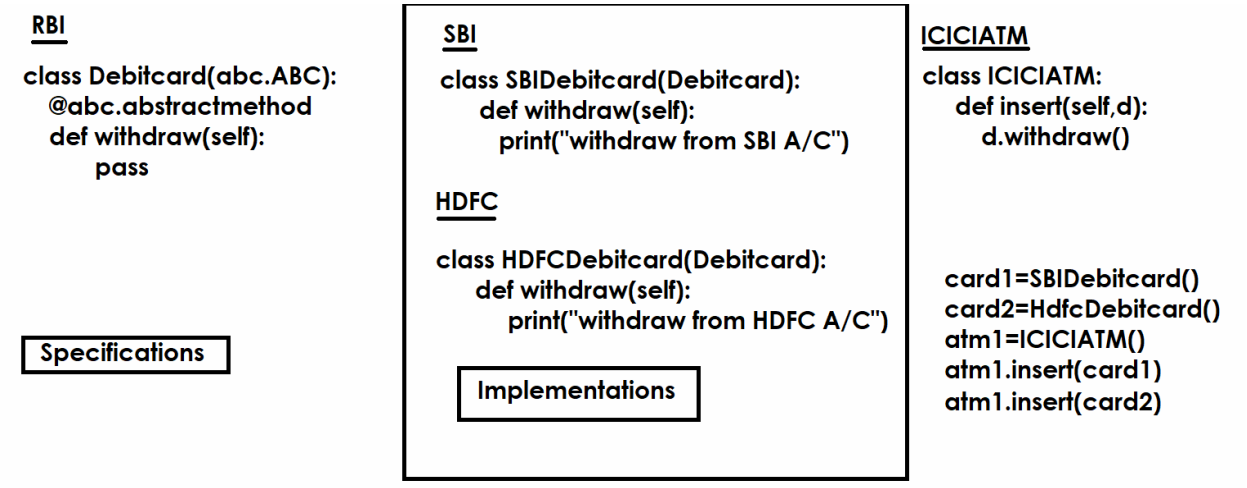
Abstraction
Hiding implementation details from caller by giving only specifications. This abstraction is achieved using abstract classes and abstract methods.



Example:

```
import abc
# TRAI
class Sim(abc.ABC):
    @abc.abstractmethod
    def connect(self):
        pass

class JioSim(Sim):
    def connect(self):
        print("connect to JIO network")

class AirtelSim(Sim):
    def connect(self):
        print("connect to Airtel network")

class Mobile:
    def insert(self,s):
        s.connect()


s1=JioSim()
s2=AirtelSim()
```

```
mobile1=Mobile()
mobile1.insert(s1)
mobile1.insert(s2)
```

Output

```
connect to JIO network
connect to Airtel network
```

Interface

An interface is pure abstract class which consist only abstract methods. This abstract class does not have non abstract methods.

Example:

```
import abc
#PYTHON
class DBDriver(abc.ABC):
    @abc.abstractmethod
    def connect(self):
        pass

#oracle
class OracleDriver(DBDriver):
    def connect(self):
        print("connect to oracle database")
#mysql
class MySQLDriver(DBDriver):
    def connect(self):
        print("connect to mysql database")

def connect(d):
    d.connect()


connect(OracleDriver())
connect(MySQLDriver())
```

Output

```
connect to oracle database
connect to mysql database
```

Duck Typing or Dynamic Typing
Duck typing in Python is a programming concept where the type or class of an object is less important than the methods it implements. This concept is derived from the saying, "If it walks like a duck and quacks like a duck, then it must be a duck.

In Python, this means that if an object has the necessary methods and properties, it can be used in a function or context that expects a certain type of object, regardless of its actual class. Instead of checking the type of an object, Python checks whether the object has the required methods and attributes. This approach promotes flexibility and code reusability

Encapsulation
Class
Object
Inheritance
Polymorphism
Abstraction
Composition
Aggregation

Nested Class or Inner classes
A class within class is called nested class (OR) defining class inside class is called nested class.

