Loop:**

* FOR AND FOR-ELSE
* WHILE AND WHILE-ELSE

**Function:**

* FUNCTION
* ITERATOR, GENERATOR
* MAP, REDUCE, FILTER

**oops:**

* CLASS
* OBJECT
* CONSTRUCTOR
* INHERITANCE
* PRIVATE, PUBLIC AND PROTECTED VARIABLE
* ABSTRACTION
* ENCAPSULATIONS
* POLYMORPHISM
* PACKAGES
* MODULES

**Exception handling:**

* INTRODUCTION TO EXCEPTION HANDLING
* EXCEPTION HANDLING KEYWORDS
* EXCEPTION HANDLING SYNTAX
* HANDLING MULTIPLE EXCEPTIONS
* HANDLING ALL EXCEPTIONS
* USING EXCEPTION OBJECT
* GETTING DETAILS OF EXCEPTION
* RAISING AN EXCEPTION
* USING FINALLY BLOCK
* CREATING USER DEFINED EXCEPTIONS

**Logging:**

**OOPs(Object Oriented Programming)**

**INTRODUCTION:**

OOPs concepts 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.

**What is class/object in python?**

* Class is nothing but a classifications or segmentation of a real-world entity for

Example: car/human/animal we will get some idea about car but not in details like every segment of car and every type of car. I won’t get a complete or precise understanding but I will get some idea about it.

* Python is an object-oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

* Why we are talking about class in Programming language:

When we are trying to build something in programming language (programming language is used to develop some software, to build a pipe line or may be to automate some processes) we have so many components (we have some components just to connect the sources, to process a file, to automate the process, to integrate API) to develop those components then we will write million line of codes which is again a tedious process.

Just to maintain the entire software or module we need to have some specific boundaries when the next person will come, they can able to go to the particular file/ directories/folder and can change the code easily, so for that we need to write the code using classes and objects.

To write the code in a modular fashion we need to define different classes and have to define the methods for the maintainability of the code.

That is the reason object-oriented programming comes into a picture.

To create a class, use the keyword class:

class MyClass:  
  x = 5

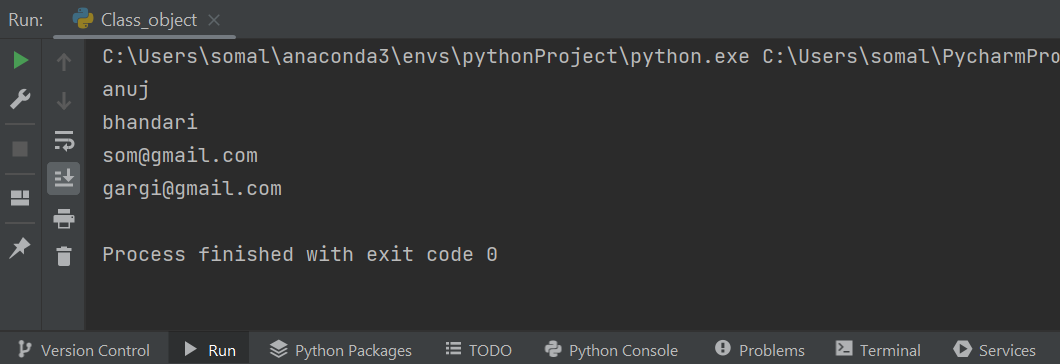
**What are the four pillars of OOPS in Python?**

* Encapsulation
* Inheritance
* Simple Inheritance
* Multiple Inheritance
* Multilevel Inheritance
* Polymorphism
* Method overloading
* Method overriding
* Abstraction

EXAMPLE:

class Person:  
 def \_\_init\_\_(self, name, surname, emailid, year\_of\_birth):  
 self.name = name  
 self.surname = surname  
 self.emailid = emailid  
 self.year\_of\_birth = year\_of\_birth  
  
anuj\_var = Person("anuj", "bhandari", "anuj@gmail.com", 1994)  
somali = Person("somali", "mishra", "som@gmail.com", 1995)  
gargi = Person("gargi", "panda", "gargi@gmail.com", 1999)  
print(anuj\_var.name)  
print(anuj\_var.surname)  
print(somali.emailid)  
print(gargi.emailid)

**OUTPUT:**



\_\_init\_\_ = It is a constructor which is used to pass the value to the class Person.

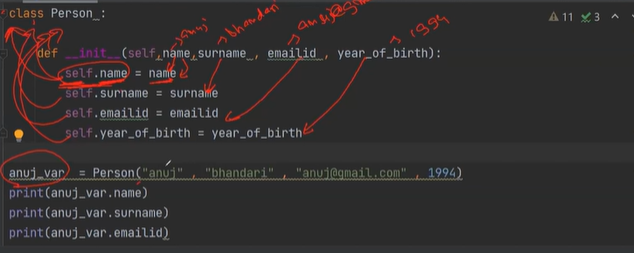
Self = Pointer (which points to the Person Class).

(1st variable inside the constructor is called pointer. Instead of self we can take any other name. self is not a reserve keyword)

name, surname, emailid, year\_of\_birth = Variable (it takes the data)

anuj\_var = Object

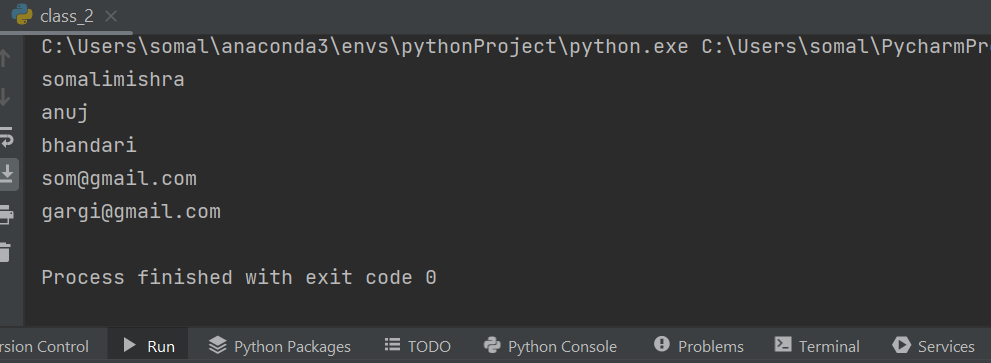
How the flow works:



**Concatenation of name and surname of object somali:**

class Person:  
 def \_\_init\_\_(som, name, surname, emailid, year\_of\_birth):  
 som.name1 = name  
 som.surname = surname  
 som.emailid = emailid  
 som.year\_of\_birth = year\_of\_birth  
  
anuj\_var = Person("anuj", "bhandari", "anuj@gmail.com", 1994)  
somali = Person("somali", "mishra", "som@gmail.com", 1995)  
gargi = Person("gargi", "panda", "gargi@gmail.com", 1999)  
print((somali.name1 + somali.surname))  
print(anuj\_var.name1)  
print(anuj\_var.surname)  
print(somali.emailid)  
print(gargi.emailid)

Output:

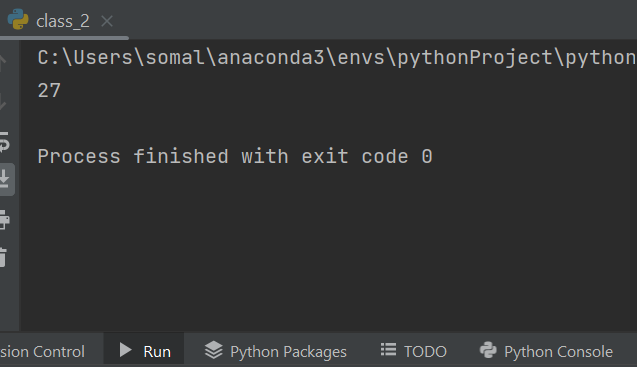


**Create a function to find the age:**

class Person:  
 def \_\_init\_\_(som, name, surname, emailid, year\_of\_birth):  
 som.name1 = name  
 som.surname = surname  
 som.emailid = emailid  
 som.year\_of\_birth = year\_of\_birth

def age(som,current\_year):  
 return current\_year-som.year\_of\_birth  
  
anuj\_var = Person("anuj", "bhandari", "anuj@gmail.com", 1994)  
somali = Person("somali", "mishra", "som@gmail.com", 1995)  
gargi = Person("gargi", "panda", "gargi@gmail.com", 1999)  
  
print(somali.age(2022))

