```
print(json.dumps(x, indent=4,
                 sort_keys=True))
```

```
x = '{"name":"John"
y = json.loads(x)
print(y["age"])
```

06/04/2022

* EXCEPTIONS:

Errors → logical errors
Errors → syntax errors

Exceptions → Synchronous — can be handled by program
Exceptions → Asynchronous — cannot be handled through program

* EXCEPTION HANDLING:

→ by it raise exceptions

* It handle synchronous exceptions

try and except blocks.

SYNTAX

```
try:
    statements where exception may occur
except exception:
    code to handle the exception
```

```
num=int(input("Enter numerator"))
den = int(input("Enter denominator"))
q = num/den
print(q)
try:
    q = num/den
    print(q)
except DivisionbyZero:
    print("Denominator should not be zero")
```

* Except block is called an exception handler

* If there are no exceptions, after the try block is executed, the code after the [try-except block] is executed as it is (ie rest of the program.

* The exception raised in try block should be same as the name given at except statement

* A single try block can have multiple except blocks:

Types of errors/exceptions:
- ZeroDivisionError
- ValueError

```
* try:
    x = int(input("Enter a no:"))
    y = int(input("Enter a no:"))
    q = x/y
    print(q)
  except ZeroDivisionError:  if y=0
    print("Mathematical error")
  except ValueError:  if x/y = character
    print("Enter a no. only:")
  else:
    print("No exception")
```

* When an exception arises in the try block; it checks for the same type of error in each except block.
Only one except block is executed.

* We can have except block without any name. This block handles any kind of exception

* Generic exception handler is a default except block which should be at the end only

```
except:  ---> any kind of exception
    print("Generic exception handler")
```

* We can also have "else" block along with try-except blocks.
If try block is executed perfectly; else part will be executed
If corresponding exception block is executed; else part will not be executed.

10/08/2022

- An exception is an error which occurs during execution
- An exception is an event which occurs during execution of program and disturbs the normal flow of program instructions
- Exception can be handled by using try & except block.
- The statements which may rise an exception are placed inside a try block. and the code that handles exceptions are written in except block.
- Python allows to have multiple except block for a single try block.

SYNTAX:

```
try:
    statements
except name:
    statements
except name:
    statements
except
    print("generic exception handler")
else:
    statements
```

⇒ Multiple Exceptions in single block:

* except (exceptionname1, name2):
    statements

```
try:
    n = int(input())
    P = int(input())
    print(n ** p)
except (KeyboardInterrupt, ValueError, TypeError):
    print("Enter a valid integer")
```

* Raise an Exception:

```
try:
    n = int(input())
    P = int(input())
    print(n**p)
    raise TypeError("Hello")  — Instantiating exception
except (TypeError) as t:
    print("Enter a valid integer:")
    print(type(t)) ; print(t.args)
print("End of the program")
```

O/P:

```
2
3
8
<class 'TypeError'>
('hello',)
end of the program
```

* Handling exceptions in Invoked functions:

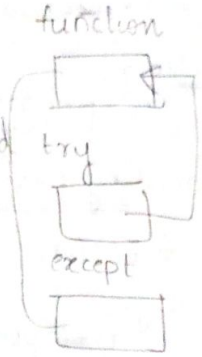
```
def func1():
    try:
        n = input()
        P = input()
        print(n**P)
    except —:

func1()
print("End of program")
```

```
def func1(n,d):
    print(n/d)

try:
    func1(4,2)
    func1(2,0)  ⟶ if except block is not specified:
except ZeroDivisionError:   there will be an error.
    print("zero cannot be the Den")

finally:
    print("finally")
```



* "finally" block should be after the except block.
When we have "finally", we shouldn't use else.
* Irrespective of error occurred or not "finally" block will definitely execute.

* USER-DEFINED EXCEPTION:

```
class AgeError(Exception):
    def display(self):
        print("Age should be >18")

n = int(input("Enter age"))
if (n<18):
    raise AgeError
else:
    print("Vote")
except AgeError as A:
    A.display()
```

Saraya is a good girl

* Assertion:

```
try:
    assert (n>18),("Error"):
    print("Vote")
except AgeError as a:
    a.display()
except AssertionError as a:
    print(a.args)
```

11/08/2022

* Re-raising the exception:

```
try:
    raise NameError
except NameError:
    print("Variable in not defined")
    raise
print("End")
```

```
* try:
    b = a + 10
    print(b)
except NameError:
```

```
print("Variable is not defined")
~~print(2+"10")~~ a=int(input())
                 print(a+10)
print("End")
```

