Exception handling: Handling of Exception in the code is called Exception handling.

Exception:

It's run time error. Mean while error occured even if the code is right being compliled and error occured at the interpretation time is called Exception.

Ex -
```
a = 5
b = 4
Print (a/b)
```
we will have result

But, for value of b as '0' (zero).
Even our code is right for a/b we will encounter a error.

How to handle Exception:

When we have some suspicious code that may raise an exception. Then we need to place those code inside the Exception handling technique.

Different Exception handling technique:

- Try and Except statement
- Try and Else Clause
- Finally
- Raise

Try and Except statement:

Try and Except are used to catch and handle exceptions in python. Statements that might catch exceptions are kept inside try clause and statements that can handle the exception inside Except clause.

EX - when we ~~passing~~ Print (a/b) for b=0 we get error as:

ZeroDivisionError

But, if we use try and except we can display actual errors.

```
try:
    a = 4
    b = 0
    Print (a/b)
except:
    Print ('Provide non-zero denominator')
```

Result:

```
Provide non-zero denominator.
```

Thus, by using try, except we better track our Exception and can display actual error.

We can use multiple except block under one try.

Try and else statement :

when else clause is used after try-except block.

Ex-

```
a = 4
b = 0
try:
    Print (a/b)
except:
    Print ('Provide non-zero denominator')
else:
    Print ((a+b)-(a-b))
```

Finally :

when there is set of code which need to be runned anyhow even if all fails. we use finally clause after all try, except blocks.

Ex-

```
try:
    c = 4/0
    Print (c)
except :
    Print ('Non-zero denominator not present')
Finally:
    Print ('This set of code will work')
```

Raise Exception:

Raise allows programmer to force a specific exception to occur. Sole argument in raise indicates the exception to be raised.

Ex-

```
try:
    raise NameError('Hi Man') # Raise error.
except:
    Print ('It's Exception')
    raise
```

Advantage of Exception handling.

- Allow us to seperate Error - Handling code from normal code.
- Let's know of the type of error.
- Let's know of exact line of error.

Logging in python

Logging is a means of tracking events that happens when software run. It's impart because it help us have the record of programme, in case of the programme crash.

With help of logging we can identify the type / cause of crash / Problem. Thus help us solve early.

Why printing is nat prefered over logging:
Printing is used and preferred for simple script. But for complex script it won't allow writing status message to a file or any output stream. Those are provided by logging, so it's prefered over printing.

logging levels:
There are five built-in log levels:

Level	Scare
Debug	10
Info	20
Warning	30
Error	40
critical	50

When we write programme using logging. We define levels as per situation present.

Suppose we just need Error information but if we set logging level as Debug, Info we keep on getting Debug and Info and Warning information which we need nat. So, it's waste of time. So, we will set logging level as Error to save time. ~~and~~

Code in logging:

```
import logging
logging.basicConfig(filename ='xyz.log',
level = (logging.DEBUG))
```

We set our logging level here as logging. DEBUG / INFO / WARNING etc...
based on our need and information need.

Oops in Python:

Oops stands for Object - Oriented Programing is a programming Paradigm that uses objects and classes in programme. Main concept is to bind the data and the functions that work on that together as a single until so that no other part of the code can access this data.

Main concept of Oops:-

- class
- Object
- Polymarphism
- Inheritance
- Encapsulation
- Data Abstraction.

* class : User defined blue print from which objects are being created.

* Object: Objects are instances of a class.

* Inheritance: It's ability of one class to drive or inherit properties from ather class. class that derive property is called child class and class from which property are being derived are called Parent class.

Type of Inheritance:-

- Single Inheritance: Enables a derived class to inherit characteristics from a single parent class.
- Multilevel Inheritance: Enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

- Hierachical Inheritance: Enables more than one derived to inherit properties from a parent class.

- Multiple Inheritance: Enable one derived class to inherit properties from more than one base class.

Ex-

```
class Person( ):     # Parent class.
        def __init__(self, name, contact):
            self.name = name.
            self.cont = contact.
        def display (self):
            Print (self.name)
            Print (self.cont)
        def detail (self):
            Print ('Name is { }'.format(self.name))
            Print ('contact is { }'.format(self.cont))
  ┌─ Class employee (Person):  # child class
Single   def __init__(self, name, contact, salary):
Inheritence  self.salary = salary.
        ┌─ def detail(self):
Hierachical     Print ('salary is { }'.formal (self.salary))
Inheritence.    Print ('Name is{ }'.formal(self.name))
   # creating instance of class.
P1 = employee ('Raj', '98353 6xxxx', '8LPA')
P1. display()
P1. detail()
```

* Polymorphism :

when same code work differently according to the input condition.

EX- a = 4 , b = 3

a+ b = 7

a = 'AMRIT' , b = 'RAJ'

a+b = 'AMRIT RAJ'

As shown ~~in~~ above same function a+b perform different as per the available variable a, b.

* Encapsulation: It describes idea of wrapping data and the methods that work on data with in one unit. Step is to restrict accessing variables and methods directly and can prevent accidental modification of data. If need to change only object variables be access by object method. These variables are also called private variables.

```
EX-   class Name :
          def --init-- (self):
              self.a = 'Amrit Raj'
              ~~self.~~
              self._A = 'AMRIT RAJ'
```

```
Class Surname (Name):
    def __init__(Self):
        Name.__init__(Self)
        Print (Self._A)


Ob1 = Name()
Print (ob1.a)  --Result-->  'Amrit Raj'
Print (ob1._A) --Result-->  Error
```

In above code we restrict direct access by using encapsulation. So getting error.

* **Data Abstraction :**

When we want to hide code details from user or don't want to give out sensitive parts of our code implementation then we use data Abstraction.