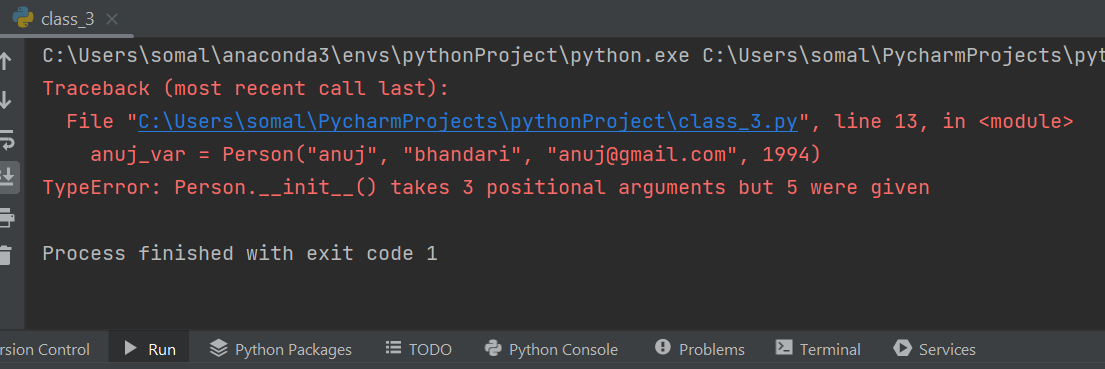
Output:



**When we create multiple \_\_init \_\_ method it is always going to consider the last one:**

class Person:  
 def \_\_init\_\_(som, name, surname, emailid, year\_of\_birth):  
 som.name1 = name  
 som.surname = surname  
 som.emailid = emailid  
 som.year\_of\_birth = year\_of\_birth  
 def \_\_init\_\_(som, name, surname):  
 som.name1 = name  
 som.surname = surname  
 def age(som,current\_year):  
 return current\_year-som.year\_of\_birth  
  
anuj\_var = Person("anuj", "bhandari", "anuj@gmail.com", 1994)  
somali = Person("somali", "mishra", "som@gmail.com", 1995)  
gargi = Person("gargi", "panda", "gargi@gmail.com", 1999)  
  
print(somali.age(2022))

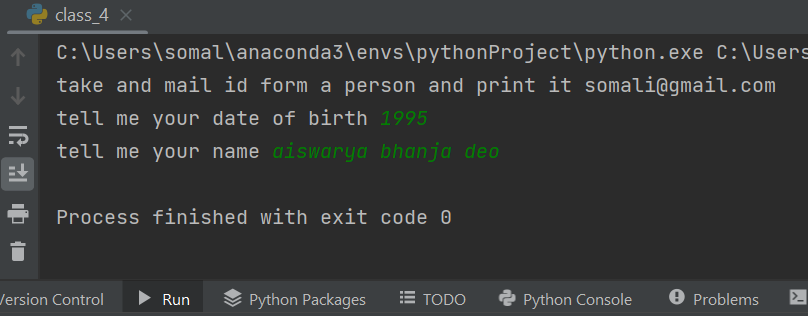
**Output:**



When we create multiple methods (without \_\_init\_\_) and can access all those through the objects:

class person:  
 def age(self,current\_year,year\_of\_birth):  
 return current\_year - self.year\_of\_birth  
  
 def email\_id\_input(self,email\_id):  
 print("take and mail id form a person and print it",email\_id)  
  
 def ask\_name(self):  
 name = input("tell me your name")  
 return name  
 def ask\_dob(self):  
 dob=input("tell me your date of birth")  
 return dob  
  
somali = person()  
aiswarya = person()  
rupali = person()  
abhilash = person()  
  
aiswarya.email\_id\_input("somali@gmail.com")  
aiswarya.ask\_dob()  
aiswarya.ask\_name()

Output:



**Type of Variables in class:**

* Public Variable
* Protected Variable
* Private Variable

The way to declare and call public, protected and private variable:

class person:  
 def \_\_init\_\_(self,name,surname,yob):  
 # \_name1 its a protected variable  
 self.\_name1 = name  
 # \_\_surname1 its a private variable  
 self.\_\_surname1 = surname  
 self.yob1 = yob  
  
som = person("somali",'mishra',1995)  
print(som.\_name1)  
#way of calling private variable  
print(som.\_person\_\_surname1)

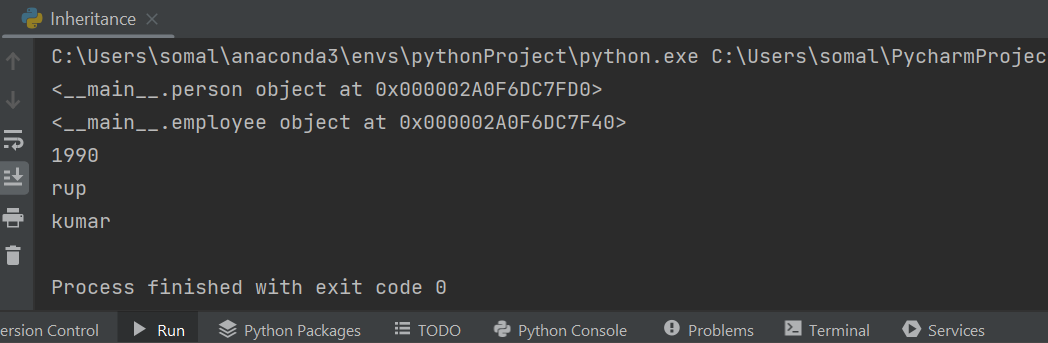
Output:



Understanding public, protected and private with simple inheritance and without using \_\_init\_\_ method:

class person:  
 \_name = "som"  
 \_\_surname = "kumar"  
 yob = 1990  
  
obj=person()  
print(obj)  
  
#inheritance  
class employee(person):  
 \_name = "rup"  
 \_\_surname = "mishra"  
  
obj1=employee()  
print(obj1)  
print(obj1.yob)  
print(obj1.\_name)  
print(obj1.\_person\_\_surname)

Output:



**Note:**

\_name and \_\_surname have been declared in both the parent and child class and \_name is protected variable and \_\_surname is a private variable. While printing the variable in \_name it is giving the value that is declared inside child class but in \_\_surname we can not reassign the value by declaring it inside the child class as it is a private variable so it is giving the value “kumar” (declared inside parent class).

Declaration and calling of private and protected functions and Without \_\_init\_\_ method:

class person:  
 \_name = "som"  
 \_\_surname = "kumar"  
 yob = 1990  
  
 def \_age(self,current\_year):  
 return current\_year-self.yob  
 def \_\_age1(self,current\_year):  
 return current\_year-self.yob  
  
obj=person()  
print(obj.\_age(2022))  
print(obj.\_person\_\_age1(2022))  
  
#inheritance  
class employee(person):  
 \_name = "rup"  
 \_\_surname = "mishra"  
 yob=1991  
  
obj1=employee()  
print(obj1.\_age(2022))  
print(obj1.\_person\_\_age1(2022))  
print(obj1)  
print(obj1.yob)  
print(obj1.\_name)  
print(obj1.\_person\_\_surname)

Output:



**Packages/Modules:**

Packages are nothing but a Directory

Modules are nothing but a file inside that class and object has been defined

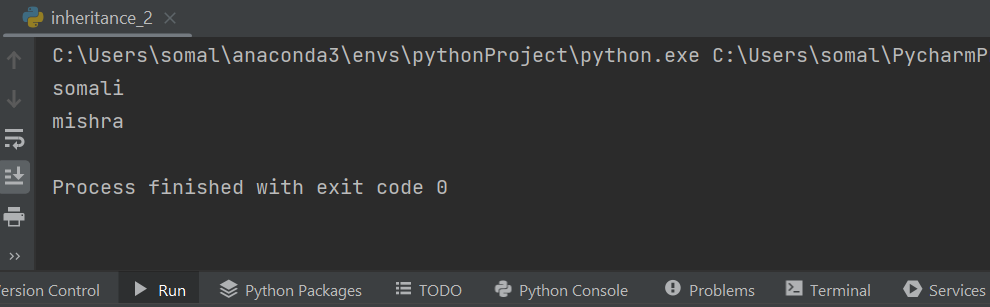
**Example:**

Suppose Inside a directory multiple files are there to call a particular file and class inside the file we can write like this:

**“From directory.file import class\_name”**

class person1:  
 def \_\_init\_\_(self,name,surname,yob):  
 self.\_name1 = name  
 self.\_\_surname1 = surname  
 self.yob1 = yob  
  
som=person1("somali","mishra",1995)  
print(som.\_name1)  
print(som.\_person1\_\_surname1)

**Output:**

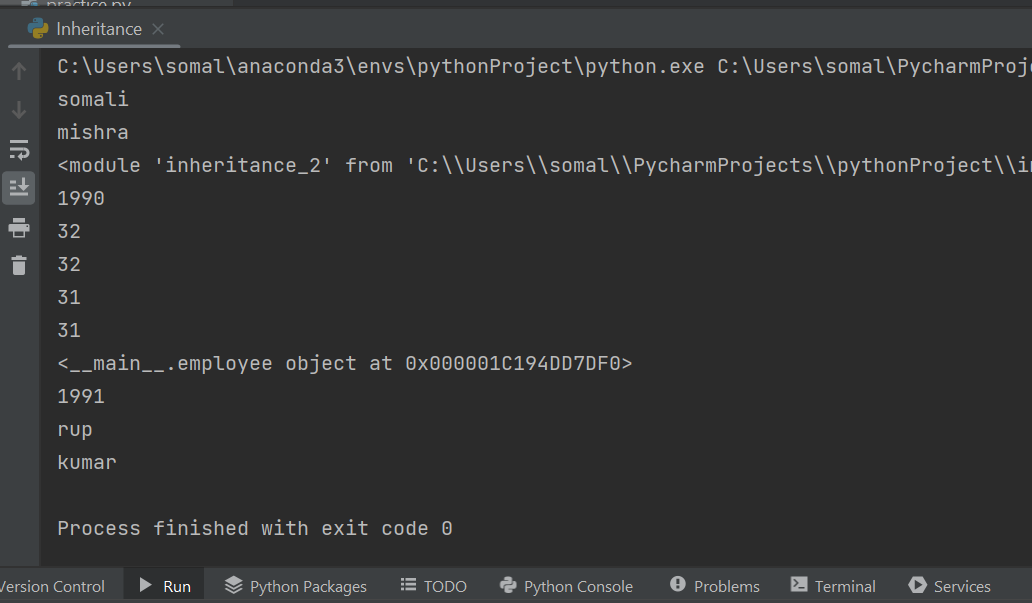
****

**How can we use the code written in one file into another file:**

Here the file inheritance\_2 has been import in to the file inheritance:

import inheritance\_2  
print(inheritance\_2)  
obj3=inheritance\_2.person1("somali","mishra",1990)  
print(obj3.yob1)  
  
class person:  
 \_name = "som"  
 \_\_surname = "kumar"  
 yob = 1990  
  
 def \_age(self,current\_year):  
 return current\_year-self.yob  
 def \_\_age1(self,current\_year):  
 return current\_year-self.yob  
  
obj=person()  
print(obj.\_age(2022))  
print(obj.\_person\_\_age1(2022))  
  
#inheritance  
class employee(person):  
 \_name = "rup"  
 \_\_surname = "mishra"  
 yob=1991  
  
obj1=employee()  
print(obj1.\_age(2022))  
print(obj1.\_person\_\_age1(2022))  
print(obj1)  
print(obj1.yob)  
print(obj1.\_name)  
print(obj1.\_person\_\_surname)

**Output:**



**Notes:**

Here in the above file the previous file ‘inheritance\_2’ has been imported into ‘inheritance’ file. Inside ‘inheritance\_2’ ‘class person1’ has been declared so all the properties of ‘class person1’ can be called inside the current file ‘inheritance’. So we no need to declare the same thing again and again.

When there is multiple classes has been defined inside a file and we have to import only one class for our use in that case we can write like this:

**“from inheritance import person”**

**INHERITANCE:**

Inheritance allows us to define a class that inherits all the methods and properties from another class. Parent class is the class being inherited from, also called base class. Child class is the class that inherits from another class, also called derived class.

**Simple Inheritance:**

#simple Inheritance  
class car: #parent class  
 def \_\_init\_\_(self,body,engine,tyre):  
 self.body = body  
 self.engine = engine  
 self.tyre = tyre  
  
 def milage(self):  
 print("milage of this car")  
  
c=car("solid",'v6',"radial")  
print(c)  
  
class tata(car): #child class  
 pass  
  
t=tata("solid1",'v8',"radial1")  
print(t)  
print(t.milage())

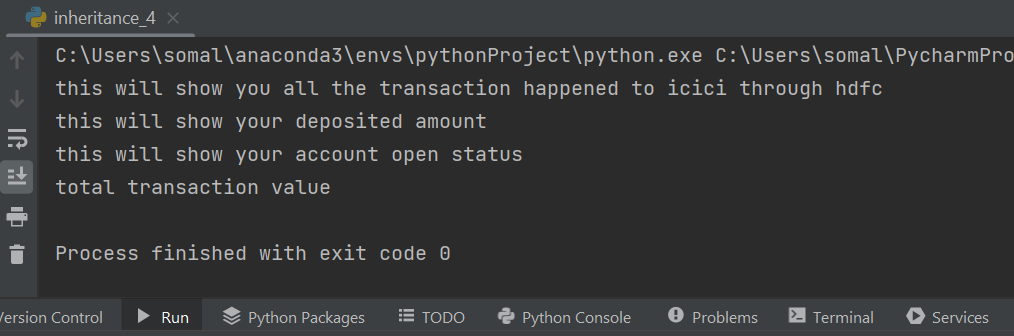
Output:



**Multilevel Inheritance:**

#Multilevel Inheritance  
class bank: #parent class  
 def transaction(self):  
 print("total transaction value")  
 def account\_opening(self):  
 print("this will show your account open status")  
 def deposit(self):  
 print("this will show your deposited amount")  
  
class HDFC\_bank(bank): #child class  
 def hdfc\_to\_icici(self):  
 print("this will show you all the transaction happened to icici through hdfc")  
  
class icici(HDFC\_bank): #child class  
 pass  
  
i=icici()  
i.hdfc\_to\_icici()  
i.deposit()  
i.account\_opening()  
i.transaction()

**Output:**

****

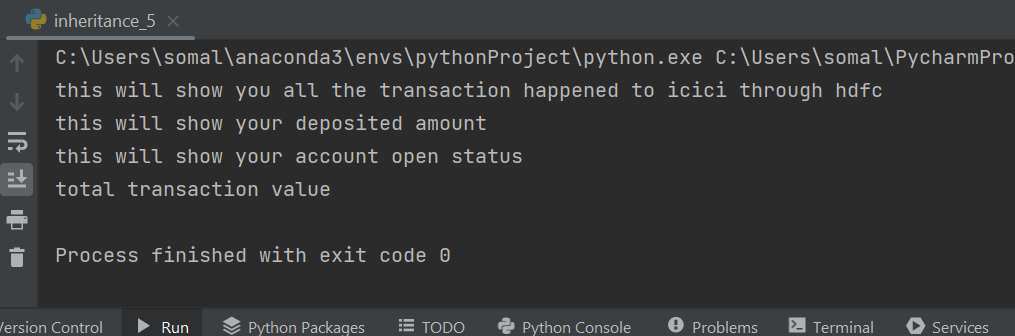
**Note:**

Here in this case the parent class ‘bank’ passed inside child class ‘HDFC\_bank’ so all the properties inside the parent class can avail inside child class and again ‘HDFC\_bank’ passed inside ‘icici’ class that means inside ‘icici’ class we can avail all the facilities inside ‘bank’ and ‘HDFC\_bank’ class. This is called multilevel Inheritance.