O/P:

~~not~~ Variable is not defined
enter 25
35
end

```
* def divide(x,y)
    try:
        print(x//y)
    except ZeroDivisionError:
        print("y is 0")
    else:
        print("No exception")
    finally:
        print("Always executes")
divide(4,2)
divide(4,0)
```

O/P.
2.
no exception
always executes
y is 0
always executes.

```
* f=open("file1.txt","w")
try:
    f.write("hello")
finally:
    f.close()
```

* Nested try-except blocks:

```
try:
    f=open("file1.txt","r")
    try:
        print(f.read())
    except IOError:
        print("Error")
    else:
        print("No exception")
    finally:
        f.close()
except FileNotFoundError:
    print("No such file")
```

hello
no exception

* Assertion:

Want to raise an exception based on a condition.
↳ When the condition fails; an assertion exception is raised.

```
* a=int(input("enter"))
  assert (a>0), ("a is zero (or) negative")
  print("a=", a)
```

- An assertion is a basic check that can be turned on or off when the program is being tested.
- It is a conditional raise.
- Using assert statement, an expression is

given and result of the expression is false then assertion error

- Assertions are usually placed at the start of the function, to check for valid input and after a function call to check for valid output

SYNTAX:

assert expression[, argument/message]

*→ ~~Write a program to convert take temper~~

*→ Write a program to accept Celsius temperature ~~Fahrenheit~~. Calculate temperature in Fahrenheit and raise an assertion if F≤32

```
c = int(input("Enter temperature in celsius:"))
f = c*9/5 + 32
print("Temp in Fahrenheit = ", f)
assert f≤32, "Temperature invalid"
print(f).
```

13/08/2022

UNIT-V

CLASSES & OBJECTS:

Object oriented concepts:

- class → user-defined datatype
- object
- polymorphism ~~S1=Stud~~
- Inheritance
- Abstraction - hiding details.
- Encapsulation

* class Student:
  - class variables
  - class methods

```
s1 = student()
s2 = student()
```
→ Object (or) instance

→ By defining a class, we are achieving encapsulation. i.e. variables & methods are placed together.

→ A class creates a new type/datatype and object is an instance (variable) of the class. Classes provides a blue print or a template using which objects are created.

→ Syntax:

Defining a class

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

* Creating Objects:

:- Once a class is defined; we can create object of that class

: The object can access the class variables and methods using '.' operator

* Syntax:

```
Objectname = classname()

classname.classmembername()
Objectname.classmembername().
```

```
* class Branch:
    bran = "CSE"  ... class variable.
b1 = Branch()  → instantiating.
print(b1.bran
```

→ "self" should be the first parameter passed to a method

```
* class student:
      count = 0  → class variables
      def read(self):
          name = input("Enter your name:")      } Instance variables
          self.rno = int(input("Enter rno:"))
          student.count += 1
      def display(self):
          print(self.name)                      } Instance variables
          print(self.rno)

object1 = student()
object1.read()
object1.display()
```

Self will passes the present object to the method.

```
* s1 = student()
  s1.read()
  print(s1.name)
```

→ Syntax:

Defining a class:

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

*Creating Objects:

- Once a class is defined: we can create object of that class.

The object can access the class variables and methods using '.' operator.

*Syntax:

```
Objectname = classname()
```

~~classname.classmembername()~~

```
objectname.classmembername()
```

* class Branch:

object bran = "CSE" ... class variable.

```
b1 = Branch()  ⟶ instantiating
print(b1.bran
```

→ "self" should be the first parameter passed to a method. 'self' points to the particular object

* class student:
  count = 0 ⟶ class variables. ⟶ only one copy.
  def read(self):
    self.name = input("Enter your name:") } Instance variables
    self.rno = int(input("Enter rno:"))
    student.count += 1
  def display(self):
    print(self.name) } Instance variables
    print(self.rno)

* Instance variables have its own copy for each individual object.

```
object1 = student()
object1.read()
object1.display()
```

Self will passes the present object to the method.

```
* s1 = student()
  s1.read()
  print(s1.name)
```

16/08/2022

```
* x = 10
  print(type(x))
```

O/P:
<class 'int'>

→ variable of a class: object/instance of a class.