Why nested classes?
1. Hiding implementations of a class
2. Modularity

Nested classes are two types
1. Member class
2. Local class

Member class
Class is a collection of members and these members can be,
1. Data members (variables)
2. Member functions (methods)
3. Member classes

Syntax:

```
class <class-name>: # Outer class
     class <member-class-name>: # Inner class
          variables
          methods
    variables
    methods
```

Example:

```
class A:
    class __B: # private Member class
        def __init__(self):
            print("B object is created")
    class C: # public Member class
        def __init__(self):
            print("C object is created")
    def __init__(self):
        print("A object is Created")
        objb=A.__B()


#objb=A.__B()
objc=A.C()
obja=A()
```

Output

```
C object is created
A object is Created
B object is created
```

Example:

```
class Person:
    class __Address:
        def __init__(self):
            self.__street=None
            self.__city=None
        def read_address(self):
            self.__street=input("Street :")
            self.__city=input("City :")
```

```
        def print_address(self):
            print(f'Street {self.__street}')
            print(f'City {self.__city}')
    def __init__(self):
        self.__name=None
        self.__add=Person.__Address()
    def read_details(self):
        self.__name=input("Name :")
        self.__add.read_address()
    def print_details(self):
        print(f'Name {self.__name}')
        self.__add.print_address()


p1=Person()
p1.read_details()
p1.print_details()
```

Output

```
Name :nit
Street :ameerpet
City :hyd
Name nit
Street ameerpet
City hyd
```

Local class

A class defined inside method is called local class. This class object is created within method but not outside method.

Syntax:

```
class <class-name>:
    def method-name(self):
            class local-class-name:
                variables
                 methods
```

Example:

```
class A:
```

```
    def m1(self):
        print("inside m1 method of A")
        class B: # Local class
            def m2(self):
                print("inside m2 method of B")
        objb=B()
        objb.m2()



obja=A()
obja.m1()
```

Output

```
inside m1 method of A
inside m2 method of B
```

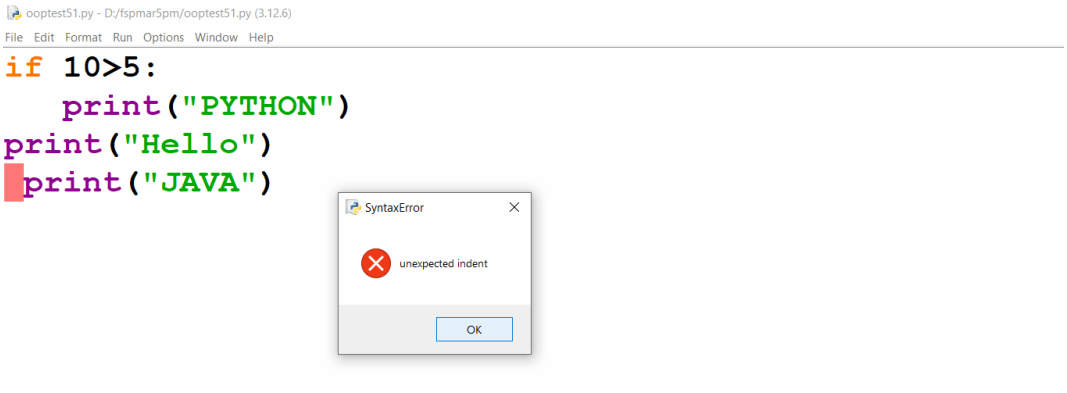
Exception Handling or Error Handling

Types of errors

1. Compile time errors
2. Logical errors
3. Runtime Errors

Compile time errors

Compiler generated errors are called compile time errors. All syntax errors are called compile time errors. If there is compile time error, program is not executed. It must be rectified by programmer in order to execute the program.



Logical Error

Logic is nothing set of instructions used to solve given problem (OR) it is procedure used to solve given problem. If there is logical error within program, program generates wrong result or output.

Example:

```
# Adding two numbers

n1=int(input("Input First Number :"))
n2=int(input("Input Second Number :"))
n3=n1-n2
print(f'Sum of {n1} and {n2} is {n3}')
```

Output

```
Input First Number :10
Input Second Number :20
Sum of 10 and 20 is -10
```

Example:

```
# Max of two numbers

n1=int(input("Input First Number :"))
n2=int(input("Input Second Number :"))
if n1>n2:
    print(f'n2={n2} is max')
else:
    print(f'n1={n1} is max')
```

Output

```
Input First Number :10
```

Input Second Number :20
n1=10 is max

Logical errors must be rectified by programmer in order to generate correct result.

Runtime Error
An error which occurs during execution of program is called runtime error. Runtime error occurs because of wrong input given to program.

If there is a runtime error, program execution is terminated. To avoid abnormal termination of program we use error handlers or exception handlers.

Exception is a runtime error.
Exception is an error which occurs during runtime.

Why exception handlers or error handlers?

1. To avoid abnormal termination of program
2. To convert technical error messages into user defined error messages

Keywords used to handler runtime errors or exception handling

1. try
2. except
3. finally
4. raise
5. assert