**Multiple Inheritance:**

#multiple inheritance  
class bank: #parent class  
 def transaction(self):  
 print("total transaction value")  
 def account\_opening(self):  
 print("this will show your account open status")  
 def deposit(self):  
 print("this will show your deposited amount")  
  
class HDFC\_bank(): #child class  
 def hdfc\_to\_icici(self):  
 print("this will show you all the transaction happened to icici through hdfc")  
  
class icici(bank,HDFC\_bank): #child class  
 pass  
  
i=icici()  
i.hdfc\_to\_icici()  
i.deposit()  
i.account\_opening()  
i.transaction()

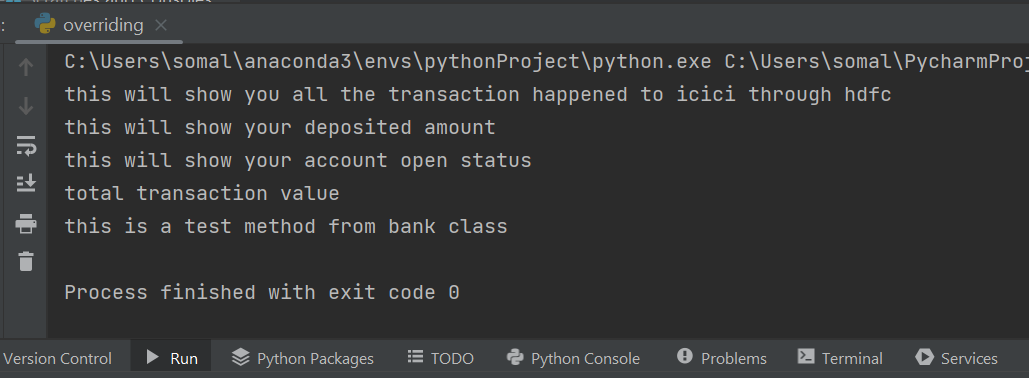
**Output:**

****

**While doing multiple inheritance when two methods having same name while calling it which one will be printing:**

#multiple inheritance  
class bank: #parent class  
 def transaction(self):  
 print("total transaction value")  
 def account\_opening(self):  
 print("this will show your account open status")  
 def deposit(self):  
 print("this will show your deposited amount")  
 def test(self):  
 print("this is a test method from bank class")  
  
class HDFC\_bank(): #child class  
 def hdfc\_to\_icici(self):  
 print("this will show you all the transaction happened to icici through hdfc")  
 def test(self):  
 print("this is a test method from HDFC\_bank class")  
  
class icici(bank,HDFC\_bank): #child class  
 pass  
  
i=icici()  
i.hdfc\_to\_icici()  
i.deposit()  
i.account\_opening()  
i.transaction()  
i.test()

**Output:**

****

**Notes:**

While doing multiple inheritance when two methods having same name here it is test() method, while calling the method inside ‘icici class’ it is printing the method that is declared inside ‘bank class’ because while declaring ‘icici class’ and passing multiple arguments i.e ‘class icici(bank,HDFC\_bank):’ ‘bank’ is passed as 1st arguments if I will pass ‘HDFC\_bank’ as 1st arguments then it will print the methods inside ‘HDFC\_bank class’. So, the arguments which we will pass 1st will return as an output while calling it.

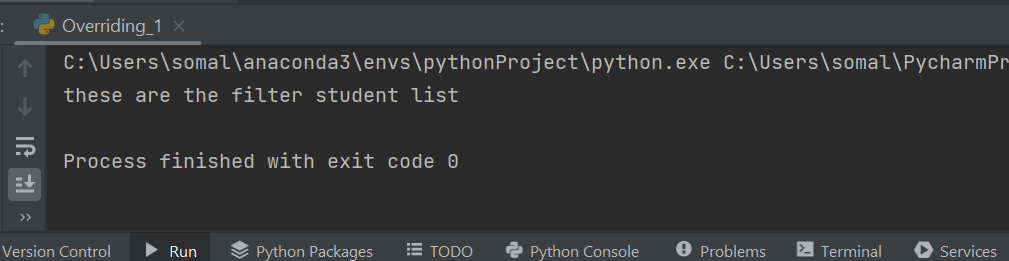
**Overriding:**

Method overriding is a feature of object-oriented programming languages where the subclass or child class can provide the program with specific characteristics or a specific implementation process of data provided that are already defined in the parent class or superclass.

How method overriding works:

#method overriding  
class ineuron:  
 def student(self):  
 print("print the details of all students")  
 def achivers(self):  
 print("prit the list of all the achivers")  
 def hall\_of\_fame(self):  
 print("print everyone from all of fame")  
  
class ineuron\_vision(ineuron):  
 def student(self):  
 print("these are the filter student list")  
  
iv=ineuron\_vision()  
iv.student()

Output:

****

Notes:

Here inside “class ineuron” we have declared “student” method to print the details of all students and later inside the child class again declared “student” method to print the filter student list when we will create an object inside child class i.e ‘IV’ and call ‘student’ method.

It will return the print statement written inside child class of ‘student’ method.

**Data Abstraction:**

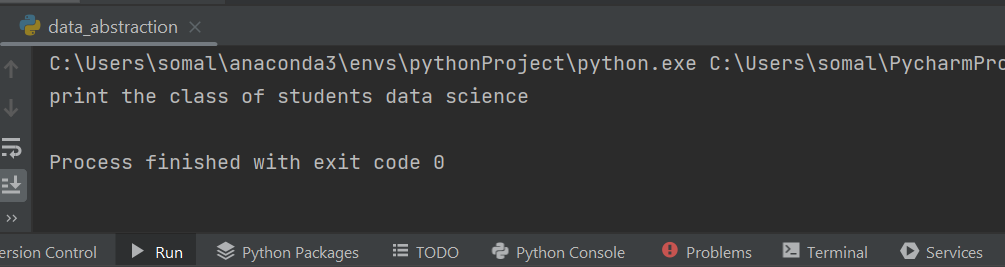
Abstraction in python is defined as a process of handling complexity by hiding unnecessary information from the user. This is one of the core concepts of object-oriented programming (OOP) languages.

**E.g.** we are using so many methods inside the list like copy, extend, remove, reverse but we don’t know the real implementation of it we only know where to use and how to use it. This is also a concept abstraction.

How Abstraction Works:

#data abstraction  
class ineuron:  
 \_\_students ="data science"  
  
 def students(self):  
 print("print the class of students", ineuron.\_\_students)  
  
i=ineuron()  
i.students()  
i.\_ineuron\_\_students

Output:



Note:

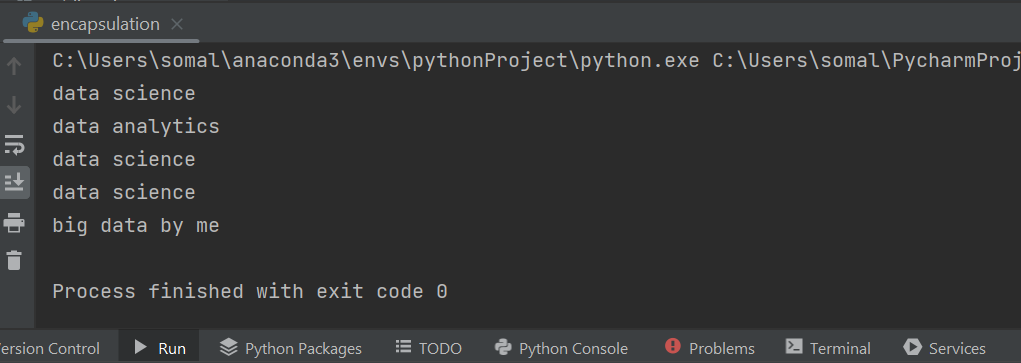
As \_\_student is a private variable we cannot access the data writing i. \_\_students. That means we are not allowing to access the data directly. That is called abstraction. To access the data we can write like this i.\_ineuron\_\_students

**Encapsulation:**

Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.