* Accessing the class variable using object name.
  - objectname.class variable

* class circle:
    def getradius(self, r):
        self.radius = r
    count = 0
    def area(self):
        return (3.14 * self.r * self.radius)

obj1 = circle()
obj1.getradius(10)
print(obj1.area())

* CONSTRUCTOR: __init__() → initializing the instance variables.
• special method

* class Rectangle:
    def __init__(self):
        self.length = 0
        self.breadth = 0
r1 = Rectangle() → methods are implicitly called.
print("area=", r1.length * r1.breadth)

class Rectangle:
    def __init__(self, l, b):
        self.length = 0
        self.breadth = 0
r2 = Rectangle(5, 6).

print("area=", r2.length * r2.breadth)

* def __init__(self, length, breadth):
        self.length = length
        length += 2
        print(length) → o/p: 5, 7

* garbage collector: → free the space allocated if the variable is not used anymore.

* Destructor: __del__()

⇒ to execute destructor, we should use 'del' command: If we want to execute whenever we want.

* The variables which are accessible outside the class are called public variables/methods
* The variables which are not accessible outside the class are called private variables/methods

* class circle:
    def getradius(self, r):
        self.__radius = r
c1 = circle()
c1.getradius(10)
print(c1.__radius) ✗

print(a.__circle__radius)

* class ABC():
    class_var = 0
    def __init__(self, var):
        ABC.class_var += 1
        self.var = var
        print("Object value:", self.__var)
        print("Value of class variable:", ABC.class_var)
    def __del__(self):
        ABC.class_var -= 1
        print("object %d is deleted:", %self.var)

obj1 = ABC(10)
obj2 = ABC(20)
obj3 = ABC(30)
del(obj1)
del(obj2)
del(obj3)

    def square(self):
        print("Square:", self.__var * self.__var)
    def __display(self):
        print(self.__var)

obj1.square()
obj1._ABC__display()

* Method inside another method.

def square(self):
    self.__display()
    print("Square:", self._var * self._var)

* obj.var = 20

If variable is existing in the class; the value is modified.

If variable is not existing; a new variable is created in the class.

* class ABC():
    def __init__(self, var):
        self.var = var
    def display(self):
        print("var=", self.var)

obj = ABC(10)
obj.display()
obj.new_var = 20
print("New var =", obj.new_var)
obj.new_var = 30
print("Modified =", obj.new_var)
del obj.new_var
print("Deletion =", obj.new_var)

* Documentation:

class ABC:
    """This is a docstring. I have created new class"""

description related to the class

pass.

print(ABC.__doc__).

- If we don't want to write anything in the class; we can mention pass

* __repr__ : returns the content of particular object.

*→

```
class ABC:
    def __init__(self,name,var):
        self.name=name
        self.var=var
    def __repr__(self):
        return repr(self.var)
    def __len__(self):
        return len(self.name)
    def __cmp__(self,obj):        → pass an object also
        return self.var-obj.var

obj=ABC("abcdef",10)
print("Value stored in object is:",repr(obj))
print("The length of name stored in object is:",len(obj))
obj1=ABC("asdfg",1)
val=obj.__cmp__(obj1)
if val==0:
    print("Equal")
elif val==1:
    print("1st value is lesser")
else:
    print("2nd value is lesser")
```

17/08/2022

*

```
class number:
    evens=[]
    odds=[]
    def __init__(self,num):
        self.num=num
        if num%2==0:
            Number.evens.append(num)
        else:
            Number.odds.append(num)
N1=Number(21)
N2=Number(32)
N3=Number(43)
N4=Number(54)
N5=Number(65)
print("Even numbers:",Numbers.even)
print("Odd numbers:",Number.odds)
```

Output:

Even numbers: [32,54]
Odd numbers: [21,43,65]

* Calling a class method from another class method:

```
class ABC:
    def __init__(self,var):
        self.var=var
    def display(self):
        print("Var is", self.var)
    def add_2(self):
        self.var+=2
        self.display()

obj=ABC(10)
obj.add_2()
```

* BUILT-IN CLASS ATTRIBUTES:

__dict__ → dictionary with instance variables.
__doc__ → prints documentation string.
__name__ → name of the obj belongs
__module__ → name of the module To which class belongs
__bases__

Eg: print(obj.__dict__) ⇒ O/P: {var:12}.
print(obj.__doc__) ⇒ O/P: None.
print(obj.__name__) ⇒ O/P: ABC
print(obj.__module__) ⇒ O/P: __main__

* Garbage Collection: Destroying objects.