#Encapsulation  
class ineuron:  
 def \_\_init\_\_(self):  
 self.students1 ="data science"  
 def students(self):  
 print(self.students1)  
  
i=ineuron()  
i.students()  
i.students1 = "data analytics"  
i.students()  
  
class ineuron1:  
 def \_\_init\_\_(self):  
 self.\_\_students1 ="data science"  
 def students(self):  
 print(self.\_\_students1)  
  
 def student\_change(self):  
 self.\_\_students1="big data by me"  
  
i1=ineuron1()  
i1.students()  
i1.\_\_students1 = "big data"  
i1.students()  
i1.student\_change()  
i1.students()

Output:



**Notes:**

Here Inside ‘ineuron’ class public variable has been declared ‘self. students1 ="data science"’ and we can able to reassign the value directly through the object but in case ‘ineuron1’ class private variable has been declared self. \_\_students1 ="data science" when we try to reassign the value from "data science" to "big data" through object it doesn’t work but when we try to reassign the value through the method it changed from “data science" to "big data by me".

Difference between abstraction and encapsulation.

Abstraction: not able to **access** the data directly

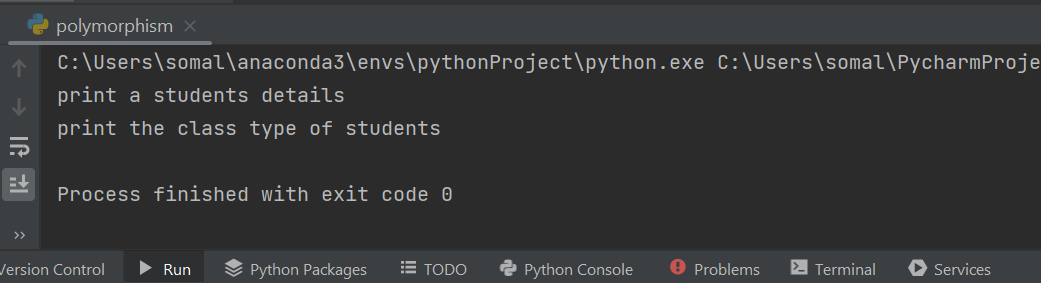
Encapsulation: when we try to do **modification**

**Polymorphism:**

Polymorphism defines the ability to take different forms. Polymorphism in Python allows us to define methods in the child class with the same name as defined in their parent class.

class ineuron:  
 def students(self):  
 print("print a students details")  
  
class class\_type:  
 def students(self):  
 print("print the class type of students")  
  
def ineuron\_external(a):  
 a.students()  
i=ineuron()  
j=class\_type()  
ineuron\_external(i)  
ineuron\_external(j)

Output:



Note:

Here we have created a common function/common interface through which we can access different methods inside different classes. So this common function behave differently when we call for different methods.

**EXCEPTION HANDLING**

Exception handling allows you to separate error-handling code from normal code. An exception is a Python object which represents an error. As with code comments, exceptions helps you to remind yourself of what the program expects. It clarifies the code and enhances readability.

* Introduction To Exception Handling
* Exception Handling Keywords
* Exception Handling Syntax
* Handling Multiple Exceptions
* Handling All Exceptions
* Using Exception Object
* Getting Details of Exception
* Raising An Exception
* Using finally Block
* Creating User Defined Exceptions

**What Is an Exception?**

Exception are errors that occur at runtime. In other words, if our program encounters an abnormal situation during its execution, it raises an exception.

For example, the statement a=10/0 will generate an exception because Python has no way to solve division by 0

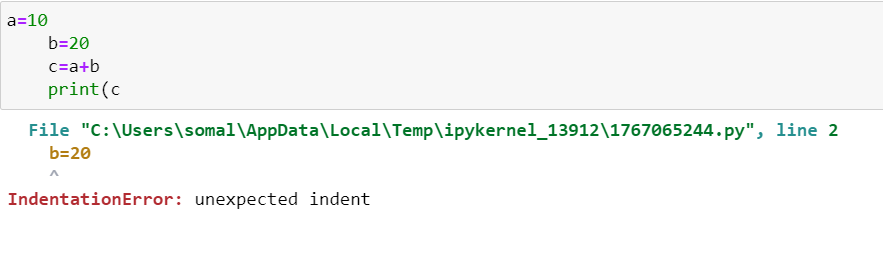
* When we talk about exception handling, we generally deal with runtime error.

Generally, 2 Types of error i.e.

* Compile time Error (syntax error)

It is a syntactical error.

e.g.



* Runtime error

It is a logical error.

e.g.



To handle those error, we do exception handling

Source code 🡪 Compiler 🡪 Bytecode (machine code)🡪 Interpreter

**What Python Does When An Exception Occurs ?**

Whenever an exception occurs, Python does 2 things:

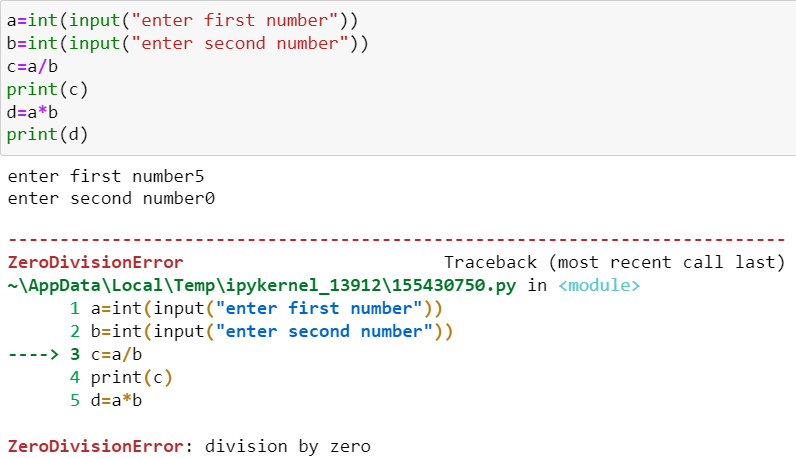
* It immediately terminates the code
* It displays the error message related to the exception in a technical way

Both the steps taken by Python cannot be considered user friendly because even if a statement generates exception, still other parts of the program must get a chance to run.

The error message must be simpler for the user to understand

**Example: Why Exception handling is Required?**

* My further code execution will stop.

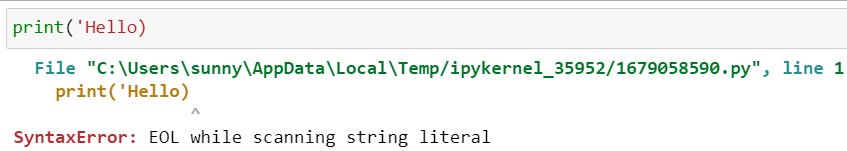


Here 5/0 can not be possible but 5\*0=0 as 5/0 throwing an exception so 2nd part of the code also not executed.

* This is not user friendly.

When there is an error in our code it is throwing a big error msg which is not user friendly when real implementation is exist.

**Different Types of Error:**



Note how we get a Syntax Error, with the further description that it was an End of Line Error (EOL) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding of these various error types will help you debug your code much faster.

This type of error and description is known as an Exception. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal.

| **Exception** | **description** |
| --- | --- |
| Exception | Base class of exception. All other exception derived from this class. |
| ArithmeticError | Base class for those exceptions that are raised for arithmetic or numeric errors. |
| ZeroDivisionError | Raised when division or modulo operation is zero |
| ModuleNotFoundError | Raised by import when imported module could not be located |
| KeyError | Raised when a mapping (dictionary) key is not found in keys of a dictionary. |
| MemoryError | Raised when an operation runs out of memory |

You can check out the full list of built-in exceptions [here](https://docs.python.org/3.8/library/exceptions.html). Now, let's learn how to handle errors and exceptions in our own code.

In python, There are two popular saying for code styles:-

1) LBYL

2) EAFP

### LBYL

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code, if key in mapping: return mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

For this, you can refer in python documentation:-

\*\* <https://docs.python.org/3/glossary.html> \*\*

In a simple language, we first check what we are going to do. For example, if we want to check if a file is available before trying to write:

**if** filename:

**with** open()..

**...**.

### EAFP

Easier to ask for forgiveness than permission.

This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style common to many other languages such as C.

In a simple language,EAFP is like we first write our code so that it performs and executes first, and then we will take care of the consequences if it doesn't work. That means we try running some code, expecting it to work, but if it failes then will handle it in exception in except block.

**try**:

**with** open(filename,'r').. *# First executes*

**...**.

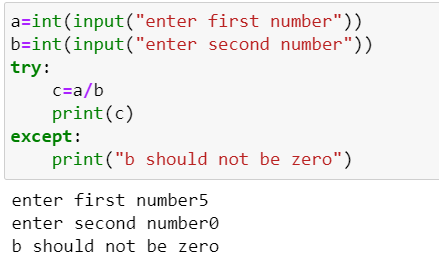
except:

*# handling exception*

**Exception Handling Keywords**

Python provides 5 keywords to perform Exception Handling:

* Try
* Except
* Else
* Raise
* Finally



* **Try and Except:**

The basic terminology and syntax used to handle errors in Python is the try and except statements. The code which can cause an exception to occur is put in the try block and the handling of the exception are the implemented in the except block of code. The syntax form is:

try:

You do your operations here...

...

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

...

else:

If there is no exception then execute this block.

* **finally**

The finally: Block of code will always be run regardless if there was an exception in the try code block. The syntax is:

try:

Code block here

...

Due to any exception, this code may be skipped!

finally:

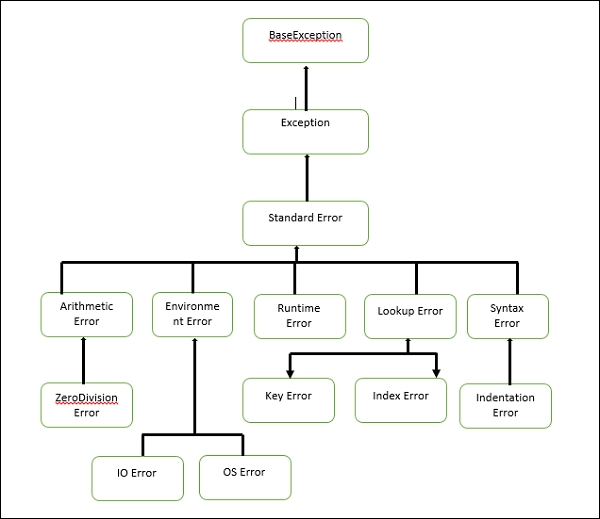
This code block would always be executed.

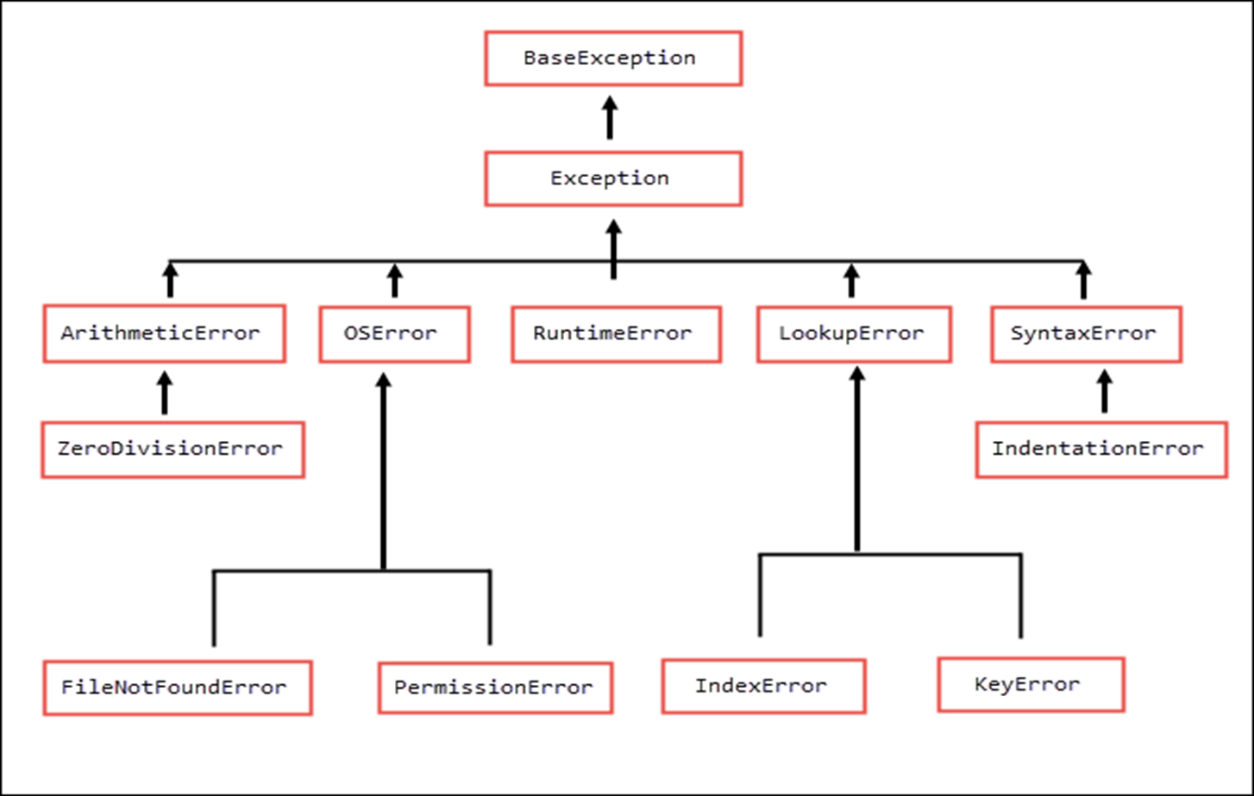
* **Raising an exception**

We can raise an exception by using raise statement.

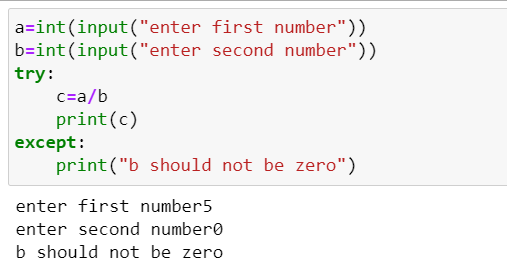
**Exception hierarchy in python:**

Python Exception hierarchy consists of various built-in exceptions. This hierarchy is used to handle various types of exceptions, as the concept of inheritance also comes into picture. In Python, all the built-in exceptions must be the instances of a class derived from Base Exception.

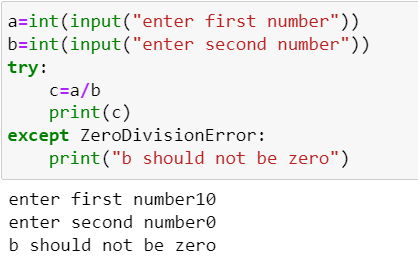
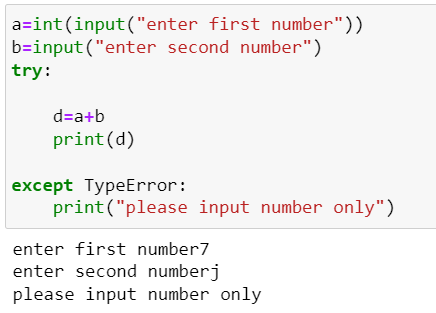




**Without giving error class name in except block:**



**We can provide Error name in except block to recognize the error more easily:**

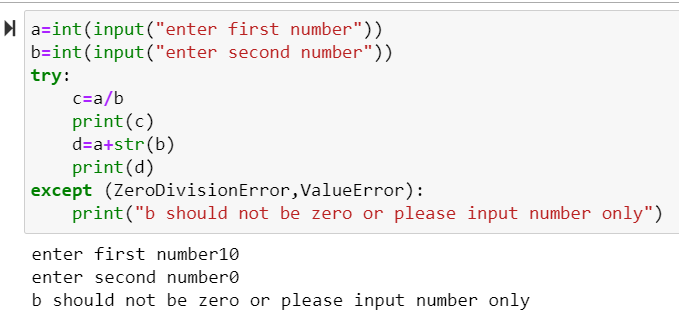
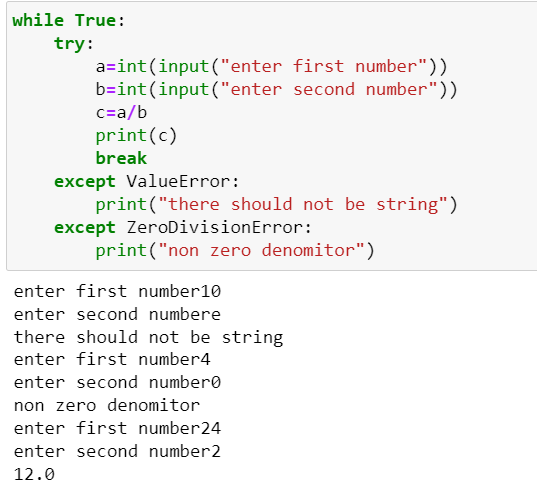
**Note:**

In the above code without writing the error class we are achieving the same result but it is always a good practice to write the error name because when in the big project we will write thousands of line of code it will be very easy to recognize which block of code is trowing which type of error.