→ Python's automatic garbage collector deletes all the objects automatically that are no longer needed and that have gone out of scope to free the memory space. The process by which python periodically re-claims the unwanted memory is known as garbage collection.

Eg:
```
var1=10
var2=var1 → referencing to same value
var3=[var2]
var2=50 → different reference
var3[0]=-1 → "
del var1
```

* INHERITANCE:

OOP CONCEPTS:

- Class
- Object
- Abstraction → providing the required details & hiding others
- Inheritance
- Polymorphism
- Encapsulation: binding the data & methods together

→ Inheriting the features from the old/existing class to the new class and add other variables & methods.

- We can reuse the existing class.
- Time required from implementation / compilation will be decreased.

old/existing class: Base class/super class/ parent class.
new/created class: derived/sub class/child class.

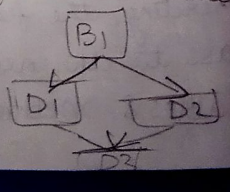
* There is (is-a) relationship established b/w base class & derived class.

* ACCESS SPECIFIERS:

- Public : Accessable anywhere.
- Private [(--) → prefix]
- Inheritance protected [(_) → prefix]
  - parent class will share certain data only to the child class.

⇒ TYPES OF INHERITANCE:

- Single → single base & derived class
- Multiple → multiple base class with single derived class
- Multi-level → B1 → D1 → D2 →
- Hierichal → single base class & multiple derived classes
- Hybrid → combination of any of the 2 inheritance problem
  - diamond problem
    - combination of multiple & hierichal



SIVANI PYTHON PROGRAMMING

19/08/2022

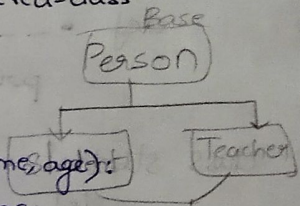
* Private variables cannot be inherited.
* The technique of creating a new class from an existing class is called inheritance
- The old/existing class is known as base/parent class
- The new class is known as derived/child class.

* Every class in python is the sub class of "object class".

SYNTAX:

```
class derived class (base class):
    body-of-the-derived-class.
```

Base class will be generalized



Ex:

```
class person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)
class student(person):
    def welcome(self):
        print("This is student class")
```

* Object created from subclass can access the details from both base & sub classes

* Method overridding:

Subclass method may override the

base class if the methods given in both base & derived classes are having same name.

* To use the super class method with same name can be executed only by

```
super().m1()
        └→ method name.
```

```
* class person:
    def __init__(sey):
        print("Base constructor")

  class student:
      def __init__(sey):
          super().__init__(sey)
          print("derived constr")

  s1 = student()
```

O/P:
Base constructor
Derived constr.

```
* class person:
    def __init__(sey):
        print("Base constru")

  class student(person):
      pass

  s1 = student()
```

MRO – Method Resolution Order.

- If there is no constructor in derived ; acc. to MRO, checks for constructor in super class.

* example:

```
* class person:
    def __init__(sey, name, age):
        sey.name = name
        sey.age = age
    def display(sey):
        print(sey.name)
        print(sey.age)

  class student(person):
      def __init__(sey, name, age, rno):
          super().__init__(sey, name, age)
          sey.rno = rno
      def displaydata(sey):
          sey.display()
          print(sey.rno)

  s1 = student("xyz", 30, 39)
  s1.displaydata()

  class Teacher(Person):
      def __init__(sey, name, age, exp):
          super().__init__(sey, name, age)
          sey.exp = exp
      def displaydata(sey):
          sey.display()
```

```
        print(self.exp)
t1 = Teacher("XYZ", 35, 13)
t1.displaydata()
```

→ Method overridding is the ability of a class to change the implementation of a method provided by one of its ancestors.

* MULTIPLE INHERITANCE:

In multiple inheritance a class can be derived from more than one base class.

*Syntax:

```
class base1:
    statement block
class base2:
    statement block
class derived(base1, base2):
    statement block
```

Example:

```
class base1(object):
    def __init__(self):
        print("Base1 class")
        super(base1, self).__init__()
class base2(object):
    def __init__(self):
        print("Base2 class")
class derived(base1, base2):
    pass
D = derived()
```

O/P:

Base1 class
Base2 class

* Acc. MRO: only base1 constructor is executed.

If only method derived
↓
base1
↓
base2
If constructor derived
↓
base1 → Super
↓
base2