**We can provide two Error name in except:**

we must remember that if we are handling parent and child exception classes in except clause then the parent exception must appear after child exception , otherwise child except will never get a chance to run.

For an Example in the above hierarchy **ArithmaticError** is the Parent class and **ZeroDivisionError** is the child class

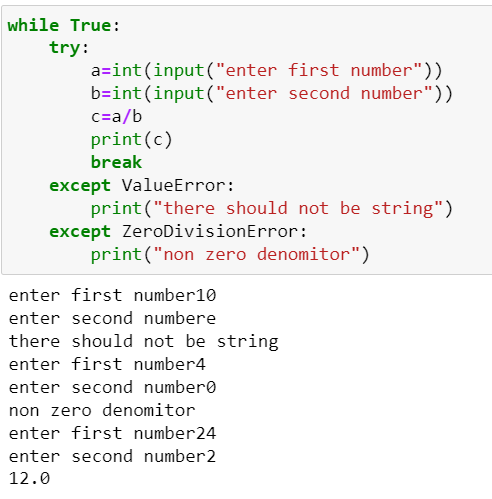
**Question:**

Write a program to ask the user to input 2 integers and calculate and print their division. Make sure your program behaves as follows:

* If the users enters a non integer value then ask him to enter only integers
* If deniminator is 0, then ask him to input non-zero denominator

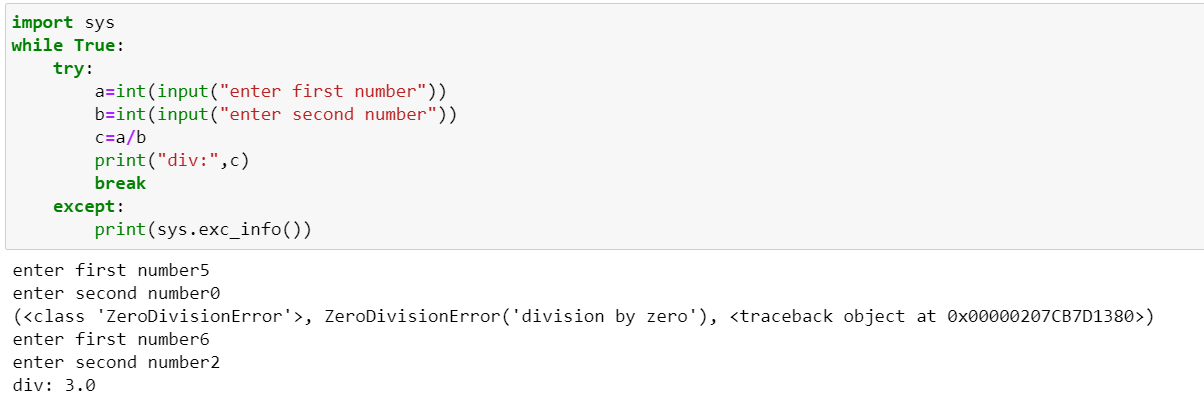
Repeat the process until correct input is given

Only if the inputs are correct then disply their division and terminate the code.

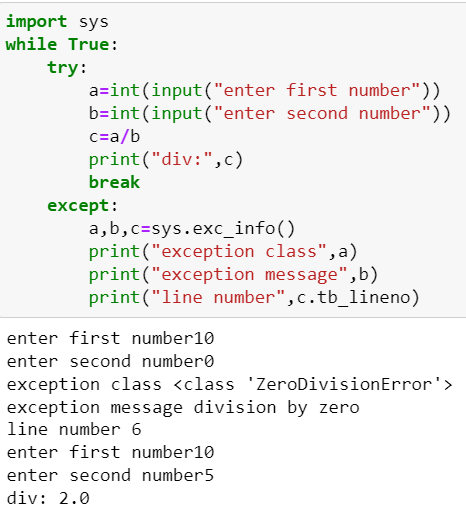


**Use of sys module and exc\_info function:**

 If the current thread is handling an exception, exc\_info returns a tuple whose three items are the class, object, and traceback for the exception. If the current thread is not handling any exception, exc\_info returns (None,None,None)



To decode the code in more easier way using the exc\_info() function:



**Use of traceback module and format\_exc function:**

This class contains a method called format\_exc( )

It is a class method of traceback class and returns complete details of the exception as a string.

**This string contains:**

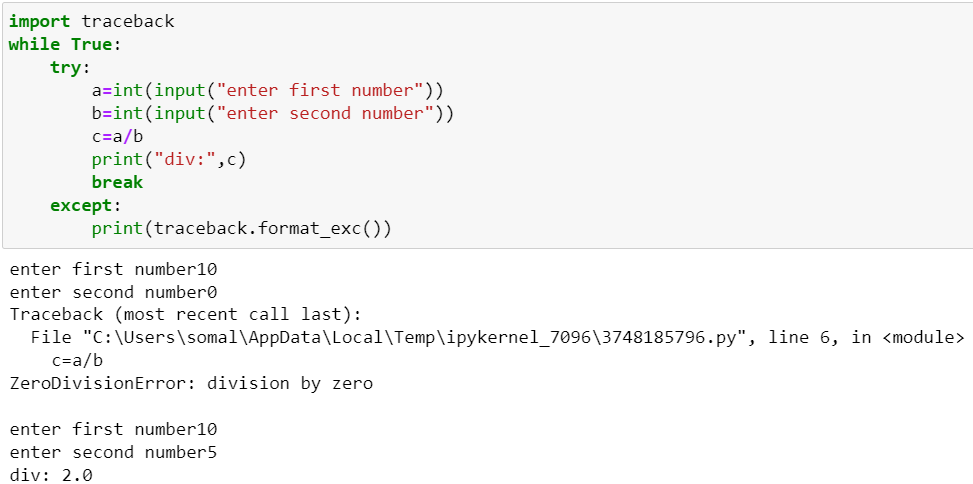
The program name in which exception occurred

Line number where exception occurred

The code which generated the exception

The name of the exception class

The message related to the exception



**raise Exception:**

We can force Python to generate an Exception using the keyword raise.

This is normally done in those situations where we want Python to throw an exception in a particular condition of our choice

Syntax: raise ExceptionClassName

raise ExceptionClassName( message )

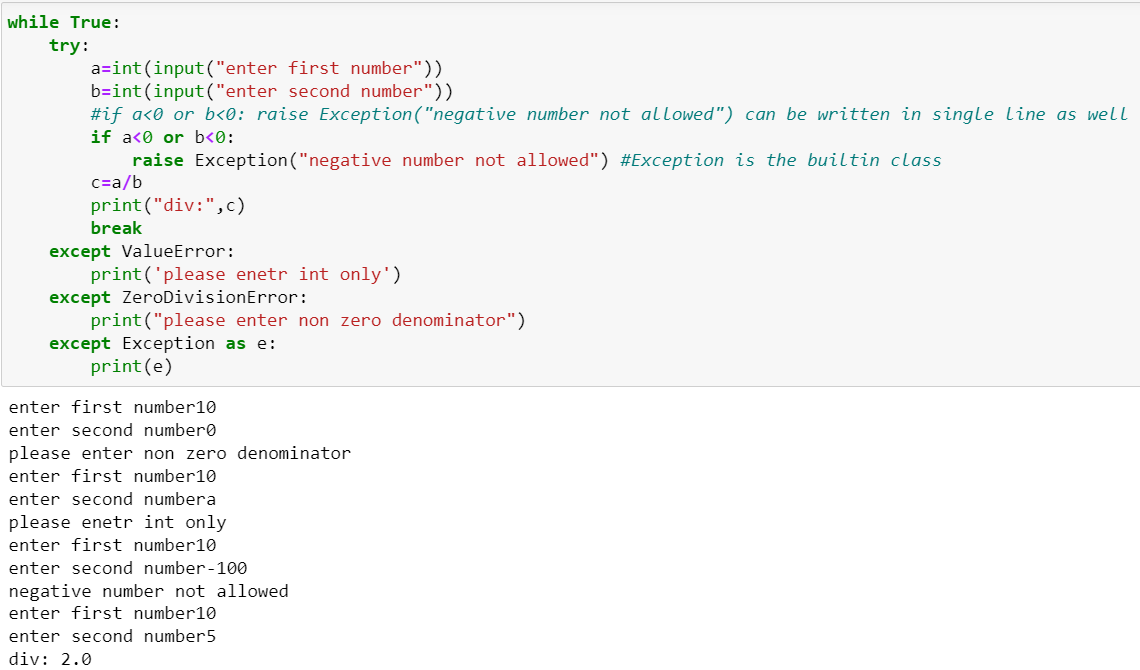
**Question:**

Write a program to ask the user to input 2 integers and calculate and print their division. Make sure your program behaves as follows:

If the user enters a non integer value then ask him to enter only integers If denominator is 0, then ask him to input non-zero denominator If any of the numbers is negative then display the message negative numbers not allowed Repeat the process until correct input is given

Only if the inputs are correct then display their division and terminate the code

using builtin class Exception:

**User Defined Exceptions:**

Python has many built-in exceptions which forces our program to output an error when something in it goes wrong.

However, sometimes we may need to create our own exceptions which will be more suitable for our purpose. Such exceptions are called User Defined Exceptions

In Python, users can define such exceptions by creating a new class.

This exception class has to be derived, either directly or indirectly, from Exception class.

Most of the built-in exceptions are also derived form this class.

Using Custom exception:

when we are trying to use custom exception we have to declare that class and Exception builtin class we have to pass in custom class(inheritance).



**Finally:**

finally will be print at any cost whether exception is there or not.

The finally: Block of code will always be run regardless if there was an exception in the try code block. The syntax is:

try:

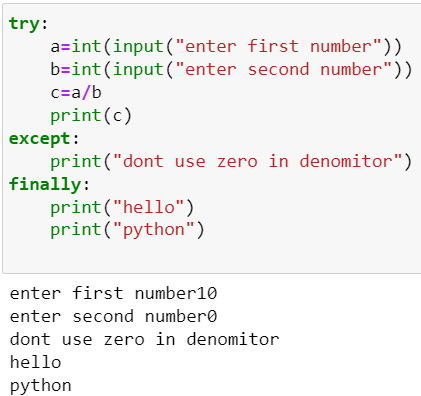
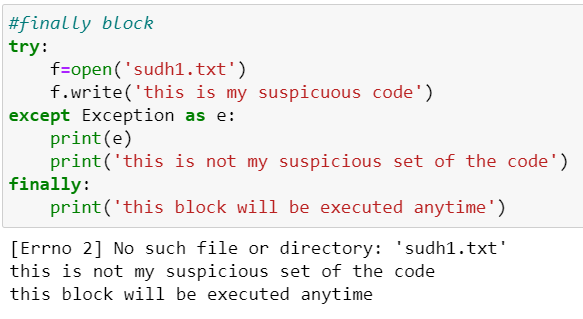
Code block here

...

Due to any exception, this code may be skipped!

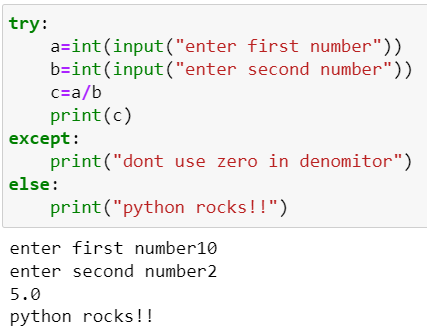
finally:

This code block would always be executed.

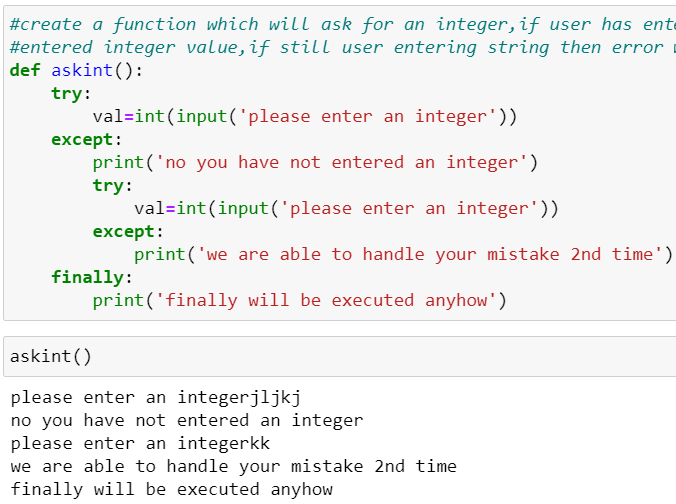
**Else:**

When there is an exception, it won’t execute else block.



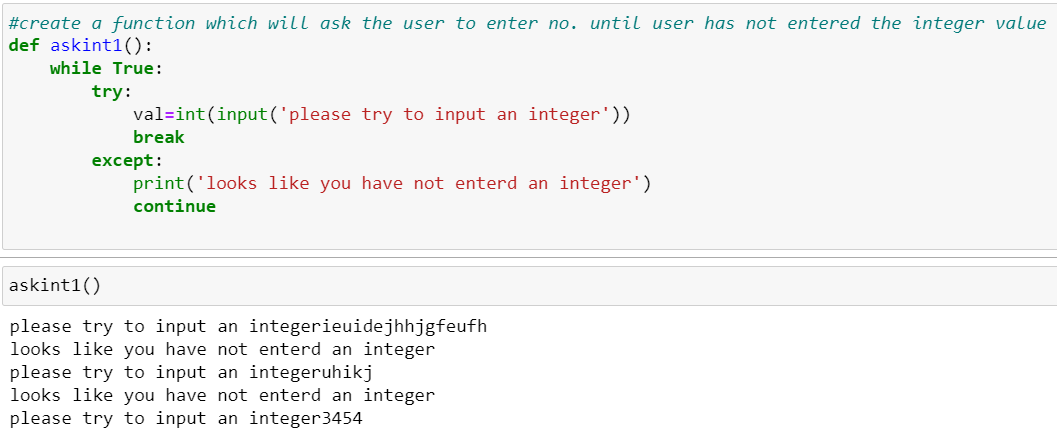
**Question:**

create a function which will ask for an integer,if user has entered a string it will give an another chnace to user to entered integer value,if still user entering string then error will be handle by except block and finally block.



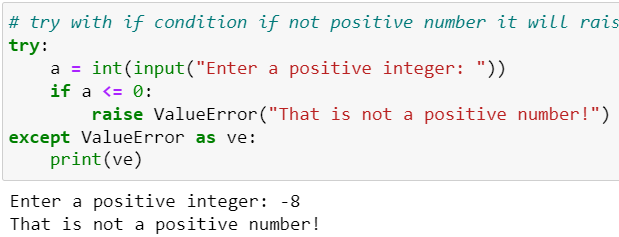
**Question:**

create a function which will ask the user to enter no. until user has not entered the integer value



**Question:**

Try with if condition if not positive number it will raise an exception using raise



### **Assertion**

Python's assert statement helps you find bugs more quickly and with less pain.

Assertions are a systematic way to check that the internal state of a program is as the programmer expected, with the goal of catching bugs. In particular, they're good for catching false assumptions that were made while writing the code, or abuse of an interface by another programmer. In addition, they can act as in-line documentation to some extent, by making the programmer's assumptions obvious. ("Explicit is better than implicit.")

For more,

**Refer:-** <https://wiki.python.org/moin/UsingAssertionsEffectively>

